Software Testing, Verification and Reliability

WILEY

SPECIAL ISSUE PAPER

Tracing Service Request Processing in Cloud

Yinqin Zhao[1] | Chang Liu[1] | Tao Yu[1] | Long Wang[1,2] | Xuanqing Shi[1] | Yong Yang[3] | Ying Li[3] | Zhengang Wang[4] | Dongdong Shangguan[4]

[1]REASONS Lab, Tsinghua University, Beijing, China | [2]Zhongguancun Laboratory, Beijing, China | [3]National Engineering Center of Software Engineering, Peking University, Beijing, China | [4]2012 Lab, Huawei Corporation, Dongguan, China

Correspondence: Long Wang (longwang@tsinghua.edu.cn)

Received: 5 July 2023 | **Revised:** 22 October 2024 | **Accepted:** 4 December 2024

Funding: This work has been supported by the NSFC Project of China under Grant 62132009.

Keywords: cloud | cloud tracing | distributed tracing | request execution path | service request

ABSTRACT

Nowadays, more and more IT services are being hosted on cloud systems, which render cloud systems to grow into a huge complex with millions of physical servers, multi-layer software stacks and the processing of cloud service requests across many servers and software layers. It is highly demanded for cloud service providers to have the capability of getting the knowledge on cloud service behaviour directly from the service execution instead of from people's expertise. This paper studies the problem of tracing cloud service's processing of requests across components in cloud environments and proposes cloud tracing mechanisms for this purpose. We also developed model-based studies of our proposed mechanisms for analysing certain designs of the mechanisms. The implementation of the proposed cloud tracing is deployed onto the environments of OpenStack, Kubernetes and Hadoop, and the experiments on these environments demonstrate that our mechanisms effectively trace cloud service behaviour and generate a single complete request execution path, while without our mechanisms the cloud tracing either fails to work or results in thousands of path segments. Our mechanisms have a low performance overhead (2.3%) in the experiments.

1 | Introduction

More and more IT services, including those critical for humans' living and life, are being hosted on cloud systems nowadays. There are a large number of servers in clouds, multi-layer software stacks run on them, and the processing of requests of cloud services span many virtual machines and/or containers across the servers and the software layers; these render the service behavior on cloud environments very difficult for people to clearly understand and carefully control. Hence, it is quite urging for cloud service providers to have the capability to acquire knowledge of the cloud service behaviour, and the capability will largely help people's understanding and management of the services in clouds. Due to the fact that new services and new releases of software are onboarding the cloud environments frequently, and many of the software are developed by third parties, even with their source code unavailable, it is highly desirable to have the capability to acquire the service behavior knowledge directly from the executions of the services themselves rather than from the expertise of software developers.

In this paper we study the problem of tracing cloud service executions for acquiring the service behaviour knowledge. In particular, we focus on *tracing cloud services processing of requests across components in cloud environments*, as most, if not all, cloud services execute in the pattern of receiving requests from clients, processing the requests and then replying to the responses to the clients. When cloud service providers want to study or understand the behavior of certain services, they can then enable tracing the selected services for a period of time

and obtain the knowledge and understanding of the services. Typically, there is no need to trace all the services all the time.

We looked into existing solutions of distributed tracing in the literature and real-world practice, including X-trace [1], Magpie [2], Google's Dapper [3], vPath [4], Facebook's Canopy [5], REPTrace [6–8], Htrace [9], Pinpoint [10] and Stardust [11]. Some of them work only on specific vendor platforms and require their specific libraries or middleware (e.g., X-trace [1], Magpie [2], Facebook's Canopy [5] and Pinpoint [10]); some require the availability of the traced applications and software's source code (e.g., Htrace [9] and Stardust [11]); vPath [4] has stringent assumptions on the patterns of components' threads and their communications and does not apply in cloud environments that involve complicated computing scenarios. REPTrace [6–8] is a recently proposed technology that transparently captures the Request Execution Path (REP) of services and Hadoop jobs, a complete end-to-end tracing path, and it supports quite comprehensive computing scenarios.

The current design of REPTrace just works for a relatively static computing platform where the complete set of components to be traced is predefined and there will be little changes to them. But services and components in clouds keep changing, and new services and components keep getting onboarded. Cloud service providers may want to trace those newly onboarded or changed services to understand their behaviours and disable tracing them after getting the knowledge. Moreover, the current design assumes all components involving the processing of target services be traced for generating the complete tracing path. As cloud service providers selectively enable and disable tracing certain services with components shared by the services having their tracing either enabled or disabled, and incremental deployments and applications of the tracing capability onto different parts of clouds are inevitable due to the large scale of clouds, we have to deal with the situation that the components involved in a service's request processing may be partially traced (i.e., some are traced, and others are not). These two limitations of the current design make the direct application of the REPTrace technology unsuitable for cloud environments.

So, we have to address the following two challenges in order to trace the processing of service requests in clouds: (i) frequent dynamic changes and (ii) situations with partial tracing where handling communications with components not being traced is required. Besides that, we also need to verify that our designs are reliable in real scenarios. This paper designs the following mechanisms, built on top of the REPTrace technology, for addressing the two cloud-specific challenges and study the reliability of the mechanisms:

- A registration mechanism that dynamically identifies and bookkeeps the set of components being traced in the cloud system;
- A boundary handling mechanism that supports the communications between traced components and not traced ones (components will crash upon such circumstances if REPTrace is directly applied) based on the registration mechanism above;
- A path-mending algorithm that mends request execution path segments, which result from the communications with not traced components, into a single complete path. REPTrace assumes all components involved in the processing of a service request are being traced; it is able to link all communications and executions of the components in this request processing into a single unseparated path. However, when not traced components are involved in the request processing, the single path cannot be generated and instead, a bunch of REP segments will be generated with the single path broken at the tracing boundary components;

This paper is an expanded version of our conference paper [12], and beyond the conference paper, here we provide thorough studies on the registry cache synchronization, an event buffering optimization, a scalability study and much more extensive deployments and experiments of the proposed mechanisms in various platforms. These make this paper a more complete presentation of our cloud tracing approach. Specifically, the new contributions of this paper compared with the conference paper include (i) studies on the synchronization of registry cache, a key issue in our approach design (Section 4), and we constructed a SAN model and ran simulations in the studies; (ii) the optimization of the REPTrace with event buffering before REPAgent sending events to the Central Generator (Section 3.5), and the experiments that measure the performance improvement brought by the optimization (Section 6.5); (iii) the discussion on the scalability of our approach in cloud platforms (Section 3.6); (iv) new deployments and experiments of tracing services/programs hosted in the platforms of Kubernetes and Hadoop (Sections 6.2 and 6.3); and (v) new experiments that compare the performance of our mechanisms in the extended REPTrace with that of the original REPTrace and new experiments that measure the performance overhead breakdown of the Component Registry (Section 6.4). What we propose in this paper is a general tracing technology that traces the service request processing behaviour in cloud environments that are highly dynamic and contain many partial tracing situations. As far as we know, currently there are no general tracing technologies that address the two challenges that are inherent in tracing service request behaviour in large-scale cloud systems. The contributions of this paper are as follows:

- We designed mechanisms to trace the processing of service requests in clouds by addressing the challenges of supporting highly dynamic computing environments and handling partial tracing situations. These comprise a registration mechanism, a boundary handling mechanism and a path-mending algorithm as described above.
- We modelled the scenario where the proposed mechanism runs in the cloud environment and studied the impact of different user-set parameters on the reliability of the proposed mechanism. Simulation results show that the proposed mechanism can guarantee high reliability with a small overhead (error probability is only 0.0096).
- We implemented the proposed mechanisms in a Linux system and deployed them onto the OpenStack, Kubernetes

[13] and Hadoop [14] environments to trace services and programs.

- We conducted experiments on OpenStack, Kubernetes and Hadoop to evaluate our proposed mechanisms, with the following results: (i) With the component registration and boundary handling mechanisms, traced services run normally and traces are successfully collected in all environments. Without these mechanisms, the services fail or crash. (ii) In all experiments, without path mending, many REP segments are produced (e.g., 2962 segments in OpenStack). With path mending, the algorithm successfully links all segments into a single complete REP. (iii) The additional overhead is low. Compared with the original REPTrace, overhead increases from 2.58% to 2.95% in Kubernetes and from 5.57% to 8.36% in Hadoop. (iv) Event buffering reduces communication overhead by 91.45%, dropping from 51.39 s (without buffering) to 4.39 s.

2 | Background: REPTrace

Our cloud tracing mechanisms are built on top of the REPTrace technology [6]. Here, we give a brief introduction to this technology. REP is a directed acyclic graph that expresses the complete execution procedure of an individual request being processed by all components among a distributed system. The path consists of many events that represent the executions and communications in the request processing and links these events into a single path according to their causal or time sequence relationships.

Figure 1 shows the architecture of REPTrace deployed in a distributed system. The distributed system consists of many nodes (physical servers, virtual machines or containers), and multiple services are running on these nodes. Each service may have multiple components, too. The REPAgents that exist in all processes of the traced components intercept those library calls or system calls of REPTrace's interests, generate events for these intercepted calls and send the events to the Central Generator. For example, an implementation of the REPAgent exploits the LD_PRELOAD mechanism [15] and is a dynamic library that is loaded into the address space of the traced process. The REPAgent library intercepts LIBC calls of the process. The intercepted calls include those communication related ones like *send(), recv(), read(), write()*, etc. (*read()* and *write()* are used in Linux for socket communications), as well as thread/process manipulation ones and other ones the users of the REPTrace may be interested in.

Note that the use of LD_PRELOAD works in tracing various platforms and services written in different languages. This is because the fundamental libraries of most languages, such as Python libraries and Java VMs, invoke system calls via the LIBC library to avoid the complications of directly invoking system calls using assembly languages. Moreover, there is no need to coordinate the execution of the LD_PRELOAD-based interceptions in different processes/components, because the scope of the LIBC call interception by the LD_PRELOAD mechanism is only within a process. More clarifications on this point are available in Section 5.

Then the Central Generator links the received events according to their specific causal or time sequence relationships, which are comprehensively analyzed and presented in depth and detail in [6], into a complete REP. Specifically, two attributes are created for the event linking purpose: MSG_ID and MSG_CTX_ID. MSG_ID is created to link the sending and receiving events of the same network message. REPAgent on the sender side generates and inserts an MSG_ID into the message so that the REPAgent on the receiver side can extract it from the message and put it into the receiving event. Then the Central Generator knows the sending and receiving events should pair up as they have the same MSG_ID values. Note that MSG_ID is added as part of the payload of messages, not in the headers of the network protocols (e.g., IP header and TCP header). MSG_CTX_ID is created to identify different parts of a thread's execution that deal with different requests, that is, it annotates which request the thread is currently processing when an event is generated. Then an event linking algorithm [6] makes use of the two attributes to link all the events into the REP.

3 | Proposed Mechanisms

REPTrace works by intercepting certain LIBC calls of a process (using the LD_PRELOAD mechanism [15]; see Sections 2 and 5 for details), recording events of these LIBC call invocations and linking the events into a complete request execution

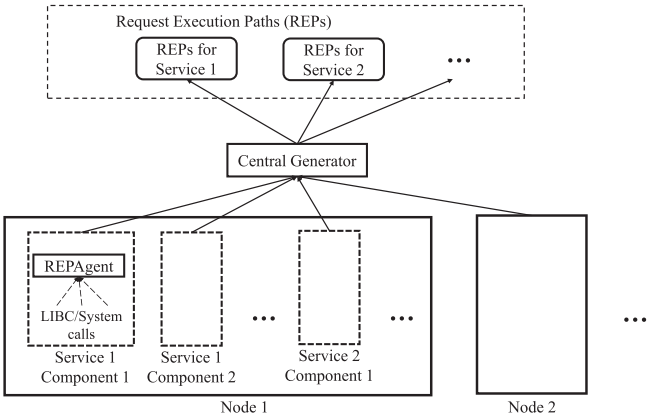


FIGURE 1 | REPTrace overview.

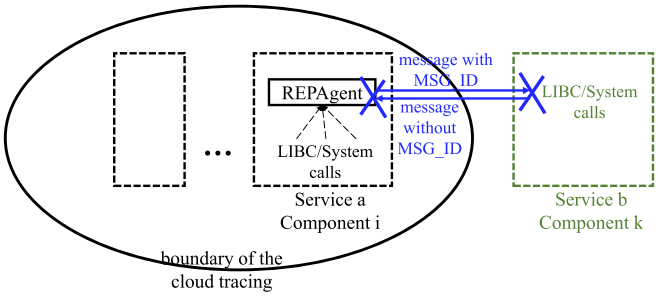


FIGURE 2 | REPTrace is unable to handle communications with not traced components.

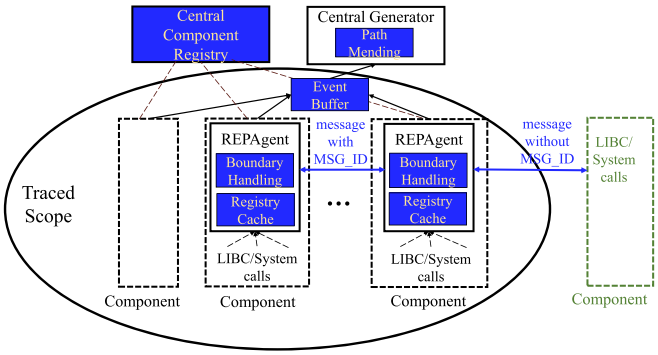


FIGURE 3 | Design of the proposed mechanisms in the REPTrace architecture to support communications between Traced Scope and its outside.

path. REPTrace works well if all components involved in the processing of the service request are traced. When REPTrace is applied to the cloud environment, there are quite complicated interdependencies among components in the cloud and the environment is under constant changes of services and components. As a result, it is impossible to deploy REPTrace onto all components of the cloud at one time and then keep the deployment not changed anymore. There are definitely situations where traced components communicate with not traced components, as shown in Figure 2.

In this situation, the REPAgent in the traced component (*Service a Component i* inside the boundary of the cloud tracing in Figure 2) sends a message with the MSG_ID attribute to, or receives a message without the MSG_ID from, a component that is not traced (*Service b Component k* in Figure 2). There is no REPAgent in the not traced component; component k does not expect the MSG_ID part in the message, and component *i* expects MSG_ID in the message but it is absent. The unexpected message format at the receiving side may lead to consequences of message refusal, message retransmission, disconnection, process crash or even fail-silent violation (in rare cases).

In order to deal with such situations, we propose three mechanisms: Central Component Registry, boundary handling and path mending. Figure 3 illustrates the design of the proposed mechanisms in the REPTrace architecture, with the new mechanisms highlighted in blue. The set of all the traced components in the cloud system is called *Traced Scope*. So, the boundary of the cloud tracing illustrated in Figure 2 is the boundary of the Traced Scope in Figure 3.

Algorithm 1 Central Component Registry

```
1: Upon initialization:
2:     Create comm_point_list, an empty list of < IP Address, P ortNumber >
3:     // or create an initial hardcoded list of known communication points of traced components
4: Upon receiving REGISTER:
5:     // < ip_addr, port_num > is the received parameter
6:     Look up < ip_addr, port_num > in comm_point_list
7:     if not found
8:         put < ip_addr, port_num > in comm_point_list
9: Upon receiving QUERY:
10:    // < ip_addr, port_num > is the received parameter
11:    Look up < ip_addr, port_num > in comm_point_list
12:    return the result (found or not found)
13: Upon receiving DEREGISTER:
14:    // < ip_addr, port_num > is the received parameter
15:    Look up < ip_addr, port_num > in comm_point_list
16:    if found
17:        remove it from comm_point_list
```

Before details of the mechanisms are given below, we would like to mention that the proposed mechanisms are general and support general distributed systems, though our motivation stems from our encounters with the described problems in cloud environments. These problems exist in general distributed systems and are particularly typical and frequent in clouds. So, we can apply the proposed mechanisms in general distributed systems to deal with the partial tracing situations.

3.1 | Security and Privacy

Security and privacy is not a focus of this paper. But note that (i) REPTrace and the proposed enhancements do not capture or store the payload of traced packets and (ii) cloud providers need to ask the service providers (i.e., service owners) for consent before tracing their services. Of course, service providers can enable tracing their own services. Service providers can also install their own Central Generator and Central Component Registry so that all traced information is transmitted to their own Central Generator, if they want to understand their services' behaviour but do not want the cloud providers to trace their services. This paper focuses on the tracing technology and privacy-related regulations are beyond this paper's scope.

3.2 | Component Registry

A registration mechanism called *Component Registry* is designed to dynamically register and maintain which components are traced in the cloud system. The registration mechanism consists of two parts: *Central Component Registry*, a stand-alone process, and *Registry Cache* in all the REPAgents (see Figure 3).

Algorithm 1 and 2 describes the algorithm executed by the Central Component Registry and the REPAgent for the component registration task.

3.2.1 | Central Component Registry

The Central Component Registry is a stand-alone process that can be placed on any node of the cloud. It keeps the list of the communication points of all the traced components/processes in the cloud, in terms of the <IP address, Port number> tuples.

When the Central Component Registry is launched, it creates an empty global list comm_point_list (Line 2 in Algorithm 1). Of course, if the user has an initial list of traced components' communication points, he/she can assign the list to comm_point_list during the initialization.

Then Central Component Registry waits for requests from REPAgents, which may send three types of requests, REGISTER, QUERY and DEREGISTER, together with the <IP address, Port number> tuple. When Central Component Registry receives the REGISTER request, it registers the tuple into comm_point_list; when it receives the QUERY request, it returns the query result to the REPAgent; when it receives the DEREGISTER request, it removes the relevant entry from comm_point_list.

3.2.2 | Registry Cache

The REPAgent of each traced component process maintains a local list of communication points, *local_comm_point_list*, that the REPAgent knows are associated with a traced or a not traced process (Line 2 in Algorithm 2). Note that the local list only contains part of the global list comm_point_list and is a cache of the registry information to improve the performance of the tracing so that there is no need to do a remote lookup in the Central Component Registry for every data transmission.

When REPAgent is initialized upon the first interception of a traced call in the process, the REPAgent creates the empty local_comm_point_list (Line 2). From the algorithm we can

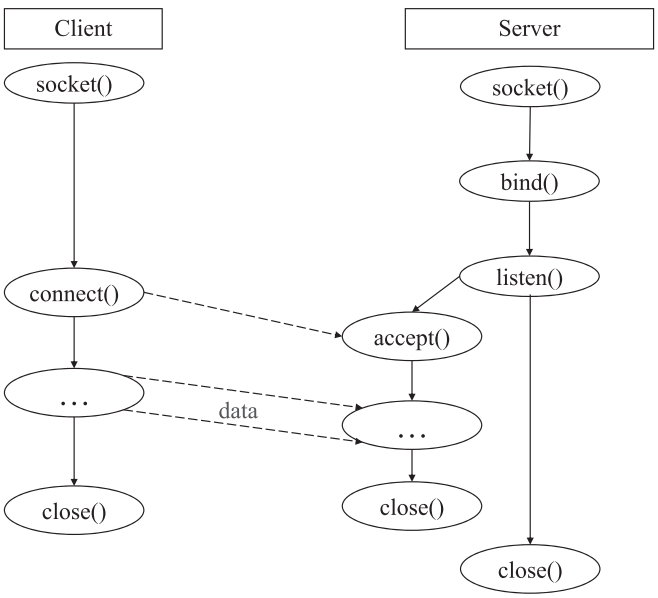


FIGURE 4 | A typical TCP communication session.

Algorithm 2 REPAgent

```
1: Upon initialization:
2:     create local_comm_point_list, an empty list of < IP Address, PortNumber, socket ID, state (traced or not traced) >
3: Upon interception of bind() or connect():
4:     get < ip_addr, port_num, socket_id > from the intercepted call's parameters
5:     send REGISTER to Central Component Registry with < ip_addr, port_num >
6:     look up < ip_addr, port_num > in local_comm_point_list
7:     if not found
8:         put < ip_addr, port_num, socket_id, traced > in local_comm_point_list
9: Upon interception of send(), recv() or equivalent calls:
10:    get < ip_addr, port_num > of remote peer from intercepted call's parameters
11:    look up < ip_addr, port_num > in local_comm_point_list
12:    if found
13:        now the state (traced or not traced) is known
14:    else
15:        send QUERY to Central Component Registry with < ip_addr, port_num >
16:        if Central Component Registry returns found
17:            put < ip_addr, port_num, traced > in local_comm_point_list
18:            now the state (traced) is known
19:        else
20:            put < ip_addr, port_num, not traced > in local_comm_point_list
21:            now the state (not traced) is known
22: Upon interception of close(): // close a socket
23:    get < ip_addr, port_num, socket_id > from the intercepted call's parameters
24:    look up < ip_addr, port_num, socket_id > in local_comm_point_list
25:    if found
26:        send DEREGISTER to Central Component Registry with < ip_addr, port_num >
27:        remove the found entry from local_comm_point_list
```

see that, besides *IP address* and *port number*, each entry of the local_comm_point_list also has two attributes *socket_id* and *state*. The *state* indicates whether this <IP address, port number> tuple is traced or not traced. As the local list is just a cache and does not have the complete information that the global list has, the absence of a tuple in the local list does not mean the tuple is not traced. So, the state attribute is used to explicitly carry the *traced* or *not traced* information. The socket_id attribute is used for the deregistration purpose, which is explained below.

To better describe the Algorithm 2 REPAgent executes (Lines 3–27), let's take a look at a typical TCP communication session illustrated in Figure 4. The server executes the sequence of socket(), bind(), listen(), accept(), a bunch of recv()/send() or other data transmission functions and two close() calls. This sequence sets up a listening Socket A and then creates another Socket B as a result of the successful acceptance of a client's connection at accept(). The first close() at the Server closes the Socket B, and the second close() closes the listening Socket A. On the client side, bind() or listen() is not needed and connect() is used instead. In order to register a new <IP address, port number> tuple at the traced server side, the REPAgent intercepts bind() or listen(), captures the tuple information and sends the tuple to the Central Component Registry in a REGISTER request. For the generality of our approach to support UDP communications, we select bind() as the interception point for sending the REGISTER request. For registering a new <IP address, port number> tuple at the traced client side, the REPAgent intercepts connect() and sends a REGISTER request with the tuple information. The REPAgent also registers the tuple in its own registry cache, local_comm_point_list, with the state of the communication point marked as *traced*. These steps are summarized in Lines 3–8. For supporting the tracing in the presence of communications across the boundary of the Traced Scope, during data transmission over network, we should check whether the remote peer is in the Traced Scope. At the interception of send(), recv() or other LIBC calls used for network transmissions like write(), read(), the REPAgent first looks up the remote peer's IP address and port number in the local list. Of course, we do a filtering so that we only handle network transmission and do not interfere with other cases like disk write/read, which are also intercepted (this filtering is not shown in Algorithm 2). If the local list contains the <IP address, port number>, the state of the remote peer, *traced* or *not traced*, is known. Otherwise, the REPAgent queries the Central Component Registry for the state via the QUERY request and updates the local list with the retrieved result by marking the state as *traced* or *not traced* accordingly (the socket ID of remote peers is not required, so '-' is filled in Lines 17 and 20). Please check out Lines 9–21 of Algorithm 2 for details.

When the REPAgent intercepts the close() call (we filter out file closing and focus on socket closing only), there are chances that the close() call closes a socket created during the accept() call, that is, socket B in Figure 4. Note that the <IP address, port number> of socket B is the same as that of socket A. If we deregister the <IP address, port number> tuple upon closing Socket B, it is incorrect behaviour because socket A is still listening at this communication point. So, we should distinguish the two types of sockets when taking care of the close() call. We use the socket_id attribute in the local list to save the IDs of the listening sockets on the server side and the connecting sockets on the client side. Then the lookup step in Line 15 of the algorithm avoids the deregistration of the communication point if the intercepted close() is the closing of accept()-associated sockets.

In certain UDP cases, the applications do not invoke bind() or connect() when initializing the IP address and the port

information of involved sockets and directly invoke calls like sendto() for data transmission. In such cases, the IP address and port information are implicitly provided. So, when handling these cases, we carefully combine the registration and query steps in Lines 4–8 and 10–21. For neatness purpose, such details are not given in the algorithm in Algorithm 2.

3.2.3 | Synchronization of Registry Cache

Without the registry cache a traced process must remotely query the Central Component Registry at every message sending or receiving, which incurs big performance overhead. But with the registry cache, there is a need to synchronize the cache state with the Central Component Registry. Here, we present a discussion of the synchronization issue.

3.2.3.1 | Are There 'Query Before Register' Scenarios?. In these scenarios, a component is traced and registering its communication point p at the Central Component Registry, but at the same time a REPAgent of another traced component is querying its local list and the Central Component Registry for p's state; then due to race condition, the query occurs before the registration, and the REPAgent thinks p is not traced, but it is traced. This means the query of p, that is, the message sending to p or receiving from p, occurs concurrently with the registration of p, that is, the port binding of p or its connection initiation (see Lines 10–21 of Algorithm 2). But this is not possible as send()/recv() sequentially follows bind() or connect(), and they never execute concurrently. So, such 'query before register' scenarios do not exist and we do not need to consider them when designing the synchronization of the Registry Cache.

3.2.3.2 | Purging Cache for 'Query After Deregister' Scenarios. In these scenarios, a server's communication point p is closed, but a client does not know this and tries to connect to p (TCP cases) or use sendto() to send data to it (UDP cases). The TCP connection attempt will fail, and the data transmission will not happen as the listening port is closed, so there is no 'query after deregister'; in the UDP transmission attempt, the client's REPAgent looks up p in its local cache and finds it is traced, so 'query after deregister' happens, but the transmission to a closed port will be detected soon by the network protocol, and it will not do big harm. So, generally speaking, 'query after deregister' does not cause trouble.

One situation where 'query after deregister' might cause trouble is related to port reuse: When a service with the communication point p terminates and a new service starts on the same machine with p's port reused, the client's connection or data transmission to the old service will be directed to the new service, which may break the new service. To avoid such situations, we should purge the cache in time, that is, remove from the REPAgents' Registry Caches those entries that are already deregistered in the Central Component Registry.

A straightforward way to do the purge is periodic cleaning. The Central Component Registry maintains a list of entries that were recently deregistered in a past time window T. All REPAgents periodically fetch this list from the Central Component Registry and remove those entries of the list from their own local lists if the entries are found. If a REPAgent does a fetch and purge at time t3 in Figure 5, the list it fetches has the entries deregistered in [t3-T, t3). If the REPAgent's last fetch and purge occurs at t1, the entries deregistered during [t1, t3-T) will get lost and not synchronized to this REPAgent's cache. So, the last fetch and purge should occur later than t3-T (like t2), and the REPAgent's purge period should be less than T. The selection of the window T is contingent upon the trade-off between the performance overhead of the periodic purge and the probability of port reuse situations in the nodes of the cloud environment.

3.2.3.3 | Proactive Cache Purging Upon Network Connection Termination. An alternative method to the periodic cache purge above is leverage the life cycle of network connections and use the termination of network connections as the trigger for cache purging. The basic idea is that the Registry Cache records not only the remote IP address and port number of a connection but also its local IP address and port number after querying the state from the Central Component Registry and then uses the termination of the connection, which is available information locally, to trigger a potential entry removal.

Specifically, we create a list of active connections in addition to local_comm_point_list during the REPAgent initialization (Line 2 in Algorithm 2). Then we add the TCP quadruple <local IP address, local port number, remote IP address, remote port number> of a new connection into the connection list after recording the remote peer's state (Lines 17 and 20). When a connection terminates and the connection's local peer is removed from local_comm_point_list (Line 27), we also remove this connection's quadruple <ip1, port1, ip2, port2> from the connection list, and, if none of the rest connection quadruples have ip2 and port2 as the remote peer, we remove the entry <ip2, port2, socket_id, state> from local_comm_point_list.

Note that this proactive purge method may remove a remote communication point even if it is not deregistered. Then, if there is another connection to this communication point, the REPAgent queries the Central Component Registry to get its state. Therefore, this alternative method has the advantage of avoiding explicit synchronization, which reduces the design/implementation complexity and is quite valuable especially when the probability of port reuse is high. The periodical purging approach requires a lot of fetching, which incurs a lot of synchronization messages. The proactive purging does not have such synchronization messages, but it has active purging messages. The overall message overhead for proactive purging could be more or less than the message overhead for periodical purging, contingent upon the situation.

FIGURE 5 | Timeline for REPAgents' periodical purge.

3.2.4 | Sharing Registry Cache Within the Same Node

The design of the Component Registry in Figure 3 shows each REPAgent has its own Registry Cache, that is, the local list of the communication points' states. Actually, the REPAgents on the same node can share their knowledge of communication points' states so that they can leverage the results other REPAgents have obtained from the Central Component Registry. So, an improvement is to put the Registry Caches of all REPAgents on the same node into a single place and share the place with all the REPAgents. We take advantage of the shared memory mechanism in the node, and all the traced component processes in the node have access to the shared memory. A mutex lock is used to coordinate these REPAgents to avoid race conditions.

3.3 | Boundary Handing

Now that we are able to distinguish which components are traced and which are not, we can handle the boundary of the Traced Scope properly. The boundary handling mechanism is designed as a function in the REPAgent, which takes care of communications between the Traced Scope and the outside components.

As shown in Algorithm 3, the boundary handling intercepts send(), recv() or other data transmission calls, captures the <remote_ip_addr, remote_port_num> tuple and checks if the remote component process is traced by executing the algorithm specified in Algorithm 2. If it is not traced, the remote peer is outside the boundary of the Traced Scope. For the purpose of event linking for the communication across the boundary, the REPAgent generates a REPTrace event, puts the current thread's MSG_CTX_ID into the event, sends it to the Central Generator and sends or receives the message without inserting MSG_ID into it or extracting MSG_ID from it. If the remote peer is traced, this data transmission is communication within the boundary and what REPTrace currently does is executed, which includes creating MSG_ID and inserting it into the to-be-transmitted message or extracting MSG_ID from the message. The procedure implements the transmission of a message with MSG_ID within the boundary and the transmission of a message without MSG_ID across the boundary illustrated in Figure 3.

3.4 | Path Mending

With the boundary handling mechanism, the tracing of cloud services can run normally without breaking the executions of cloud components when there are communications across the Traced Scope boundary. But such communications cannot be linked into a single complete REP as REPTrace assumes all communications have MSG_IDs and the linking of a *send* event and its corresponding *recv* event is based on the same MSG_ID value in the two events. Without the MSG_ID in the events associated with the communications across the boundary, the event linking algorithm of the REPTrace technology is unable to link these events, and as a result, a number of REP segments are generated which are divided at the cross-boundary communications.

We design an algorithm that mends the complete REP by linking the events associated with cross-boundary communications correctly into the single complete REP. The algorithm is presented in Algorithm 4. After the Central Generator collects the trace events from all REPAgents and performs REPTrace's event linking, it executes the path mending algorithm. Specifically, it checks all events of *send, recv* and other data transmission types and see if they have MSG_ID or not. Those events without MSG_ID were generated for cross-boundary communications.

We use the MSG_CTX_ID attribute to link these events into the REP. For such an event e, the Central Generator searches all the events and finds the event p such that p has the same MSG_CTX_ID value as e and p's timestamp is closest before e's timestamp. The event p is then the parent of the event e, that is, we add a link p → e (Lines 5 and 6 in Algorithm 4). The rationale

Algorithm 3 Boundary Handling

```
1: Upon interception of send(), recv() or equivalent calls:
2:     // < remote_ip_addr, remote_port_num > is the remote component's IP address and port number
3:     Get < remote_ip_addr, remote_port_num > from the intercepted call's parameters
4:     Check if < remote_ip_addr, remote_port_num > is traced (see the algorithm in the component registry section)
5:     If < remote_ip_addr, remote_port_num > is not traced
6:         Get < MSG_CTX_ID > from the thread context, which is maintained by REPTrace
7:         Put the < MSG_CTX_ID > into the event
8:         Send the event to the central generator
9:         Invoke the original send(), recv(), or the equivalent calls to complete the function
10:    Else // now it is traced
11:        Do what REPTrace does
```

Algorithm 4 Path-mending

```
1: Upon receiving an event e of send(), recv() or equivalent types:
2:     Check if e has a MSG_ID
3:     If e does not have a MSG_ID:
4:         // This means e is generated by the tracing boundary components
5:         Find event p from the event set with the same MSG_CTX_ID as e, where p's time_info is closest before e
6:         Set e's parent as p
```

behind the algorithm is that we do not trace the execution of the request processing out of the Traced Scope, so we create a short circuit at a boundary component process' communication with a component out of the Traced Scope. The short circuit directly connects the communication (represented by event e) with the execution of the same thread immediately ahead of the communication (represented by event p). The event p may correspond to *send*, *recv* or any other type of event. If the event e is a parent of another event q, the linking of $e \rightarrow q$ is done by REPTrace's event linking if q is not for cross-boundary communication. If q is for cross-boundary communication, the $e \rightarrow q$ linking is also done by the short circuit in the path mending algorithm. Here is an example: A component A inside the Traced Scope sends a piece of message to another component B out of the Traced Scope, and then A receives another piece of message from B. Then the A's *send* event is linked as the parent of the A's *recv* event. By this means, we short-circuit the trace path at the boundary of the Traced Scope so that the single complete REP is constructed within all the traced components.

Note that only those events carrying the same MSG_CTX_ID value have their local timestamps (i.e., *time_info*) compared in the Central Generator for determining the order among them. Those events carrying the same MSG_CTX_ID are all generated from within the same single process. So, there is no inconsistency among these events' local timestamps. The local timestamps in the events with different MSG_CTX_ID values are not compared at all in REPTrace's event linking or our path mending algorithm.

3.5 | Event Buffering

As shown in Figure 3, during REPTrace operations, REPAgent needs to send the events it has captured to the Central Generator. Then the Central Generator links these events. The REPTrace is designed to send events immediately to the Central Generator when the events are generated upon the interception of relevant calls. Such frequent message transmissions have a high communication overhead.

To minimize the communication overhead and hence improve the REPAgent performance of event transmission, an optimization is for all REPAgents to buffer their events when they are generated and transmit them to the Central Generator in batches. We implemented an event buffer in every REPAgent for this purpose. The event buffer is a continuous block of memory. Events are placed in the buffer as they are generated. When the buffer is full and there is not enough space to fill a new event, all the events in the buffer are sent to the Central Generator, then the REPAgent clears the event buffer and stores the new event in it. Our experiments show that when the buffer size is 2048 bytes, the communication overhead is reduced by about 91.45% on average (Section 6.5). So, the buffer size may be set as 2048 bytes for optimizing our extended REPTrace.

3.6 | Scalability

The cloud tracing mechanisms above were proposed for cloud service providers to obtain knowledge on cloud service behaviour directly from the service execution. As cloud systems typically have a large scale, tracing all services in the clouds all the time incurs a lot of network bandwidth and event linking operations at the Central Generator, but it is actually unnecessary. Cloud service providers typically just want to study and understand certain services at a time and do not intend to trace all services all the time. They just enable tracing selected target services for a certain period at a time.

Moreover, the proposed tracing mechanisms are also quite scalable. REPAgents and event buffers are placed on distributed nodes, so they do not have the scalability issue. Here, we discuss the scalability of the Central Generator (with path mending inside) and Central Component Registry (Central Registry for short).

3.6.1 | Network Bandwidth

A service typically consists of tens of component processes (a component is a component process in this scalability discussion), for example, the OpenStack Nova service and the Robot Shop service in our experiments have 51 and 12 components, respectively (see Sections 6.1 and 6.2). Suppose services have 50 components each by average and a component's REPAgent generates around 250 events per second on average (247.9 events/s in our experiments on the OpenStack Nova service and 8.1 events/s on the Robot Shop service). Then one service generates $50 \times 250 = 12{,}500$ events per second. In our implementation, an event has 100 bytes (40 B for the packet header and 60 B for the payload). So, we use $12{,}500 \times 100 \times 8 = 10\,Mbps$ to estimate one service's bandwidth requirement for the Central Generator by average.

Now 10 Gbps network cards are popularly deployed in clouds and data centers (routers and switches support 10 Gbps well), so Central Generator is able to support tracing 1000 services (50 K components) at the same time with a 10 Gbps network card. This should be enough for cloud service providers to selectively enable the tracing of their target services at the same time in a cloud with hundreds of thousands of components.

Central Registry maintains the registries for the currently active network connection endpoints of traced components. Central Registry processes one REGISTER, one QUERY and one DEREGISTER request for each network connection endpoint. Though the REPAgent in the connection's peer node may query this endpoint for multiple times, only the first query arrives at the Central Registry because the endpoint will then be cached by the peer node. So, Central Registry processes only three requests for one connection endpoint. Each request has 50 bytes (40 B for the packet header and 10 B for the payload), the response of QUERY has 44 B, and there is no response for the other two requests. Suppose the average birth rate of a component's network connections is 100 per second (a large server usually has multiple replicas, and a replica process may receive hundreds of, or more, connection requests per second; but most components in clouds are not such large servers and have far less connections). So, the estimate of the Central Registry's average network bandwidth requirement for one service is $(50 \times 3 + 44)$B/endpoint $\times 100$ endpoints/(comp·s) $\times 50$comp $= 970000$B/sec. Then

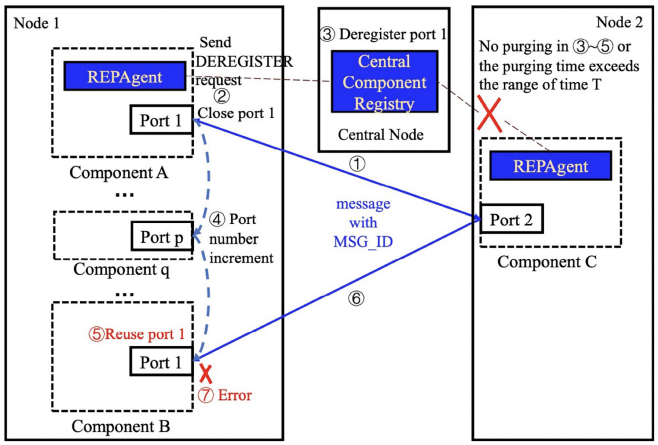


FIGURE 6 | Sending messages with MSG_ID to non-traced components during port reuse scenario.

Central Registry's network bandwidth requirement for 1000 services (50 K components) is $1000 \times 970{,}000 \times 8 = 7760$ Mbps. This can be well accommodated by modern 10 Gbps networking capability.

3.6.2 | Execution Time

Central Generator runs the event linking algorithm. The most time-consuming operation in the algorithm is to find the parent event for a given event. As we applied a hash mechanism in our implementation, this operation has O(1) time complexity and is pretty fast. According to the measurement in our experiments, the processing of 32,273 events for event linking in Central Generator takes 60.3 ms on a single CPU core of Intel(R) Xeon(R) Silver 4214R CPU at 2.40 GHz; so its execution rate is 32,273/0.0603 = 535 K events/s. 1000 services generate $12{,}500 \times 1000 = 12.5$ M events/s, which requires 12.5 M/535 K = 23.4 CPU cores. Therefore, a server of Central Generator running on 24 CPU cores supports tracing and linking events for 1000 target services at the same time (one Intel(R) Xeon(R) Silver 4214R CPU has 48 cores). So, Central Generator works well with large clouds.

Central Registry maintains the registry table by using a hash mechanism with the complexity of looking up one entry in the registry table being O(1) (see Algorithm 1) and processes a request much faster than Central Generator's processing of an event because the registration algorithm is much simpler than the event linking algorithm. According to the measurement in our experiments, the average execution time of Central Registry's processing one operation request is no more than $(0.23 + 0.60 + 0.50)/3 = 0.44$ μs (see Table 7). Central Registry processes 3reqs/endpoint × 100endpoints/(comp·s) × 50 comp = 15000reqs/s for one service; for 1000 services, it processes 15 M requests, and the execution time is no more than 15 M × 0.44 μs = 6.6 s on a single CPU core. A server of Central Registry running on seven or more CPU cores is sufficient for supporting 1000 target services at the same time. So, Central Registry also works well with large clouds.

3.6.3 | Memory Size

Central Generator does not need to save all events in the memory. After linking events into REPTrace paths, the Central Generator keeps only distinct paths and disposes of the events. When deployed for detecting anomalies, Central Generator links received events, matches them against the kept paths and then disposes of the events. So, Central Generator does not have a large demand for memory.

In Central Registry, an entry for an endpoint has 40 bytes. For the birth rate of 100/s connections and an average lifetime of 10 s for each connection (most connections are short-lived and not so long), the number of a component's active network connection endpoints is 1000, and tracing 1000 services (50 K components) requires $40 \times 1000 \times 50\text{K} = 2000$ MB. This is not a big memory size for modern servers.

4 | Studies on Registry Cache Synchronization

We discussed the synchronization of the registry cache in Section 3.2.1, which is a key issue in the design of the Component Registry. In this section, we conduct studies to further investigate this issue, which includes 'query before register' and 'query after deregister' scenarios.

4.1 | 'Query Before Register'

In 'query before register' scenarios, a query of a communication point is generated by a REPAgent that intercepts send(), recv() or equivalent calls. But the communication point is not registered yet, that is, there was no bind() or connect() call associated with this communication point before. Such scenarios violate the TCP programming pattern in which bind() sequentially precedes send() or recv() on the server side and connect() sequentially precedes send() or recv() on the client side.

Specifically, if a process executes send() or recv() before the process executes bind() or connect(), the send() or recv() LIBC call fails immediately with its return value indicating the failure. Upon such return values, the REPAgent removes the communication point from its local_comm_point_list if it had added the point into the list at Line 20 of Algorithm 2. This mandatory behaviour of TCP socket programming forces bind() of a socket to be finished in the process before recv() of the socket executes in the same process (on the server side), and forces connect() of a socket to be finished in the process before send() of the socket executes in the same process (on the client side).

To more clearly demonstrate what we discussed above, we implement a client-server application. The server does not bind to any address but instead directly calls accept() to accept incoming TCP connections, and the client does not invoke connect() to establish a connection. Then the client attempts to send data to the server. Its send() call fails immediately and the service request is unsuccessful. This experiment confirms that 'query before register' scenarios do not exist.

4.2 | 'Query After Deregister'

Different from 'query before register' scenarios, port reuse is possible, and there may exist 'query after deregister' scenarios. We construct stochastic activity network (SAN) models to study such scenarios and compute the probability of the occurrences of 'query after deregister'.

4.3 | Target Scenario

The modelled target system is shown in Figure 6. There are multiple components in each node (a node may be a physical machine or a VM). Component A and Component B are in Node 1, Component C is in Node 2, and Central Component Registry runs in the central node. REPAgents run in Components A and C, but there is no REPAgent in Component B.

We model the target system and compute the probability of 'query after deregister' occurrences from the model. Such a scenario is illustrated as the steps shown in Figure 6. The steps are explained below.

1. Component A establishes a connection with Component C (Node1.Port 1 to Node2.Port 2) and then communicates with C. Both A and C are in the traced scope and both know it.
2. A disconnects from C, and A closes its Port 1. The REPAgent of A intercepts the close() call and sends a DEREGISTER message to Central Component Registry.
3. Central Component Registry removes Node1.Port 1 from the registry.
4. New connections and communications continue to be established. New port numbers on Node 1 are allocated by the operating system for the connections. By default, port numbers are allocated by incrementing the last allocated port number. When the allocated number reaches the end of the port number range in the system, the allocation mechanism starts at the beginning of the range and searches for the next port number that is not currently in use. This port range is (32,768, 60,999) [16] in the Linux system by default.
5. After a while, Component B attempts to set up a connection with C, and the port number allocated to C is Port 1, the same port number as A's Port 1. So, Port 1 is now reused in B.
6. During the period from Step 3 to Step 5, Node 2 was not synchronized with the Central Component Registry in time. Then Node1.Port 1 is still recorded as staying in the traced state in Node 2's registry cache. As a result, B establishes a TCP connection with C and C regards B as being in the traced scope.
7. Then C communicates with B by assuming there is MSG_ID in the transmitted messages. But B is not in the traced scope, and the communication fails.

4.4 | SAN Model

We employ SANs to model the target scenario. SAN models provide a clear understanding of how systems behave and their execution policies for evaluating system performance and dependability [17]. We use Möbius as the modeling tool [18]. Möbius is a software tool for modeling complex systems. With support for multiple modeling formalisms and solution techniques, Möbius offers flexibility to represent and solve systems accurately and efficiently.

In Möbius, a blue circle represents a Place in a SANs model, an orange circle represents an Extended Place, a wider rectangle represents a Timed Activity, and a narrower rectangle represents an Instantaneous Activity. A red triangle pointing to the left represents an Input Gate which controls whether its associated activity (Timed Activity or Instantaneous Activity) is enabled or not. A black triangle pointing to the right represents an Output Gate, which allows for specifying actions when its associated activity is triggered. The arrow at the end of an arc represents the direction of the transition from one Place to the next after completing an Activity.

Our model consists of four submodels, as shown in Figure 7. Among them, submodels a, b and c model different aspects of the target scenario behaviour, and submodel d models the whole experimental process. Here, P1 indicates the Port 1 in Figure 6.

- The **port_p1_service** submodel captures the behaviour of Component A, with Port 1 open, doing data transmission, closing Port 1 and requesting Component Registry (CR) to deregister Node1.Port 1 (Steps 1, 2 and 3). The timestamp of the deregistering is recorded for use in the **exprs** submodel.
- The **opened_ports_in_node1** submodel captures the behaviour of Node 1's allocating port numbers to its hosted processes and the incrementing of the ephemeral port number within the given port range [16], as new connections and communications continue to be established in Node 1 (Steps 4 and 5). The port number incrementing is modelled in the *open_port* output gate, and the *cur_opened_port* extended place keeps the value of the currently allocated port number. When the value of the *cur_opened_port* is equal to the given Port 1 value (the *is_P1_reused* input gate does this check), the Port 1 is allocated to a new connection again, that is, is reused, and *P1_reused* is set as 1 for controlling the termination of the experiment in the *exprs* submodel.
- The **node2_sync** submodel captures the behaviour of Node 2 Registry Cache's synchronization with the Central Component Registry. The *synchronize* activity models Node 2's synchronization with the Component Registry. The synchronizations are done periodically. Some are done before the deregistering of the Port 1 (the *sync_b4_P1_dereged* instantaneous activity) until the last synchronization that updates the state of Node 1's Port 1 in Node 2's Registry Cache as not traced (the *sync_with_P1_dereged_state* instantaneous activity, which is enabled by *P1_deregistered_in_CR*). After this synchronization, the following ones are not captured in our model (*P1_deregistered_in_CR* becomes 0 after the completion of the *sync_with_P1_dereged_state* activity, which is not enabled any more).

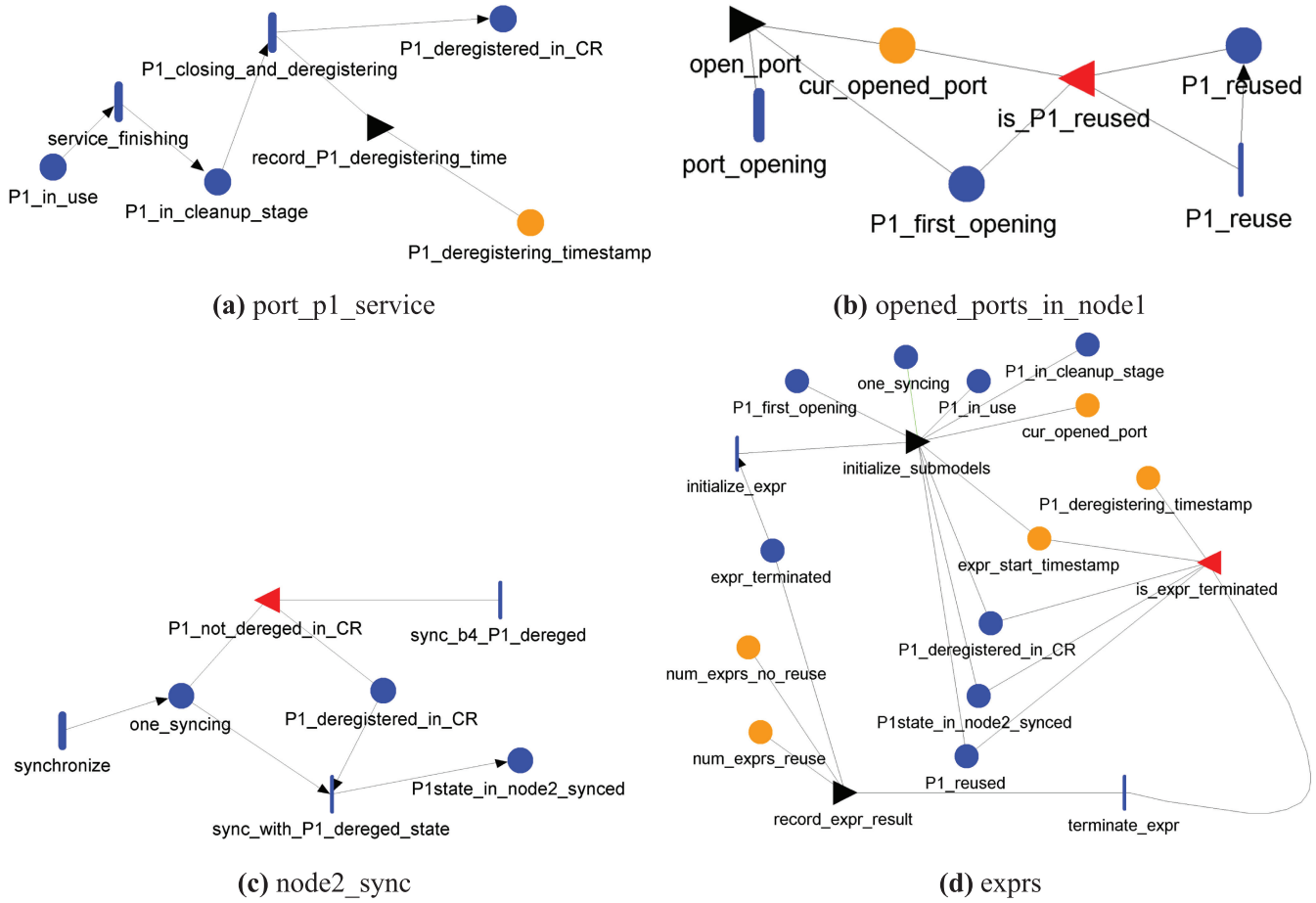


FIGURE 7 | The SAN model for studying the 'query after deregister' scenario.

The **exprs** submodel captures the entire experiment process. The experiment starts with Port 1 being used by Component A. In the *initialize_submodels* output gate, we initialize the token (i.e., value) of each place and record the timestamp at the beginning of the experiment. There are three conditions for the termination of an experiment, and there is no port reuse only when the experiment terminates with condition iii.

i. The deregistering of Node 1's Port 1 is synchronized to Node 2's Registry Cache and the port reuse does not happen.

ii. The deregistering of Node 1's Port 1 is not synchronized to Node 2's Registry Cache and the port is reused.

iii. The time window for purging registry cache T has passed since the deregistering of Node 1's Port 1, but neither the deregistering is synchronized to Node 2 nor the port reuse happens within the time window T. Note that T is a parameter of the cache purging mechanism as detailed in Section 3.2.1 and Figure 5. After T elapses, the incorrect state of Node 1's Port 1 in Node 2 will not be synchronized with the Central Component Registry anymore, and the port will be eventually reused.

After an experiment terminates we record the experimental results, that is, accumulating the number of the experiments with port reuse or the number of those without port reuse, in the *record_expr_result* output gate. Then another experiment starts at the *initialize_expr* activity. After all experiments are simulated, we are able to compute the probability of port reuse scenarios as *num_exprs_reuse/(num_exprs_reuse+num_exprs_no_reuse)*.

4.5 | Study Results

The SANs model is simulated to compute the probability of port reuse scenarios, that is, the probability of 'query after deregistering' scenarios. We set the parameters of the model as shown in Table 1. There is a description of each parameter in the third column.

We set the port number range as 32,768–60,999 according to the range of port numbers assigned by the Linux system [16]. The value of the P1_PORT parameter needs to be between 32,768 and 60,999, and we set P1_PORT as 32,769. According to a field study of network communications [19], we set port_opening_rate as 0.2, deregistering_rate as 0.15 and mean_service_time as 45.8 s. The expr_time parameter represents the total simulation time of all experiments. In our study, we consider the system runs for two days, so we set expr_time as 172,800 s.

The model parameters above are about the target system and workload, but T and sync_period are parameters of the proposed mechanisms and are determined by the user. T's

TABLE 1 | Model parameters.

Parameter	Value/range	Comments
Port number	32,768–60,999	The range of port numbers assigned by the Linux system
P1_PORT	32,769	Port number of P1
port_opening_rate	0.2	Rate of service opens new ports in cloud systems
deregistering_rate	0.15	Rate of service close/deregister ports in cloud systems
mean_service_time	45.8 s	The average lifetime of service in cloud systems
expr_time	172,800 s	Experimental simulation duration
T	40–225 s, with an increment of 10	Past period that the list of deregistered ports maintained by the Central Registry
sync_period	22.9, 45.8, 91.6, 3600 and 86,400 s	Synchronization period in the periodical purging mode

TABLE 2 | Probabilities of the port reuse, that is, 'query after deregister', for various synchronization periods and T intervals.

	sync_period(s)				
T (s)	22.9	45.8	91.6	3600	86,400
40	0.4883	0.4887	0.4881	0.4874	0.5969
60	0.3154	0.3160	0.3153	0.3153	0.4693
80	0.2046	0.2046	0.2039	0.2006	0.3613
100	0.1316	0.1319	0.1319	0.1302	0.2939
120	0.0853	0.0850	0.0853	0.0867	0.2473
140	0.0549	0.0552	0.0550	0.0562	0.2627
160	0.0356	0.0355	0.0354	0.0368	0.2309
180	0.0229	0.0229	0.0232	0.0208	0.1980
200	0.0149	0.0147	0.0146	0.0150	0.2028
220	0.0096	0.0096	0.0097	0.0101	0.1920

value is set in the range of 40–225 s. 40 s is slightly lower than mean_service_time (45.8 s), and 225 s is close to five times of mean_service_time (137.4 s). We also study the impact of synchronization frequency on the probability of port reuse scenarios. The first range of sync_period (22.9, 45.8, 91.6) belongs to frequent synchronization, and they correspond to 0.5, 1 and 2 times of mean_service_time. The second range of sync_period is (3600, 86,400), which represents infrequent synchronization.

4.5.1 | Study Results

Table 2 shows the probabilities of the port reuse, that is, 'query after deregistering' scenarios with various T and sync_periods combinations, as the simulation results of our model in Möbius. It can be seen that during frequent synchronization (sync_period = 22.9, 45.8, 91.6), the port reuse rate is mainly determined by T: the longer T, meaning that the Central Registry will maintain a list of more deregistered endpoints, resulting in the larger probability that Node 2's Registry Cache of Node 1's Port 1 will be synchronized, hence resulting in the smaller probability of the port reuse. The experimental results show that when T reaches 5 times of the mean_service_time (220), the port reuse probability is only 0.0096.

In the case of infrequent synchronization, the increase of the port reuse probability is also very small. When T = 220 s (3.7 min), we only need to set the Registry Cache to synchronize with the Central Registry every 3600 s (1 h), and the port reuse rate is only 0.0101. Only when it is synchronized every 86,400 s (1 day), the port reuse rate is not acceptable. In general, our periodic purging design can effectively avoid the overhead caused by the synchronization message, because it only needs to be synchronized once an hour for a purging time window of 220 s when the Central Component Registry maintains those entries deregistered within the past 220 s.

The simulation results show that the probability of 'query after deregister' occurrences is very small with proper selection of T and sync_period values. Moreover, considering that the service reusing Port 1 in Node 1 may be another service traced by REPTrace, the probability of failures caused by 'query after deregister' is even smaller than our simulation results (the port reuse scenario does not cause a failure in such situations).

5 | Implementation

We implemented the proposed cloud tracing mechanisms on top of the REPTrace software [6–8] in the Linux system. The Registry Cache and Boundary Handling are implemented in the REPAgent of the REPTrace software, which is a dynamic library loaded into each traced process. The REPAgent library leverages the LD_PRELOAD mechanism [20] to intercept LIBC calls of traced processes. We deployed REPTrace and our mechanisms in tracing services hosted on different platforms (OpenStack, Kubernetes and Hadoop), and the experiments performed well. These services were programmed using Python, C, Java or other languages. As Python libraries, Java VMs and the fundamental libraries of many other languages invoke system calls via the LIBC library (so that they avoid the complications of directly invoking system calls using assembly languages), the LD_PRELOAD mechanism allows for the event interception in various platforms and services.

There is no need to coordinate the execution of the LD_PRELOAD-based interceptions in different processes/subsystems. This is because the LD_PRELOAD mechanism of the LIBC library interception is per process, that is, the scope of the LIBC call interception by the LD_PRELOAD mechanism is only within a process. All monitored processes are launched with the REPTrace mechanism. This can be done by simply setting the LD_PRELOAD environment variable value for all those processes launched by shell commands or scripts, and then child processes inherit environment variable values from their parent processes recursively. So, each process has its own copy of the REPAgent (code and data). A REPAgent of a process only captures and handles those events (LIBC calls) within its own process and hence does not interfere with other REPAgents at all. The REPAgents of different processes may access shared memory as registry cache, and they use mutex lock to avoid race conditions in accessing the shared memory.

The Central Component Registry is implemented as a stand-alone user-mode process waiting for connections and requests om REPAgents. It maintains the registration information in memory for fast responses to requests while also keeping an up-to-date copy of the registration in a database to deal with potential crashes of the Central Component Registry. Hashing-based search is implemented in the Central Component Registry to support a large number of entries with good performance. The Path Mending is implemented as part of REPTrace's Central Generator, which is also a stand-alone user-mode process.

In our implementation, we chose the design of the proactive cache purging triggered by the termination of network connections to avoid outdated local cache of the registry. So, we do not have to implement the complex of the synchronization protocol in the periodic cache purge.

We also implemented the shared registry cache to reduce the number of REPAgents' queries to the Central Component Registry. It is implemented based on Linux's shared memory and mutex mechanisms. Linux's shared memory mechanism allocates a block of memory that is shared with many processes on the same node (physical server or virtual machine). Each block of shared memory has a unique ID (or the key term in Linux) that is specified by the user. All Registry Caches have the same specified ID hardcoded in their code. REPAgents call the shmget() function provided by the Linux system to get the access to the shared memory block (if the block is not allocated yet the first invocation of shmget() allocates this block). In order to protect the shared memory from being written by multiple REPAgents simultaneously, we employ the pthread mutex mechanism for synchronization. Typically, the pthread mutex mechanism synchronizes threads of the same process; in our implementation, we created a pthread mutex object residing in another shared memory block and shared the mutex with all REPAgents. Then the REPAgents are able to invoke pthread_mutex_lock() over this mutex object shared across processes for inter-process synchronization in a node.

REPTrace is open-sourced and available at https://github.com/alexvanc/REPTrace.git. We integrated our proposed mechanisms with the REPTrace as enhanced REPTrace, also open-sourced at https://github.com/reasonsyinqin/Extended_REPTrace.git. So, for those systems without REPTrace, it is convenient for cloud service providers to deploy the enhanced REPTrace (i.e., both the REPTrace and the proposed mechanisms in this paper) at once with our deployment scripts. There is no need to re-implement anything or deploy REPTrace first and the proposed mechanisms second (if the REPTrace was deployed in the systems, our deployment scripts can also replace it with the enhanced REPTrace). Moreover, the proposed mechanisms are transparent to traced services, and the partial deployment of REPTrace enabled by the proposed mechanisms makes the maintenance easy: If there are any issues with the tracing of certain components, we can simply disable tracing them and fix the issues offline while the services are normally running.

6 | Experimental Evaluation

We deployed the implementation of our mechanisms into multiple platforms and conducted experiments, including the OpenStack cloud [21], the Kubernetes cloud [13] and the Hadoop platform [14]. We also compared experimental results between our extended REPTrace and the original REPTrace. Finally, we perform experiments to study the performance improvements brought by the event buffering optimization.

Here is a summary of our experimental results before we dive into details of these experiments.

- We could provision VMs and capture the trace events successfully when our two mechanisms were applied, while the VM provisioning service failed when they were not applied. Besides, we could link all of the 32,273 trace events into a single completed REP with a path-mending algorithm, but the events could only be linked into 2962 REP segments without the algorithm. Extended REPTrace has an average of around 2.3% time overheads and 5.9% traffic overheads compared with VM provisioning in the OpenStack cloud environment without cloud tracing.
- The extended REPTrace can generate REP without affecting the normal operation of the robot-shop service in the Kubernetes environment, whereas the original REPTrace will make the request of the service unable to be executed. The extended REPTrace could link the 4522 events into 6 request REPs (each experiment contains 6 requests), whereas the original REPTrace produced 219 segments.
- The overheads caused by the Component Registry and boundary handling mechanisms are small. Compared to the original REPTrace, the overhead increases from 2.58% to 2.95% when tracing the Robot Shop service. For tracing Hadoop, the overhead rises from 5.57% to 8.36%. The breakdown of the Component Registry's performance overhead, based on the average number of operations recorded over 1000 experiments, shows that when tracing the Robot Shop service in Kubernetes, the QUERY, REGISTER and DEREGISTER operations account for 9.04%, 45.48% and 45.48% of the total operations, respectively. In the Hadoop tracing experiments, these percentages are 24.78%, 37.61% and 37.61%.

- To improve the capability of transferring events of the REPAgent to the Central Generator, we implemented an event buffer and studied the impact of its size on the performance of REPTrace on OpenStack, Kubernetes and Hadoop. Compared to directly sending events the reduction in overheads on OpenStack, Kubernetes and Hadoop is approximately 91.45%, when the buffer size is equal to 2048 bytes.

6.1 | Experiments of Tracing VM Provisioning in OpenStack

6.1.1 | Experimental Setup

The OpenStack environment contains a number of services: keystone for authentication, nova for virtual machine management, glance for image management, neutron for network management, cinder for block storage management, swift for object storage management, ironic for management of leasing physical servers to clients, etc. Each of these services also contains multiple components. For example, nova is responsible for managing computing resources and the life cycle of all VMs in the cloud and includes nova-compute, nova-conductor, nova-scheduler and other components.

To evaluate the effectiveness of the proposed mechanisms in supporting the incremental deployment of the cloud tracing capability, we deployed the REPTrace with our Component Registry and boundary handling mechanisms onto part of the OpenStack environment. As the service we targeted is VM provisioning, nova components and rabbitmq, which is intensively used by nova components for communications, are traced in our experiments.

Figure 8 shows the deployment of the cloud tracing capability in the experiments. Nova components and rabbitmq components are in the Traced Scope: nova-scheduler selects which physical server for provisioning a VM; nova-compute manages the life cycle of VMs by invoking the right API of the underlying hypervisor (like KVM and QEMU), including provisioning, start, power-off, power-on and de-provisioning of VMs; nova-conductor accesses the configuration management database of the OpenStack environment on behalf of nova-compute (for security and scalability purposes, nova-compute is not allowed to access the configuration database directly in OpenStack); rabbitmq provides message queue for nova components to communicate with each other. Note that one component may have multiple replica processes, for example, nova-conductor has the number of replica processes the same as the number of logical CPU cores in the system.

The Central Component Registry and the Central Generator are deployed on a separate physical server out of the OpenStack environment. Central Component Registry must start before all REPAgents get loaded; otherwise, REPAgent cannot register themselves to Central Component Registry.

We exploit the LD_PRELOAD mechanism to deploy the REPTrace and proposed mechanisms into the nova components and rabbitmq. These components all run as system service daemons in CentOS 7 Linux, and it is a little challenging to start system service daemons with the LD_PRELOAD enabled. When we launch a program via a shell command, we simply set the shell environment variable LD_PRELOAD properly to enable the mechanism; when we launch a service daemon, we have to edit the service's configuration file (e.g., /usr/lib/systemd/system/openstack-nova-compute.service for nova-compute in CentOS 7) and set LD_PRELOAD as one environment variable of the system service in this file, for enabling the mechanism. Moreover, we need to properly set the permissions of certain directories and files related to the service and disable SELinux (Security-Enhanced Linux) [22] so that the system service daemon can start with REPAgent successfully preloaded and the REPAgent are able to write logs to disk.

The OpenStack environment we set up for experiments is an All-in-One installation on a single server. The server has a CPU Intel Xeon Silver 4214R 2.4G, which has 24 logical CPU cores and 100 GB memory with the CentOS 7 Linux system.

6.1.2 | Experimental Results

We conducted experiments to evaluate the effectiveness and efficiency of our cloud tracing mechanisms.

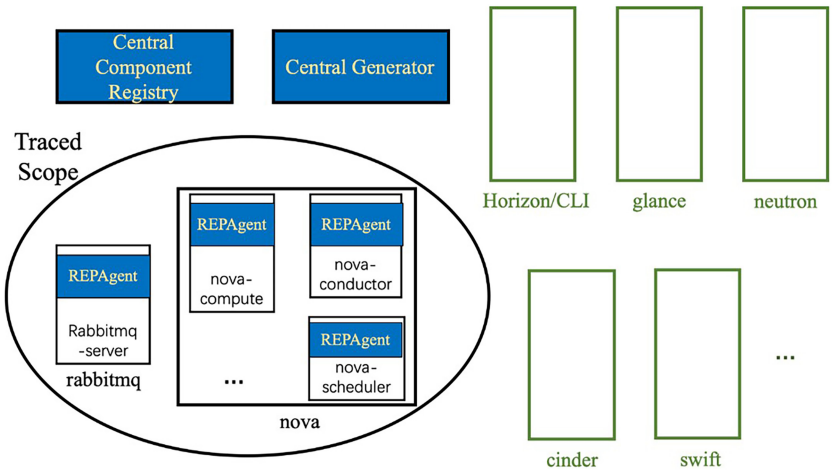


FIGURE 8 | The deployment of the cloud tracing capability onto OpenStack cloud system in our experiments.

6.1.2.1 | Effectiveness of Boundary Handling. We evaluate the effectiveness of the proposed cloud tracing mechanisms by running 10 experiments with REPTrace and the Component Registry and Boundary Handling mechanisms deployed, and during each experiment we issued a command line 'openstack server create <parameters>' to provision a VM. In all the experiments, a VM was successfully provisioned and we captured the trace events as expected. We also ran control experiments of the VM provisioning with REPTrace deployed but our Component Registry and boundary handling mechanisms were not deployed, for comparison purposes. In the control experiments, both nova components and rabbitmq crashed. As the nova service was unable to start, we could not run the 'openstack server create <parameters>' command at all (so no trace events were generated). This result clearly demonstrates the effectiveness of our mechanisms in coping with the boundary of the Traced Scope and reinforces our discussion of REPTrace's incapability for cloud tracing (around Figure 2 in Section 3).

6.1.2.2 | Effectiveness of Path Mending. We collected the trace events generated in the 10 experiments with REPTrace and the Component Registry and boundary handling mechanisms deployed. We captured 32,273 trace events on average in an experiment. After we run REPTrace's event linking and our path mending, the result is one single REP. If we run REPTrace's event linking only and do not run the path mending algorithm, the result is 2962 REP segments, which are broken at the cross-boundary communications. So, the path mending, which is based on the boundary handling mechanism, successfully links the cross-boundary communications into the path, and captures the entire processing of the VM provisioning request within the Traced Scope as a single REP. This result clearly demonstrates the effectiveness of the proposed mechanisms in constructing the complete REP.

6.1.3 | Event Analysis

We analyzed 32,273 trace events. As shown in Figure 9, the vertical axis indicates the time, which begins at 0 and ends after 400 s, and the horizontal axis lists the names of the components (Rabbitmq, Nova-conductor, Nova-scheduler and Nova-compute). We count the events associated with each involved component process within every 5-s interval. The count is actually the number of parent–child linkings between two processes. Then we sum up the counts associated with all processes of a component within the same interval and get the number of communications between the components within the interval. We plot the largest number of communications for all the 5-s intervals as lines in Figure 9 (the arrows point from the parent to the child). The line thickness represents how frequent the communications are. For example, the line between Nova-scheduler and the Rabbitmq during the 5-s interval (300, 305) is very thick, because their communications in this interval are the most frequent in our experiments (2662 events). The following findings can be deduced from Figure 9:

1. Nova Components and Rabbitmq interact with each other most frequently during the provisioning of virtual machines in OpenStack;
2. The provisioning of virtual machines is divided into four stages in OpenStack:
   a. Stage 1: Nova-conductor begins to query the configuration management database.
   b. Stage 2: Nova-scheduler plans the use of resources.
   c. Stage 3: Nova-compute provisions virtual machines, and during the provisioning, Nova-compute asks Nova-conductor to obtain configurations of the VMs via the Rabbitmq.
   d. Stage 4: Nova-scheduler updates and Nova-conductor updates the resource database.

6.1.4 | Overhead Evaluation

To evaluate the performance of the cloud tracing capability (REPTrace + our mechanisms), we conducted experiments to measure the time overhead and traffic overhead.

Table 3 lists the results of the time overhead in our experiments. We ran two batches of experiments with the cloud tracing deployed and 10 experiments were run in each batch. In every experiment of the first batch, we issued a command to OpenStack for provisioning 5 VMs. Then we measured the time from the issuing of the command until all of the 5 VMs

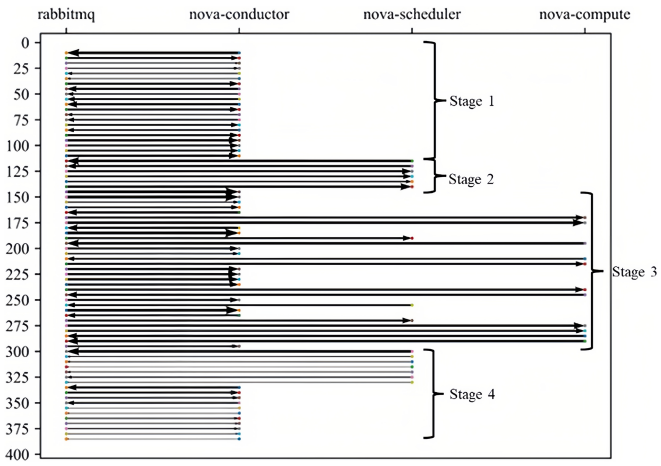


FIGURE 9 | Trace events analysis.

TABLE 3 | Results of time overhead in our experiments.

Times of experiments	Number of VM provisioned	Average time without tracing (ms)	Average time with tracing (ms)	Time overhead
10	5	19,275	19,721	2.3%
10	10	31,783	32,545	2.4%

TABLE 4 | Results of traffic overhead in our experiments.

Average total message size without tracing (bytes)	Average total message size with tracing (bytes)	Traffic overhead
3,655,595	3,870,143	5.9%

TABLE 5 | The components of the Robot Shop service.

Component	Container	Process	Category
nginx-web	robot-shop-master_web_1	nginx:master process nginx-debug	Frontend
payment	robot-shop-master_payment_1	uwsgi–ini payment.ini	Backend
shipping	robot-shop-master_shipping_1	java-jar shipping.jar	
cart	robot-shop-master_cart_1	node server.js	
user	robot-shop-master_user_1	node server.js	
catalogue	robot-shop-master_catalogue_1	node server.js	
ratings	robot-shop-master_ratings_1	apache2-DFOREGROUND	
rabbitmq	robot-shop-master_rabbitmq_1	/bin/sh/opt/rabbitmq/sbin/rabbitmq-server	
dispatch	robot-shop-master_dispatch_1	/bin/sh-c dispatch	
redis	r robot-shop-master_redis_1	redis-server *:6379	Database (not traced)
mongoDB	obot-shop-master_mongodb_1	mongod–bind_ip_all	
mysql	robot-shop-master_mysql_1	mysqld	

were successfully created (i.e., for each of the 5 VMs, the "nova show VM-id" command shows the VM's state as active). The time values in Table 2 are the averages of the 10 experiments. The second batch of experiments is similar to the first batch except that in every experiment, we issued the command to provision 10 VMs. In both batches, we did experiments with and without the cloud tracing deployed and calculated the time overheads incurred by the tracing. The time overhead is around 2.3% ((19,721–19,275)/19,275). The time overhead closely agrees with the range of the time overhead reported in REPTrace, which ranges from 3.1% to 6.0% in their experiments (note that the time overhead is dependent on the specific traced applications or services).

Another type of overhead we measured is the traffic overhead, which is caused by the MSG_ID added to the message. We recorded the original sizes and actual sizes of all transmitted messages that were intercepted by REPAgents in the 10 experiments we conducted for the effectiveness evaluation. These messages include those transmitted within and across the Traced Scope.

Table 4 lists the results of the traffic overhead in our experiments. The traffic overhead is computed as (total of actual sizes − total of original sizes)/total of original sizes. Here, the original sizes of the messages are actually the message sizes if there were no tracing deployed. The traffic overhead is 5.9%.

6.2 | Experiments of Tracing Robot Shop in Kubernetes

We deployed the implementation of our mechanisms in the Kubernetes cloud and traced the REP of the Robot Shop service running in the cloud. Stan's Robot Shop [23, 24] is an e-business application of an online shop created by IBM to show how different services can work together on Kubernetes.

6.2.1 | Experimental Setup

As shown in Table 5, Stan's Robot Shop consists of 12 components that are categorized into three parts: frontend, backend

and database. The frontend is handled by a single process, the Nginx server. The backend includes various components such as Java (Spring Boot) for payment processing, NodeJS for shopping cart functionalities and PHP for ratings. Other services of the backend, such as user management, cataloguing, shipping and dispatch, are developed in Java and Golang. The database part includes Redis, MongoDB and MySQL. In order to demonstrate the capability of our mechanisms in supporting partial deployment of the tracing, we put the frontend and backend parts into the Traced Scope and do not trace the database part. The Kubernetes consists of three virtual machines, each equipped with 16 Intel Xeon CPU at 2.40 GHz cores and 8 GB memory, running Ubuntu 18.04.6 LTS.

Figure 10 illustrates the deployment of the Robot Shop service with the extended REPTrace in a Kubernetes cloud. We initiated resource requests to the Kubernetes system via kubectl. Upon receiving the requests, the kube-api-server stored pod information in etcd (Step 1 in Figure 10). The Controller-Manager, utilizing the watch interface of kube-api-server, detected updates in pod information, integrated the topology structure dependent on the resources and subsequently forwarded the corresponding information to kube-api-server (Step 2). The Scheduler, utilizing the watch interface of kube-api-server, updated pod scheduling information, allocated nodes to pods using algorithms and conveyed pod-node binding information to kube-api-server (Step 3). The kube-api-server dispatched pod information to the kubelet running on the node where the pod was scheduled (Step 4). Upon receiving the pod information, the kubelet organized the data and forwarded it to the Docker engine (Step 5). Docker initialized the containers (Step 6). Central Component Registry and Central Generator were launched on another single node (Step 7). The client sent the request to the Robot Shop (Step 8). The Robot Shop processed the request and sent the response to the client (Steps 9–13). REPAgent conducted activities such as querying, registering, and deregistering with the Central Component Registry and traced the requests during Steps 8–13, and transmitted events to the Central Generator. The Central Generator linked the events with event linking and path mending algorithm.

We ran experiments on all functions of the Robot Shop using a load generation tool from the Robot Shop GitHub repository (/load-gen/robot-shop.py) and employed REPTrace with Component Registry and boundary handling mechanisms. There are 6 types of functions: browsing products, voting for items, adding to a cart, deleting from a cart, calculating the shipping and making payments. 1000 experiments were conducted sequentially. In each experiment, the load generation tool sent 6 requests (one for each type of function) and measured the total execution time of all six requests.

6.2.2 | Experimental Results

6.2.2.1 | Effectiveness of Boundary Handling. The Robot Shop worked normally in all the experiments. We could capture the tracing events as expected. For comparison purpose, we deployed the Robot Shop service with the original REPTrace (without our Component Registry and boundary handling mechanisms). In the control experiments, Robot Shop did not work properly, and all requests failed to get a response as the boundary between the frontend/backend components and database components was not properly handled by the original REPTrace. Moreover, the REPs generated by the original REPTrace were also wrong.

6.2.2.2 | Effectiveness of Path Mending. We collected tracing events from the 1000 experiments using our extended REPTrace. On average, we caught 4522 tracing events in each experiment. After running the event linking and path mending algorithms, the result was 6 REPs, matching the fact that the load generation tool sent 6 requests in each experiment. If we just run REPTrace's event linking algorithm without the path mending algorithm, the result is 219 broken REP segments due to cross-boundary communications.

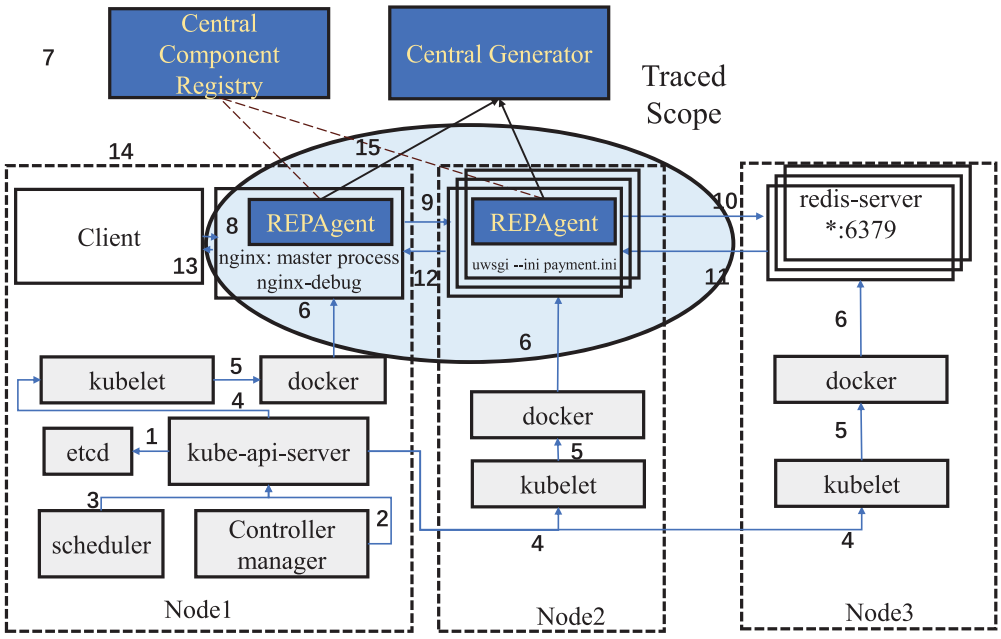


FIGURE 10 | The Robot Shop service with the extended REPTrace deployed in a Kubernetes cloud.

6.2.2.3 | Overhead Evaluation. We also evaluated the overhead of the extended REPTrace tracing the Robot Shop service in the Kubernetes cloud. We sent 6000 requests to the Robot Shop service and recorded the average execution time. Experimental results indicate that the extended REPTrace still has a low overhead of 2.95%: The service's average execution time increases from 715.26 ms (without tracing) to 736.32 ms, as listed in Table 6.

6.2.3 | Event Analysis

Figure 11 shows the request execution path drawn from the REPs generated by the extended REPTrace. The request execution path of the Robot Shop upon receiving a request to purchase a robot is explained here. The *client* sends a request to *nginx-web*, which handles incoming web traffic (path segment 1 in Figure 11). The request from the *nginx-web* is then transmitted to *cart*, which handles shopping cart operations (path segment 2). If the request involves user details, it is directed to *user*. If the request involves product information, it is directed to *catalogue* (segment 3). Once the cart is ready and the user decides to checkout, the request is sent to *payment* to process the payment (segment 4). After the payment is successful, *shipping* takes over to arrange for the shipping of the items purchased (segment 5). Component *dispatch* is informed about the order to manage the logistics and confirm the order is out for delivery (segment 6). After making a purchase, customers are directed to leave feedback through *rating* (the rating system) (segment 7). The message broker *rabbitmq* participates asynchronously at different steps to facilitate communication between components (segment 8). The *cart* sends the response back to *nginx-web* (segment 9). Finally, the *nginx-web* sends the response back to the *client*, concluding the request–response cycle (segment 10).

6.3 | Experiments of Tracing WordCount in Hadoop

We deployed the implementation of our mechanisms in a Hadoop cluster platform, and traced the REP of the WordCount [25] program running in the platform. WordCount is a MapReduce program that counts the number of occurrences of each word in a set of input data.

6.3.1 | Experimental Setup

The Hadoop cluster comprises three VMs, with each node (VM) configured with 8 GB memory and 8 CPU, running Ubuntu

TABLE 6 | Comparison of the time overhead in different environments. For Robot-Shop the results here are the total execution time of all 6 requests in one experiment (the average of 1000 experiments); for Hadoop WordCount, the results are the execution time of a WordCount job in one experiment (also the average of 1000 experiments).

	Kubernetes (Robot-Shop)		Hadoop (WordCount)	
	Average time (ms)	Overhead	Average time (ms)	Overhead
Without tracing original	715.26	0%	64418.34	0%
REPTrace	733.71	2.58%	68007.08	5.57%
Extended REPTrace	736.32	2.95%	69803.71	8.36%

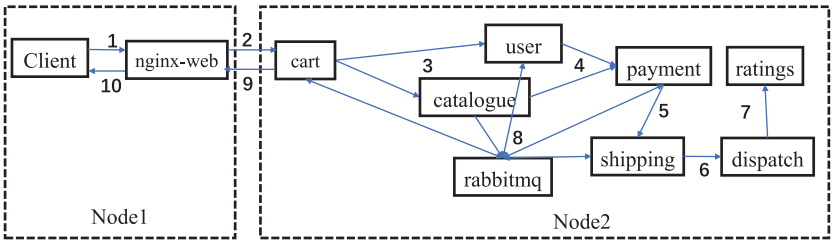


FIGURE 11 | The request execution path of Robot Shop.

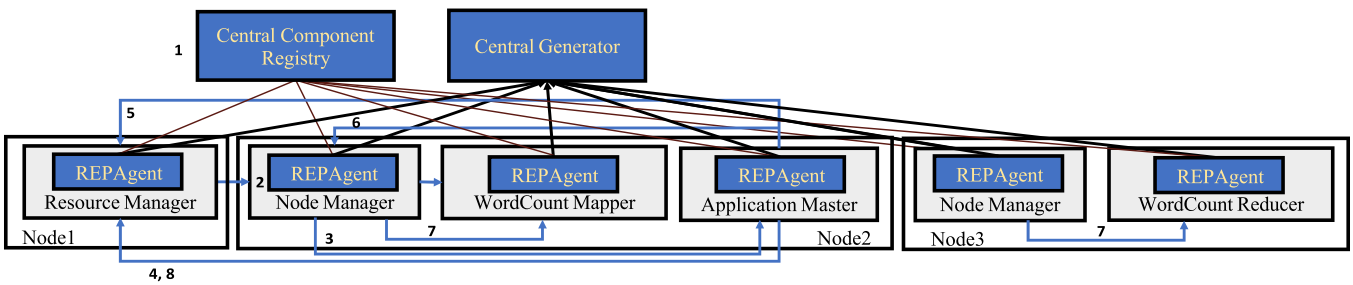


FIGURE 12 | The deployment of the extended REPTrace in a Hadoop cluster platform.

TABLE 7 | Overhead breakdown of the Central Registry mechanism for different types of operations. The results here are the average time (in μs) of each processing segment across all experiments.

Operation processing segment	Kubernetes (Robot-Shop)	Hadoop (WordCount)
QUERY	83.66	80.93
Local Registry Query	0.63	0.73
Query Request Transmission	42.20	44.10
Central Registry Query	0.23	0.70
Query Reply Transmission	40.60	35.40
REGISTER	53.04	45.74
Local Registry Query	0.70	0.32
Local Registry Registration	0.54	0.65
Registration Request Transmission	51.20	44.10
Central Registry Registration	0.60	0.67
DEREGISTER	57.15	68.44
Local Registry Query	0.71	0.32
Local Registry Deregistration	0.54	0.80
Deregistration Request Transmission	55.40	66.90
Central Registry Deregistration	0.50	0.42

18.04.6 LTS. Figure 12 shows the architecture of deploying the extended REPTrace across the three nodes of the Hadoop cluster, where we trace the execution path of the WordCount application running on Hadoop.

The Central Component Registry and Central Generator ran, and the REPAgent was launched with LD_PRELOAD upon starting Hadoop (Step 1 in Figure 12). So, we can trace all components of the Hadoop platform. The WordCount application was submitted through the Hadoop Client, where the ResourceManager allocated a container for the WordCount program and communicated with the corresponding Node Manager (Step 2). The Node Manager initiated the ApplicationMaster for the WordCount application on Node 2 (Step 3). The ApplicationMaster first registered with the ResourceManager and then requested resources for WordCount tasks until completion, repeating Steps 5–7 (Step 4). The ApplicationMaster polled the Resource-Manager via the RPC protocol to request and acquire resources (Step 5). Upon acquiring resources, the ApplicationMaster communicated with the corresponding NodeManager to start the WordCount tasks (Step 6). NodeManager set up the runtime environment for WordCount and started the WordCount tasks (Step 7). After the completion of the WordCount execution, the ApplicationMaster deregistered with the ResourceManager and shut down itself (Step 8). Throughout Steps 2–8, the REPAgents performed activities such as querying, registering and deregistering with the Central Component Registry and traced the requests. The REPAgents transmitted events to the Central Generator, and the Central Generator linked the events with the event linking and path mending algorithms. 1000 experiments were conducted sequentially. In each experiment, a client submitted a WordCount job and measured the execution time of the job.

6.3.2 | Experimental Results

6.3.2.1 | **Effectiveness.** We ran experiments of running WordCount in the Hadoop cluster where the extended REPTrace with Component Registry and boundary handling mechanisms are deployed. The WordCount tasks ran normally in all the experiments, and we also captured the tracing events as expected. Same as in the experiments in the OpenStack and Kubernetes clouds, partial deployment of the original REPTrace in the Hadoop cluster resulted in communication failures due to the incorrect handling of the traced scope boundary, and the REPs generated by the original REPTrace were also wrong.

6.3.2.2 | **Overhead Evaluation.** We also evaluated the overhead of the extended REPTrace tracing the WordCount task in the Hadoop cluster. We executed 1000 WordCount tasks on Hadoop and recorded the average execution time. Experimental results indicate that the extended REPTrace has an overhead of 8.36%. The average execution time increases from 64,418.34 ms (without tracing) to 69,803.71 ms, as listed in Table 6.

6.4 | Overhead Study

6.4.1 | Overhead Comparison With Original REPTrace

We compared the performance overheads of event tracing by the original REPTrace and the extended REPTrace. Particularly, we ran the same experiments of tracing the Robot Shop service in the

Kubernetes cloud and tracing the WordCount tasks in the Hadoop cluster as discussed in Sections 6.2 and 6.3, but with the original REPTrace deployed instead of the extended REPTrace.

The overhead comparison is shown in Table 6. The experimental results indicate that the extended version of REPTrace only adds a small amount of overheads compared to the original REPTrace (from 2.58% to 2.95% when tracing Robot Shop

in the Kubernetes cloud and from 5.57% to 8.36% when tracing WordCount in the Hadoop platform).

6.4.2 | Overhead Breakdown

We did experiments to study the overhead breakdown. As the overhead of the proposed mechanisms almost all associated with the registration and its query, we studied the overhead breakdown for the operations associated with the Central Component Registry mechanism. The overhead of registration queries during the boundary handling is also included in this study. The other parts of the boundary handling and the path mending mechanism have negligible overheads beyond the original REPTrace, as you can see in Algorithms 3 and 4.

We measured the overhead of each segment along the processing of the Central Registry mechanism in the Robot-Shop and Hadoop experiments. For a single QUERY operation, the segments are (i) REPAgent's query at the local registry (0.63 μs in the Robot-Shop experiments and 0.73 μs in the Hadoop ones, by average), (ii) transmission of the operation request to the Central Registry if the endpoint is not found during the local query (42 μs in the Robot-Shop ones and 44 μs in the Hadoop ones), (iii) Central Registry's processing of the operation request (0.23 and 0.70 μs, respectively) and (iv) Central Registry's sending back the results to the REPAgent (41 and 35 μs, respectively). So, the entire QUERY operation (without hit in the local cache) takes 83.66 μs. You can refer to Table 7 for the overhead breakdown for the QUERY and the other two types of operations. Here, the local registry query takes more time than the central registry query because a hashing-based search is used in the Central Registry but not in the REPAgent in the current code.

We also counted the numbers of the three types of operations received by the Central Registry in the experiments. When tracing Robot-Shop, we observed an average of 14.8 QUERY, 74.2 REGISTER and 74.2 DEREGISTER operations in an experiment, which constitute 9.1%, 45.5% and 45.5% of the total operation requests, respectively. When tracing the Hadoop WordCount job, there were 8292.68 QUERY requests (24.8%), 12587.40 REGISTER ones (37.6%) and 12587.40 DEREGISTER ones (37.6%) in an experiment by average. The Central Registry is supposed to receive one REGISTER, one QUERY and one DEREGISTER request for each network connection. But the local registry cache shared by components in the same node made most local queries hit because many connections have the same destination <IP, port>. This largely reduced the number of queries sent to the Central Registry.

Note that if we add up all segments' execution time in an experiment, the sum is far larger than the overhead directly measured an experiment. For example, all of QUERY, REGISTER and DEREGISTER operations in a Robot-Shop experiment take $83.66 \times 14.8 + 53.04 \times 74.2 + 57.15 \times 74.2 = 9414.266\,\mu s$, around 9.4 ms. It is much larger than the 2.61 ms in Table 6 (736.32–733.71). This is because a service has multiple components, and most segments of their QUERY, REGISTER and DEREGISTER operations (e.g., operation request transmissions) run in parallel.

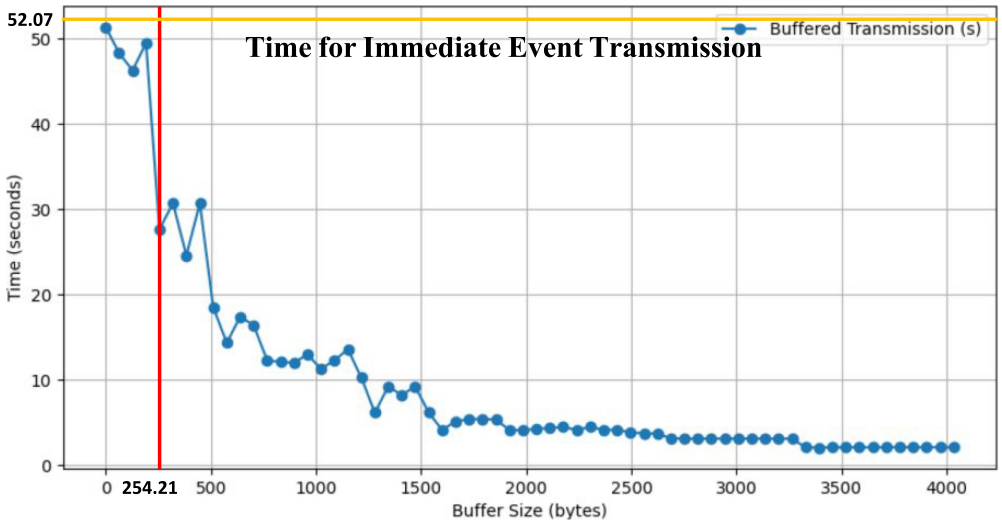
6.5 | Performance Improvement With Event Buffering

Section 3.5 discusses an optimization of event buffering that can minimize the overhead of REPAgent transmitting events to the Central Generator and improve the REPTrace performance. We carried out experiments to demonstrate the performance improvement of this optimization in the OpenStack, Kubernetes and Hadoop platforms. The experimental setups are the same as those described in Sections 6.1, 6.2 and 6.3. In each experiment, REPAgents transmit 1000 events to the Central Generator with each REPAgent using a buffer of a given size.

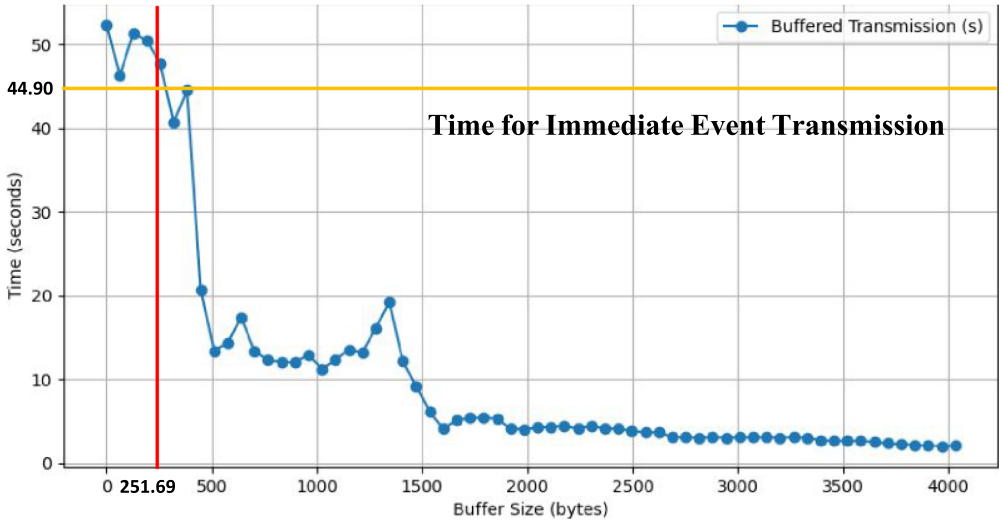
Figure 13 shows the experimental results, and the results evaluate how different buffer sizes affect the time it takes to transmit 1000 events on the OpenStack, Kubernetes and Hadoop platforms. The horizontal axis represents the buffer sizes in bytes ranging from 0 to 4096 at 64-byte intervals. The vertical axis shows the time taken to transmit these 1000 events. The orange line shows the average time it takes for REPAgent to send the 1000 events immediately to the Central Generator without the event buffering (the original design). The blue line shows the time taken with the event buffering (the optimized design). The red vertical line in each figure indicates the average message size of an event, which is the average of all the 1000 events' message sizes. Note that the orange line is a horizontal straight line because the transmission time is not affected by the buffer size at all. From the results, we see that, compared to sending events immediately when they are generated, the transmission time is hugely reduced. When the buffer size is 2048 bytes, the average transmission time of those measured in the three platforms is 4.39s, but the average transmission time when immediate event sending is used is 51.39 s. Then the transmission time is reduced by about 91.45% ((51.39 − 4.393)/51.39) by the event buffering optimization with 2048-byte buffers. So, we choose 2048 bytes as the buffer size in the current extended REPTrace design. We also see that, when the buffer size is equal to or larger than the average size of an event message, the transmission time decreases abruptly. This also clearly demonstrates the large performance improvement provided by the event buffering.

7 | Related Work

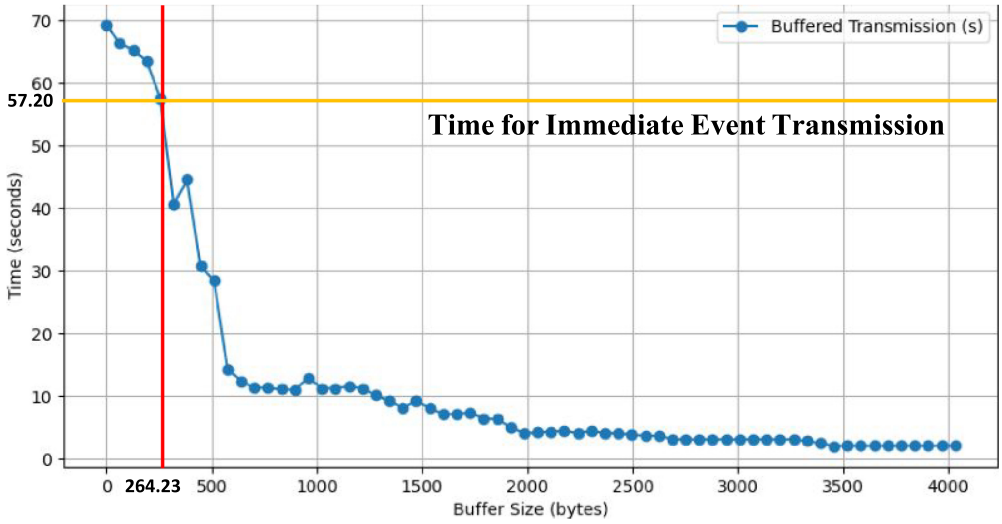
There are summarized technologies of distributed system tracing. X-trace [1], Magpie [2] and Pinpoint [26], they associate trace events with a global identifier, but the knowledge of communication between components is not captured. Dapper [3] is Google's internal tracing system that requires all service components to use the homogeneous control flow/RPC libraries, and it works in the Google scenario. vPath [4] traces distributed systems by monitoring thread and network behaviours with stringent assumptions on the patterns of components' threads and their communications and does not apply in cloud environments that involve complicated computing scenarios. HTrace [9] is a distributed system tracing framework that supports HDFS and HBase; however, manual modification of the distributed system is needed. Stardust [11] can discover request-processing paths but requires source code. The paper [27] designs a data-centric approach to trace the path of processing requests, but it suffers from heavy flow congestion of the centre server dispatcher.



(a) OpenStack



(b) Kubernetes



(c) Hadoop

FIGURE 13 | Time of transmitting 1000 events with the event buffering on different platforms in our experiments (in seconds). The figures show the impacts of the event buffer sizes on the transmission time.

Some approaches to discover request-processing path is to apply statistical technologies to the per-component logs for inferring correlation and causality. They use the methods of statistical analysis [26, 28], code analysis [29], time series analysis [30] or timestamps [31, 32] to generate the end-to-end execution path. Nevertheless, there are difficulties with generating the accurate complete request-processing path as logs may not follow any specific statistical patterns and, in many cases, events may not be exposed by the logs.

Tracing mechanisms like OpenTelemetry [33, 34], Jaeger [35] and Zipkin [36] are currently adopted in cloud systems. A trace_id field and a span_id field are added in HTTP headers in these mechanisms for identifying the request currently being processed. They can only trace HTTP handling functions. Dapper [3], which is also adopted in cloud systems, extended the trace_id and span_id based tracing from HTTP to certain other protocols like RPC by adding these fields into these protocols' headers. In all these tracing mechanisms, only the executions of those functions that have access to the current packet's header are linked together. In-depth processing of the relevant services typically perform application logics and do not directly deal with network packets. These processings cannot be identified and linked by these mechanisms. As a result, these mechanisms only trace the top-level network packet handling but cannot trace into lower-level request processing behaviour of components in the multi-level software stack. REPTrace and the mechanisms proposed in this paper were proposed to achieve the latter objective.

REPTrace [6–8] is a recently proposed technology of tracing in distributed systems, and it supports many comprehensive computing scenarios. REPTrace transparently captures a complete end-to-end tracing path of services and Hadoop jobs. However, the current design of REPTrace is only applicable in a relatively static distributed system, where the complete set of components to be traced is predefined, and the distributed system will have few changes. REPTrace assumes all components involving the processing of target services are traced for generating the complete tracing path. But in the real-world cloud environment, services and components in the cloud keep changing, new services and components are constantly getting onboarded, and old services and components are frequently retiring, it is inevitable to trace the incremental deployment and application on different parts of the cloud platforms. Moreover, the dependencies between components are so complicated that it is impossible to clearly define or determine all cloud components that involve the processing of a single service. Therefore, the direct application of the REPTrace technology is unsuitable for cloud environments.

8 | Conclusion

The capability of obtaining knowledge on cloud service behaviour directly from the execution of the service request/processing is highly desired for cloud service providers. In this paper, we extend the REPTrace technology and propose the mechanisms of component registration, boundary handling and path mending for tracing cloud service request processing (the original REPTrace does not apply in cloud computing environments where there are frequent dynamic changes and also situations of handling communications between traced components and not traced components). The mechanisms are implemented in Linux and are deployed onto the OpenStack, Kubernetes and Hadoop environments. We conducted model-based studies which show that the probability of failures of our Component Registry design is very low (0.0097). Our experimental evaluation shows that the proposed mechanisms effectively trace the request processing of the VM provisioning service in the OpenStack cloud and the Robot Shop service in the Kubernetes cloud and generate a single complete request execution path, while without the mechanisms the cloud tracing could not work at all. With the component registration and boundary handling but without path mending the cloud tracing works, however, the tracing results are an average of 2962 (in the experiments of tracing the OpenStack service) and 219 request execution path segments (in the experiments of tracing the Robot Shop service). Our mechanisms' performance overhead is low.

Data Availability Statement

Our source code is available online: https://github.com/reasonsyinqin/Extended_REPTrace.git.

References

1. R. Fonseca, G. Porter, R. Katz, and I. Stoica, "X-Trace: a Pervasive Network Tracing Framework," in *Proceedings of the 4th USENIX Conference on Networked Systems Design Implementation* (2007), 20.

2. P. Barham, R. Isaacs, R. Mortier, and D. Narayanan, "Magpie: Online Modelling and Performance-Aware Systems," *HotOS*, (2003): 85–90.

3. G. Sigelman, L. Barroso, M. Burrows, et al., "Dapper, a Large Scale Distributed System Tracing Infrastructure," tech. rep., Google Inc., (2010).

4. B. Tak, C. Tang, C. Zhang, S. Govindan, B. Urgaonkar, and R. Chang, "vPath: Precise Discovery of Request Processing Paths From Black-Box Observations of Thread and Network Activities," in *USENIX Annual Technical Conference* (2009).

5. J. Kaldor, J. Mace, M. Bejda, et al., "Canopy: An End-to-End Performance Tracing and Analysis System," in *ACM Symposium on Operating Systems Principles* (2017).

6. Y. Yang, L. Wang, J. Gu, and Y. Li, "Transparently Capturing Request Execution Path for Understanding Service Behavior and Detecting Anomalies," *IEEE Transactions on Services Computing* 16 (2022): 996–1010, https://doi.org/10.1109/TSC.2022.3149949.

7. J. Gu, L. Wang, Y. Yang, and Y. Li, "KEREP: Experience in Extracting Knowledge on Distributed System Behavior Through Request Execution Path," *IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)* 2018 (2018): 30–35.

8. Y. Yang, L. Wang, J. Gu, and Y. Li, "Transparently Capturing Execution Path of Service/Job Request Processing," in *International Conference on Service-Oriented Computing (ICSOC)* (2018), 11236.

9. HTrace, accessed 2024, http://htrace.incubator.apache.org/.

10. M. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, "Pinpoint: Problem Determination in Large, Dynamic Internet Services," in *Proceedings International Conference on Dependable Systems and Networks* (2002), 595–604.

11. R. Thereska, B. Salmon, J. Strunk, et al., "Stardust: Tracking Activity in a Distributed Storage System," *ACM SIGMETRICS Performance Evaluation Review* 34, no. 1 (2006): 3–14.

12. Y. Zhao, C. Liu, T. Yu, et al., "Tracing Processing of Service Requests in Cloud Environments," in *2022 IEEE 27th Pacific Rim International Symposium on Dependable Computing (PRDC)* (2022), 24–33.

13. Kubernetes, accessed 2024, https://kubernetes.io/.

14. Apache Hadoop, accessed 2024, https://hadoop.apache.org/.

15. Tutorialspoint, "The LD PRELOAD Trick on Linux," accessed 2024, https://www.tutorialspoint.com/what-is-the-ld-preload-trick-on-linux.

16. "Ephemeral Port," accessed 2024, https://en.wikipedia.org/wiki/Ephemeral_port.

17. K. Fujimoto and W. H. Sanders, "Performance Evaluation of Distributed Systems Using Stochastic Reward Nets," Perform. Illinois. Edu (2001).

18. "The Möbius Tool, Home," accessed 2024, https://mobius.illinois.edu/.

19. C. Huang and C. Lei, "Bounding Peer-to-Peer Upload Traffic in Client Networks," in *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)* (2007), 759–769.

20. "LD-PRELOAD," accessed 2024, https://www.aldeid.com/wiki/LD-PRELOAD#:~:text=LD-PRELOAD%201%20Description.%20The%20LD_PRELOAD%20enables%20to%20hook,the%20program%20will%20display%20%22Oh%20no%21%22%3A%203%20Comments.

21. OpenStack, accessed 2024, https://www.openstack.org/.

22. "What Is SELinux?," accessed 2024, https://www.redhat.com/en/topics/linux/what-is-selinux.

23. "Stan's Robot Shop," GitHub Repository (2024).

24. Team II, "Sample Microservices Application Running on Kubernetes," IBM Blog (2024).

25. "Hadoop 3.4.0 MapReduce Tutorial," accessed April 26, 2024, https://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html.

26. M. Chow, D. Meisner, J. Flinn, D. Peek, and T. Wenisch, "The Mystery Machine: End-to-End Performance Analysis of Large-Scale Internet Services," *OSDI* (2014): 217–231.

27. N. M. Popa and A. Oprescu, "A Data-Centric Approach to Distributed Tracing," *IEEE International Conference on Cloud Computing Technology and Science (CloudCom)* 2019 (2019): 209–216, https://doi.org/10.1109/CloudCom.2019.00039.

28. A. Anandkumar, C. Bisdikian, and 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* (2008), 133–144.

29. X. Zhao, Y. Zhang, D. Lion, et al., "lprof: A Non-intrusive Request Flow Profiler for Distributed Systems," *OSDI* 14 (2014): 629–644.

30. T. Wang, C. Perng, T. Tao, et al., "A Temporal Data-Mining Approach for Discovering End-to-End Transaction Flow," in *2008 IEEE International Conference on Web Services* (2008), 37–44.

31. K. Kc and X. Gu, "ELT: Efficient Log-Based Troubleshooting System for Cloud Computing Infrastructures," in *2011 IEEE 30th International Symposium on Reliable Distributed Systems* (2011), 11–20.

32. B. Tak, S. Tao, L. Yang, C. Zhu, and Y. Ruan, "LOGAN: Problem Diagnosis in the Cloud Using Log-Based Reference Models," in *2016 IEEE International Conference on Cloud Engineering (IC2E)* (2016), 62–67.

33. "OpenTelemetry: High-Quality, Ubiquitous, and Portable Telemetry to Enable Effective Observability," (2024), https://opentelemetry.io/.

34. A. Thakur and M. Chandak, "A Review on Opentelemetry and HTTP Implementation," *International Journal of Health Sciences* 6 (2022): 15013–15023.

35. "Jaeger: Open Source, Distributed Tracing Platform," (2024), https://www.jaegertracing.io/.

36. Zipkin, (2024), https://zipkin.io/.