Tracing Processing of Service Requests in Cloud Environments

Yinqin Zhao[1], Chang Liu[1], Tao Yu[1], Long Wang[1,2], Xuanqing Shi[5], Yong Yang[3], Ying Li[3], Zhengang Wang[4], Dongdong Shangguan[4]
{zyq21, chang-liu22}@mails.tsinghua.edu.cn, {yu_tao, longwang}@tsinghua.edu.cn, xuanqing.shi@nuaa.edu.cn, {yang.yong, li.ying}@pku.edu.cn, {wangzhengang, shangguandongdong1}@huawei.com
[1]REASONS Lab, Institute for Network Sciences and Cyberspace, Tsinghua University
[2]Zhongguancun Laboratory
[3]National Engineering Center of Software Engineering, Peking University
[4]Huawei Corporation
[5]Nanjing Univeristy of Aeronautics and Astronautics

2022 IEEE 27th Pacific Rim International Symposium on Dependable Computing (PRDC) | 978-1-6654-8555-5/22/$31.00 ©2022 IEEE | DOI: 10.1109/PRDC55274.2022.00017

***Abstract*— Cloud computing is growingly popular for hosting IT services, and is also growing 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 behavior directly from the service execution instead of from people's expertise. This paper studies the problem of tracing cloud services' processing of requests across components in cloud environments, and proposes cloud tracing mechanisms for this purpose. The implementation of the proposed cloud tracing is deployed onto an OpenStack cloud environment, and the experiments performed on the cloud environment shows that our mechanisms effectively trace cloud service behavior and generate a single complete request execution path, while without our mechanisms the cloud tracing either could not work or results in thousands of path segments. Our mechanisms' performance overhead is low (2.3%).**

Keywords—cloud, cloud tracing, distributed tracing, service request, request execution path

I. Introduction

Cloud computing has been popularly adopted in the world for hosting more and more IT services, including those critical for humans' living and life. At the same time the cloud environments are growing into a huge complex; the millions of physical servers, the multi-layer software stacks running on them, and the processing of requests of cloud services spanning many virtual machines and/or containers across the servers and the software layers, all 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 a capability to acquire the knowledge on the cloud service behavior, 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 behavior 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 the responses to the clients.

We looked into existing solutions of distributed tracing in the literature and real-world practice, including X-trace [5], Magpie [6], Google's Dapper [7], vPath [8], Facebook's Canopy [9], REPTrace [2][3][4], Htrace [10], Pinpoint [11], and Stardust [11]. Some of them work only on specific vendor platforms and require their specific libraries or middleware (e.g. X-trace [5], Magpie[6], Google's Dapper [7], Facebook's Canopy [9], and Pinpoint [11]); some require the availability of the traced applications and software's source code (e.g Htrace [10]and Stardust [12]); vPath [7] has stringent assumptions on the patterns of components' threads and their communications, and does not apply in cloud environments which involve complicated computing scenarios. REPTrace [2][3][4] 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.

However, 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. Moreover, the current design assumes all components involving the processing of target services are traced for generating the complete tracing path. The two limitations make the direct application of the REPTrace technology unsuitable for cloud environments. This is because, services and components in clouds keep changing, new services and components keep getting onboarded, and old services and components keep retiring all the time and

2473-3105/22/$31.00 ©2022 IEEE
DOI 10.1109/PRDC55274.2022.00017

frequently, and a predefined set of components is insufficient for tracing cloud behavior; concerning the huge sizes of real-world cloud platforms, complicated dependencies within cloud services, and inter-dependencies among them, incremental deployments and applications of the tracing capability onto different parts of the cloud platforms are inevitable, and it is also not easy (sometimes impossible) to clearly define or determine all cloud components involving the processing of an individual service.

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. This paper designs the following mechanisms, built on top of the REPTrace technology, for addressing the two cloud-specific challenges:

- 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 involving 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.

What we propose in this paper is a general tracing technology that traces the service request processing behavior 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 which are inherent in tracing service request behavior 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 implemented the proposed mechanisms in Linux system and deployed them onto an OpenStack cloud system.
- We conducted experiments of tracing the virtual machine (VM) provisioning service in the OpenStack cloud system. The experiments demonstrated the effectiveness and efficiency of the proposed mechanisms. The experimental results show that, we could trace the request processing of provisioning VMs and generate a single complete request execution path with the proposed registration, boundary handling and path mending mechanisms, while the cloud tracing could not work without the registration or boundary handling, and there are 2962 request execution path segments without the path mending. The performance overhead of our cloud tracing is low, with a 2.3% time overhead and a 5.9% traffic overhead.

The rest of the paper is organized as follows: Section II describes the background of the REPTrace technology. Section III presents the design of our proposed mechanisms. Section IV gives the implementation details. In Section V experimental evaluations of our tracing mechanisms are reported. Section VI summarizes the related work and Section VII concludes the paper.

II. Background: REPTrace

Our cloud tracing mechanisms are built on top of the REPTrace technology [4]. Here we give a brief introduction of this technology.

Request Execution Path (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.

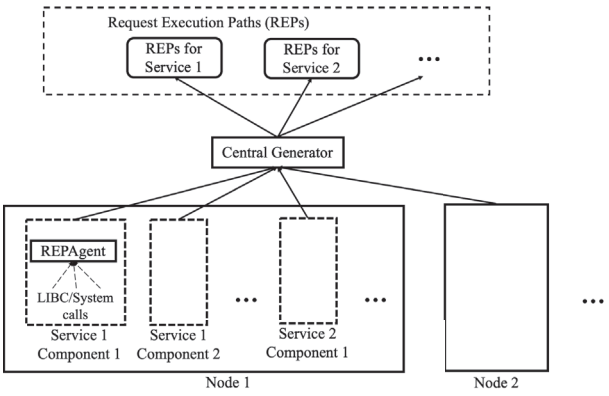

Figure 1: REPTrace Overview

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 which 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 [?] 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.

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 details in [4], 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, TCP header). MSG_CTX_ID is created to identify different parts of a thread's execution that deal with different requests, i.e., it annotates which request the thread is currently processing when an event is generated. Then an event-linking algorithm [4] makes use of the two attributes to link all the events into the REP.

III. Tracing Request Processing of Cloud Services

REPTrace works well if all components involved in processing of the service request are traced. When REPTrace is applied to the cloud environment, there are quite complicated inter-dependencies 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 any more. There are definitely situations where traced components communicate with not traced components, as shown in Figure 2.

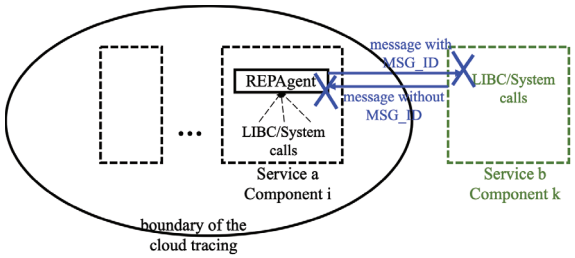

Figure 2: REPTrace is unable to handle communications with not traced components

In this situation the REPAgent in the traced component (*Service a Component i* inside the boundary of the cloud tracing in Figure 2) sends message with the MSG_ID attribute to, or receives 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: 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. Details of the mechanisms are presented below.

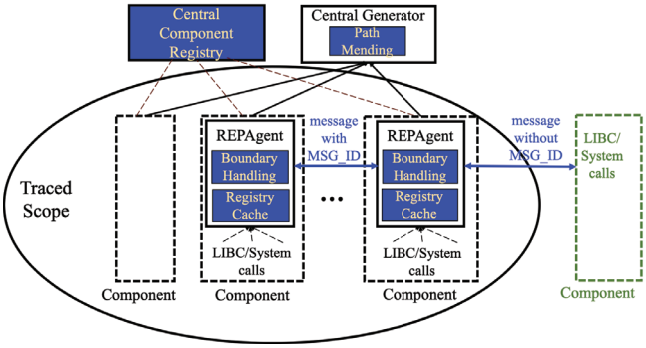

Figure 3: Design of the proposed mechanisms in the REPTrace architecture to support communications between Traced Scope and its outside

A. 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 standalone process, and *Registry Cache* in all the REPAgents (see Figure 3).

Figure 4 describes the algorithm executed by the Central Component Registry and the REPAgent for the component registration task.

(1) Central Component Registry

The Central Component Registry is a standalone 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 Central Component Registry is launched, it creates an empty global list *comm_point_list* (line 3 in Figure 4). 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.

(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 26 in Figure 4). 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.

```
1  Central Component Registry
2      Upon initialization:
3        create comm_point_list, an empty list of <IP Address, Port Number>
4        // or create an initial hardcoded list of known communication points of traced
5        // components
6
7      Upon receiving REGISTER:
8        // <ip_addr, port_num> is the received parameter
9        look up <ip_addr, port_num> in comm_point_list
10       if not found
11         put <ip_addr, port_num> in comm_point_list
12
13     Upon receiving QUERY:
14       // <ip_addr, port_num> is the received parameter
15       look up <ip_addr, port_num> in comm_point_list
16       return the result (found or not found)
17
18     Upon receiving DEREGISTER:
19       // <ip_addr, port_num> is the received parameter
20       look up <ip_addr, port_num> in comm_point_list
21       if found
22         remove it from comm_point_list
23
24 REPAgent
25     Upon initialization:
26       create local_comm_point_list, an empty list of <IP Address, Port Number,
27       socket ID, state (traced or not traced)>
28
29     Upon interception of bind() or connect():
30       get <ip_addr, port_num, socket_id> from the intercepted call's parameters
31       send REGISTER to Central Component Registry with <ip_addr, port_num>
32       look up <ip_addr, port_num> in local_comm_point_list
33       if not found
34         put <ip_addr, port_num, socket_id, traced> in local_comm_point_list
35
36     Upon interception of send(), recv() or equivalent calls:
37       get <ip_addr, port_num> of remote peer from intercepted call's parameters
38       look up <ip_addr, port_num> in local_comm_point_list
39       if found
40         now the state (traced or not traced) is known
41       else
42         send QUERY to Central Component Registry with <ip_addr, port_num>
43         if Central Component Registry returns found
44           put <ip_addr, port_num, -, traced> in local_comm_point_list
45           now the state (traced) is known
46         else
47           put <ip_addr, port_num, -, not traced> in local_comm_point_list
48           now the state (not traced) is known
49
50     Upon interception of close():  // close a socket
51       get <ip_addr, port_num, socket_id> from the intercepted call's parameters
52       look up <ip_addr, port_num, socket_id> in local_comm_point_list
53       if found
54         send DEREGISTER to Central Component Registry with <ip_addr,
55         port_num>
56         remove the found entry from local_comm_point_list
```

Figure 4: The distributed algorithm for the component registry mechanism

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 26). From the algorithm we can 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 REPAgent executes (lines 29-56), let's take a look at a typical TCP communication session illustrated in Figure 5. 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. At 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 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 29-34.

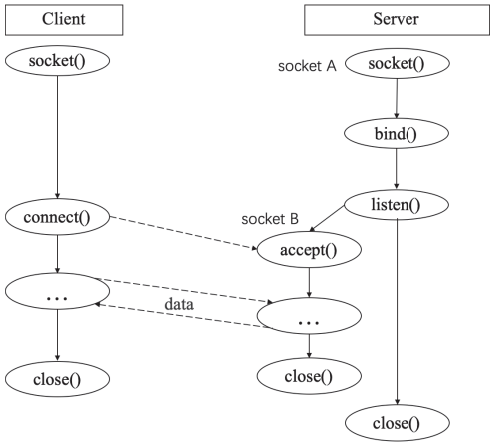

Figure 5: A typical TCP communication session

For supporting the tracing in 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 Figure 4). 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 44 and 47). Please check out lines 36-48 of Figure 4 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, i.e. socket B in Figure 5. 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 of socket B, it is incorrect behavior because the 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 at the server side and the connecting sockets at the client side. Then the lookup step in line 52 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 30-34 and 37-48. For neatness purpose such details are not given in the algorithm in Figure 4.

(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.

Are there "query before register" scenarios? In these scenarios a component is traced and is registering its communication point p at the Central Component Registry, while 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, i.e. the message sending to p or receiving from p occurs concurrently with the registration of p, i.e. the port binding of p or its connection initiation (see lines 29-48 of Figure 4). 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.

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, i.e., 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 t_3 in Figure 6, the list it fetches has the entries deregistered in $[t_3\text{-}T, t_3)$. If the REPAgent's last fetch and purge occurs at t_1, the entries deregistered during $[t_1, t_3\text{-}T)$ will get lost and not synchronized to this REPAgent's cache. So the last fetch and purge should occur later than t_3-T (like t_2), 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.

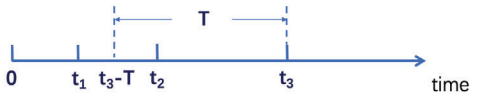

Figure 6: Timeline for REPAgents' periodical purge

Proactive cache purging upon network connection termination. An alternative method to the periodic cache purge above is to leverage the life cycle of network connections and use 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 27). 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 44 and 47). When a connection terminates and the connection's local peer is removed from local_comm_point_list (line 56), 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 the 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 incur 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 situations.

(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, i.e. 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 the access to the shared memory. A mutex lock is used to coordinate these REPAgents for avoiding race conditions.

B. 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.

```
1  boundary_handling() {// called upon interception of send(), recv() or equivalent calls
2          // <remote_ip_addr, remote_port_num> is the remote peer's ip address
3          and port number
4          get <remote_ip_addr, remote_port_num> from the intercepted call's
5          parameters
6          check if  <remote_ip_addr,  remote_port_num> is traced (see the algorithm in
7          the Component Registry section)
8          if <remote_ip_addr, remote_port_num> is not traced
9             get MSG_CTX_ID from current thread context // MSG_CTX_ID is
10            maintained by REPTrace
11            create a REPTrace event and put the MSG_CTX_ID into the event
12            send the event to the Central Generator
13            invoke the original send(), recv() or the equivalent calls to complete the
14            function
15         else // now it is traced
16            do what REPTrace does
17 }
```

Figure 7: Boundary Handling in REPTrace

As shown in Figure 7, 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 Figure 4. If it is not traced, the remote peer is outside the boundary of 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 message with MSG_ID within the boundary and the transmission of message without MSG_ID across the boundary illustrated in Figure 3.

C. 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 Request Execution Path 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 Figure 8.

```
1  path_mending() {
2      for each event e of the collected events
3        if event e is of send(), recv() or other data transmission types
4          if e has no MSG_ID
5            // e was generated for cross-boundary communication
6            find the event p from the whole event set such that p has the same
7            MSG_CTX_ID value as e and p's timestamp is closest before e's
8            set  e's parent as p
9  }
```

Figure 8: The Path-Mending algorithm

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, i.e. we add a link p->e (lines 6-8 in Figure 8). The rationale 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->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->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.

IV. Implementation

We implemented the proposed cloud tracing mechanisms on top of the REPTrace software [2][3][4] in 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 [22] to intercept LIBC calls of traced processes. The Central Component Registry is implemented as a standalone user-mode process waiting for connections and requests from

REPAgents. It maintains the registration information in memory for fast responses to requests while also keeps an up-to-date copy of the registration in a database to deal with potential crashes of the Central Component Registry.

The Path Mending is implemented as part of REPTrace's Central Generator, which is also a standalone user-mode process.

In our implementation we chose the design of the proactive cache purging triggered by the termination of network connections, for avoiding 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.

V. Experimental Evaluation

We deployed our implementation onto an OpenStack cloud environment and traced the OpenStack's processing of VM provisioning requests in our experimental evaluation. We choose OpenStack for our experiments as it is a popular cloud platform [19].

Here is a summary of the results of our experimental evaluation, before we dive into details of the experiments.

- We could provision virtual machines and capture the trace events successfully when our Component Registry and Boundary Handling mechanisms were applied, while the VM provisioning service failed when they were not applied.
- We could link all of the 32,273 trace events into a single complete request execution path (REP) with the path-mending algorithm, while the events could only be linked into 2962 REP segments without the algorithm.
- Our mechanism have averagely around 2.3% time overhead and 5.9% traffic overhead compared with VM provisioning in the original OpenStack cloud environment without cloud tracing.

A. 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 incremental deployment of the cloud tracing capability, we deployed the REPTrace and 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 9 shows the deployment of the cloud tracing capability in the experiments. Nova components and rabbitmq component 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 deprovisioning 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, e.g. nova-conductor has the number of replica processes the same as the number of logical CPU cores in the system.

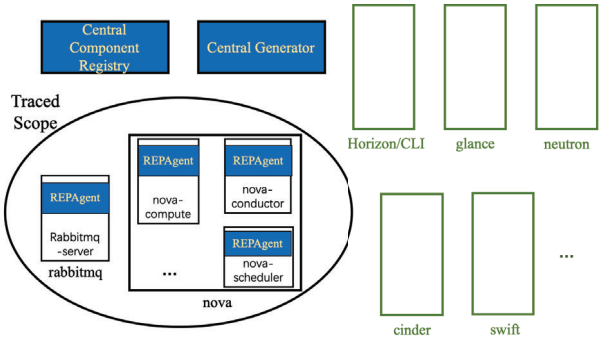

Figure 9: The deployment of the cloud tracing capability onto OpenStack cloud system in our experiments

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) [23] 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 of Intel Xeon Silver 4214R 2.4G, which has 24 logical CPU cores, and 100GB memory with the CentOS 7 Linux system.

B. Experimental Results

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

(1) Effectiveness Evaluation

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 not deployed, for a comparison purpose. 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 III).

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 by 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 are 2,962 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 request execution path.

(2) Event Analysis

We analyzed 32,273 trace events. As shown in Figure 10, the vertical axis indicates the time, which begins at 0 and ends after 400 seconds, and the horizontal axis lists the names of the components (Rabbitmq, Nova-conductor, Nova-scheduler, and Nova-compute). We count the events associate with each involved component process within every five-second 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 five-second intervals as lines in Figure 10 (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 five-second 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 the Figure 10:

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: Stage 1: Nova-conductor begins to query the configuration management database; Stage 2: Nova-scheduler plans the use of resources; 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; Stage 4: Nova-scheduler updates and Nova-conductor updates the resource database.

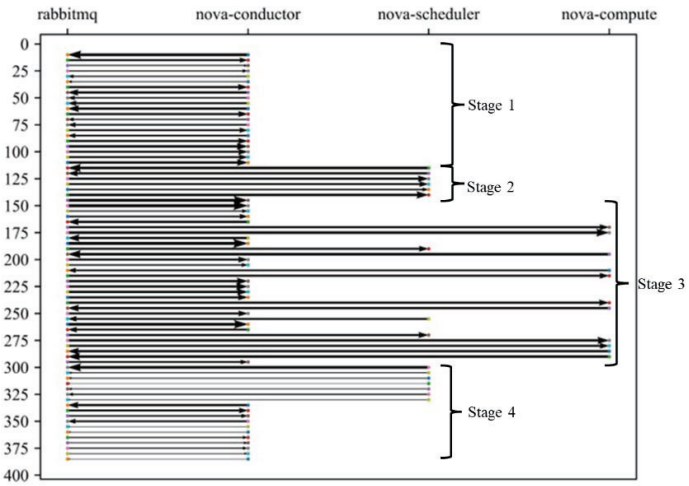

Figure 10:Trace events analysis

(3) 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 1 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 are 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 1 are the averages of the

10 experiments. The second batch of experiments are 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% ((19721-19275)/19275). The time overhead closely agrees with the range of the time overhead reported in REPTrace [4], which ranges from 3.1%~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 which 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 2 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%.

TABLE 1: Results of Time Overhead in our Experiments

Times of Experiments	Number of VM provisioned in each experiment	Average time without tracing (ms)	Average time with tracing (ms)	Time Overhead
10	5	19275	19721	2.3%
10	10	31783	32545	2.4%

TABLE 2: 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%

VI. RELATED WORK

There are summarized technologies of distributed system tracing. X-trace [5], Magpie [6] and Pinpoint [11], they associate trace events with a global identifier, but the knowledge of commutication between components are not captured. Dapper [7] is Google's internal tracing system that requires all service components use the homogeneous control flow/RPC libraries, and it work in the Google scenario. vPath [8] traces distributed systems by monitoring thread and network behaviors with stringent assumptions on the patterns of components' threads and their communications, and does not apply in cloud environments which involve complicated computing scenarios. HTrace [10] is a distributed system tracing framework, which supports HDFS and HBase, however, the manual modification of the distributed system is needed. Stardust [12] can discover request-processing paths but requires source code. [20] traces activities of distributed systems by extending service mesh, it is limited in that it can not tracing the communications between traced components and not traced ones. [21] designs a data-centric approach to trace the path of processing requests, but it suffers from heavy flow congestion of the center server dispatcher. 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[13][14], code analysis[15], time series analysis[16], or timestamps[17][18] 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 expose by the logs.

REPTrace [2][3][4] is a recently proposed technology of tracing in distributed system, 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 be little 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, old services and components are frequently retiring, it is inevitable to trace the incremental deployment and application on different parts of 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.

VII. CONCLUSION

The capability of obtaining the knowledge on cloud service behavior directly from the execution of the services' request/ processing is highly desired for cloud service providers. But state-of-the-art tracing technologies cannot apply in the cloud computing environment where there are much frequent dynamic changes and also situations of handling communications between traced components and not traced components.

In this paper we propose the mechanisms of component registration, boundary handling and path mending for tracing cloud service' request processing. The mechanisms are implemented in Linux and are deployed onto an OpenStack cloud environment. Our experimental evaluation shows that the proposed mechanisms effectively trace the request processing of the OpenStack cloud's VM provisioning service 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 request execution path segments instead of a single path in our experiments. Our mechanisms' performance overhead low: the time overhead is around 2.3% and the traffic overhead is 5.9% in the experimental evaluation.

ACKNOWLEDGEMENT

This work has been supported by the NSFC Project of China under Grant 62132009. Long Wang is the corresponding author.

REFERENCES

[1] https://en.wikipedia.org/wiki/Cloud_computing

[2] Y. Yang, L. Wang, J. Gu, Y. Li. “Transparently Capturing Execution Path of Service/Job Request Processing,” International Conference on Service-Oriented Computing (ICSOC) 2018. Lecture Notes in Computer Science, vol 11236.

[3] J. Gu, L. Wang, Y. Yang, Y. Li, “KEREP: Experience in Extracting Knowledge on Distributed System Behavior through Request Execution Path,” 2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW). IEEE, 2018: 30-35.

[4] Y. Yang, L. Wang, J. Gu and Y. Li, "Transparently Capturing Request Execution Path for Understanding Service Behavior and Detecting Anomalies," in IEEE Transactions on Services Computing, doi: 10.1109/TSC.2022.3149949.

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

[6] P. Barham, R. Isaacs, R. Mortier, D. Narayanan, “Magpie: Online Modelling and Performance-aware Systems,” HotOS.2003: 85-90.

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

[8] B.C. Tak, C. Tang, C. Zhang, S. Govindan, B. Urgaonkar, R.N. Chang, “vPath: Precise Discovery of Request Processing Paths from Black-box Observations of Thread and Network Activities,” USENIX Annual technical conference. 2009

[9] J. Kaldor, J. Mace, M. Bejda, E. Gao, W. Kuropatwa, J. O’Neal, K.W. Ong, B. Schaller, P. Shan, B. Viscomi, V. Venkataraman, K. Veeraraghavan, Y.J. Song, “Canopy: an End-to-end Performance Tracing and Analysis System,” ACM Symposium on Operating Systems Principles, 2017

[10] HTrace, http://htrace.incubator.apache.org/

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

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

[13] M. Chow, D. Meisner, J. Flinn, D. Peek, T.F. Wenisch, “The Mystery Machine: End-to-end Performance Analysis of Large-scale Internet Services,” OSDI. 2014: 217-231.

[14] A. Anandkumar, C. Bisdikian, D. Agrawal, “Tracking in a Spaghetti Bowl: Monitoring Transactions Using Footprints,” In SIGMETRICS ’08: Proceedings of the 2008 ACM SIGMETRICS international conference on Measurement and modeling of computer systems, pages 133-144, New York, NY, USA, 2008. ACM.

[15] X, Zhao, Y. Zhang, D, Lion, M.F. Ullah, Y. Luo, D. Yuan, M. Stumm, “lprof: a Non-intrusive Request Flow Profiler for Distributed Systems,” OSDI. 2014, 14: 629-644.

[16] T. Wang, C. Perng, T. Tao, C. Tang, E. So, C. Zhang, R. Chang, L. Liu, “A Temporal Data-mining Approach for Discovering Endto-end Transaction Flow,” Web Services, 2008. ICWS'08. IEEE International Conference on. IEEE, 2008: 37-44.

[17] K. Kc, X. Gu, “ELT: Efficient Log-based Troubleshooting System for Cloud Computing Infrastructures,” Reliable Distributed Sys-tems (SRDS), 2011 30th IEEE Symposium on. IEEE, 2011: 11- 20.

[18] B.C. Tak, S. Tao, L. Yang, C. Zhu, Y. Ruan, “LOGAN: Problem Diagnosis in the Cloud Using Log-based Reference Models,” Cloud Engineering (IC2E), 2016 IEEE International Conference on. IEEE, 2016: 62-67.

[19] https://www.OpenStack.org/

[20] D. Cha and Y. Kim, "Service Mesh Based Distributed Tracing System," 2021 International Conference on Information and Communication Technology Convergence (ICTC), 2021, pp. 1464-1466, doi: 10.1109/ICTC52510.2021.9620968.

[21] N. M. Popa and A. Oprescu, "A Data-Centric Approach to Distributed Tracing," 2019 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), 2019, pp. 209-216, doi: 10.1109/CloudCom.2019.00039.

[22] 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

[23] https://www.redhat.com/en/topics/linux/what-is-selinux