**Regular Paper**

# Low Overhead TCP/UDP Socket-based Tracing for Discovering Network Services Dependencies

Yuuki Tsubouchi[1,2,a),b)] Masahiro Furukawa[3,c)] Ryosuke Matsumoto[1,d)]

Received: May 31, 2021, Accepted: December 3, 2021

***Abstract:*** The widespread use of cloud computing has made it easier for service providers to develop new features and handle increased access. However, the network dependencies among components in distributed applications deployed in the cloud are becoming more complex because the number and types of components are increasing. When system administrators make changes to a system, they cannot specify the impact of the changes, which may lead to larger failures than expected. Current methods of automatically discovering dependencies trace network flows included in TCP/UDP sockets in the Linux kernel on all hosts deployed in distributed applications. However, as the rate of communication increases, the number of flows transferred from the kernel space to user space increases, which increases CPU usage for tracing. We propose a low-overhead method of bundling multiple flows with the same network service into a single flow in a kernel to discover dependencies. The proposed method reduces the number of transferred flows to the user space, thus reducing CPU usage. Experimental results from evaluating our method indicate that the method maintains a CPU overhead below 2.2% when the number of flows increases.

***Keywords:*** tcp/udp, ebpf, kernel, monitoring, site reliability engineering

## 1. Introduction

As cloud computing becomes more widespread, distributed applications deployed in the cloud are required to increase the number of accesses, develop new features, and operate for a long period. Distributed applications are composed of many different OS processes (henceforth, network services) that communicate over a network. Service providers use the scale-out method to distribute the access by increasing the number of replicas of each network service. When adding new features, different types of network services called middleware may be added to the system to execute load balancing, caching, full-text text search, and other features. During long-term operation, migrating the platform where the applications are deployed may be necessary. In such cases, old and new systems can be temporarily mixed within the applications. These factors complicate the network dependencies among network services.

System administrators must continue to modify a system to improve security and add functionality while maintaining a high level of reliability for complicated distributed applications [2]. An anomaly in a particular network service, such as a failure or performance degradation, can affect other network services through the network. If system administrators make a problematic modification, the negative impact propagate to all the network services that depend on the modification. If system administrators modify the system without knowing the dependencies among the network services, the modification can cause a larger-than-expected failure. To avoid widespread failures, system administrators need to know the dependencies and identify in advance the areas that will be affected by the modification. However, since dependencies change dynamically, it is difficult for system administrators to remember all dependencies. Therefore, automatically discovering dependencies is desirable.

There are currently two approaches for discovering dependencies: one that modifies the application or middleware code and one that does not. Application-modified methods involve intercepting network communication [3], [5], [6], [7], [8], [19], [23], [25], [30], [31]. Application-unmodified methods involve adding discovery processing to the application code before and after network communication [12], [26], [27]. To avoid extra tasks, discovering dependencies without adding applications or middleware code is desirable. To avoid application-performance degradation, discovering dependencies with low overhead is necessary.

Application-unmodified methods [5], [8], [19], [25], [30] trace the source and destination tuples (henceforth, flows) from network sockets in the Linux kernel. These methods do not require modifying the user-space code because of the independence on higher-level application layer protocols. Neves et al.'s

[1] SAKURA Internet Research Center, SAKURA Internet Inc., Osaka 530–0011, Japan

[2] Graduate School of Infomatics, Kyoto University, Kyoto 606–8501, Japan

[3] Hatena Co., Ltd., Kyoto 604–0835, Japan

[a)] y-tsubouchi@sakura.ad.jp

[b)] y-tsubouchi@net.ist.i.kyoto-u.ac.jp

[c)] masayoshi@hatena.ne.jp

[d)] r-matsumoto@sakura.ad.jp

A preliminary version of this paper has already been published at the 1st IEEE International COMPSAC Workshop on Advanced IoT Computing (AIOT 2020) in July 2020 [28]. In contrast to the preliminary version, this paper expands the novelty of the proposed method and enhances the experiments comparing the proposed method with other existing methods.

method [25] intercepts the TCP/UDP processing path in the Linux kernel, thus aggregates identical flows in kernel within a certain period. The aggregation reduces CPU usage for transferring the flows from the kernel space to user space when the number of TCP/UDP messages increases. However, the number of connections per time unit increases in a network service having a large number of short-lived TCP connections compared with one in which persistent connections are used. Increasing the number of connections increases the number of flows, which in turn increases the transferring cost.

We propose a new application-unmodified method that maintains low overhead even when the number of flows increases to automatically discover dependencies via TCP/UDP. The proposed method bundles multiple TCP/UDP flows, when the destination of the flows are the same network service, into a single flow in the Linux kernel. This can reduce the number of flows transferred to the user space by bundling flows in the kernel space. We implemented our in-kernel flow-bundling method with the extended Berkeley Packet Filter (eBPF) [4], [13], which allows arbitrary programs to be executed within the constraints of a sandbox in the Linux kernel. We then conducted experiments to evaluate the effectiveness of our method. The experimental results indicate that our method maintains CPU overhead below 2.2% as the number of flows increases and that the overhead of application latency is sufficiently small.

The rest of this paper is organized as follows. In Section 2, we give background and discuss current methods and our motivation. In Section 3, we present the design of our proposed method and its implementation with eBPF. We discuss the experiments we conducted to evaluate the effectiveness of the proposed method in Section 4 then present the results along with limitations of the proposed method in Section 5. We conclude the paper in Section 6.

## 2. Background and Motivation

### 2.1 Distributed Applications in Cloud

Distributed applications have complex network dependencies because of the development of new features, load balancing, and network communication among different applications.

The three-tiered architecture, which is composed of a web server, an application server, and a database server [20] is used in distributed applications deployed in the cloud. Distributed applications often include middleware servers suitable for the feature requirements of web services. For example, application developers use a NoSQL [14] such as a key-value store (KVS) and a full-text search engine, as the additional middleware. This is in accordance with the requirements of user access patterns and the required data structure which cannot be satisfied using traditional relational databases. To access the application, application users query an external DNS server and obtain the IP address of the web server. When using domain names for connecting internal endpoints in private networks, each service queries the internal DNS server and obtains the destination address.

The load can be balanced with multiple network services with the same configuration in accordance with the increase in the number of accesses because of the limitation of hardware capacity [15].

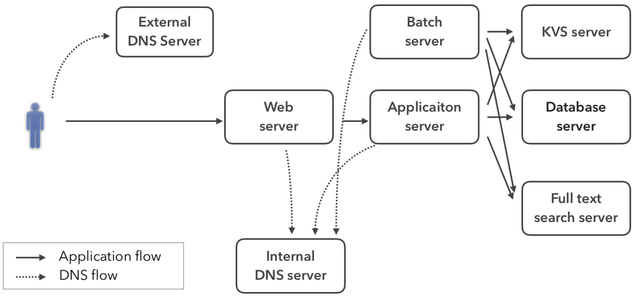


**Fig. 1** Example of system configuration of distributed applications.

When features of an application are reused by other applications, these other applications communicate with the application over the network. The microservice architecture [11] divides a single and monolithic application into many small applications, and loosely couples them to reuse each service.

### 2.2 Definition of Dependencies

We use the definition of dependencies reported by Zand et al. [31]. A service $S_2$ depends on another service $S_1$ if the latency, degradation, or failure in $S_1$ directly or indirectly causes latency, degradation, or failure in $S_2$.

This definition is attributed to the implementation of TCP/UDP in the UNIX OS as follows. Network services using TCP are divided into two network-service sides: connection-requesting side and listening side. The connection-requesting side is called a client, and the listening side is called the server. The server opens a fixed listening port and waits for connection requests. If no port is specified, the client opens a short-lived port randomly assigned by the OS and connects to the server's listening port. Since UDP is a connectionless protocol, it has no connection-requesting side. In UDP, we refer to the side that listens on a fixed port as the server and the side that does not listen as the client. The client will not operate if the server stops, but the server will maintain the listening state even if the client stops. Within the scope of TCP/UDP, $S_1$ is a server, and $S_2$ is a client.

The application layer protocols used in the cloud can also distinguish between the requesting and responding sides, such as HTTP, DNS, and database query and response. The requesting side is dependent on the responding side in application layer protocols as well as in TCP/UDP. There is a practice of matching the direction of dependency between these two layers in many network services deployed in the cloud. In **Fig. 1**, for example, the web and application servers are both the TCP connection listeners and the HTTP request listeners, and the database servers are the TCP connection listeners and listen for queries using their own custom protocols.

### 2.3 Short-lived and Persistent TCP Connections

Distributed applications use a short-lived or persistent TCP connection. A short-lived connection establishes a TCP connection for each request. A persistent connection reuses an established TCP connection to reduce the overhead of establishing and terminating the connection. To discover dependencies of various applications, we should trace both short-lived and persistent connections with a low overhead.

Although persistent connections are advantageous in performance, Moon et al. [22] argue that not all applications benefit from persistent connections. To support persistent connections, applications need to manage the states of the connections. For example, applications need to periodically check for connection timeouts and terminate idle connections. If a network device relaying connections tracks the connection states, the tracking table can become full as the lifecycle of the connections becomes longer. This makes the processing and configuration more complicated and error-prone. In PHP applications, persistent connections are often discouraged to prevent resource abuse [1].

### 2.4 Weights of Edges in the Dependency Graph

Weighting the edges of a dependency graph with network services as vertices and flows as edges enables system administrators to determine which vertex or edge has a problem. Examples of edge weights are throughput per flow, round trip time (RTT), number of messages, number of errors in sending and receiving messages, and number of connections.

Because some of these metrics cannot be measured at the time of TCP connection establishment, it is necessary to obtain the metrics for each TCP message. For example, the throughput can be calculated from the total length of messages transferred per unit of time. Thus, a message length must be obtained each time the message is sent or received.

### 2.5 Related Works

Packet-based, socket-based, and transaction-based methods have been proposed for discovering dependencies.

The packet-based methods collect packets from the existing traffic and discover dependencies on the basis of the destination and source information contained in the packets [3], [6], [7], [23], [31]. Since we collect packets from network devices or an OS kernel on the host, system administrators do not require modifying existing applications. If the packets are observed at network switches, deploying additional processes on the hosts is unnecessary because system administrators modify only the switches. However, if all the packets are observed, the CPU usage for packet collection and analysis becomes large when the packets rate is large.

Transaction-based methods trace transactions on application layer protocols such as HTTP requests and responses, and database queries and responses [12], [26], [27]. These methods can discover the path of each transaction and measure application-level statistics, such as response time and the number of errors. However, they require application developers to modify the code of the application and off-the-shelf middleware. The additional logging increases the overhead of collecting logs.

Socket-based methods trace flows included in network sockets in an OS kernel [5], [8], [19], [25], [30]. Current visualizing systems of a dependency graph use socket-based methods for discovering dependencies. CauseInfer [5] and Microscope [19] require the dependency graph as prior knowledge to be input into a statistical model to diagnose the cause of performance issues. Weave Scope [30], Datadog Network Performance Monitoring [8], and Neves et al.'s method [25] enable system administrators to visually understand network usage by application by visualizing the dependency graph. We classify socket-based methods into the following three categories: snapshot polling, streaming, and in-kernel flow aggregation.

**Snapshot polling method** CauseInfer periodically takes snapshots of the list of flows from the Linux proc file system. We call this method as snapshot polling method. This method does not require modifying the kernel code and obtains only the data provided by the kernel in the user space. However, this method can miss flows for TCP connections that are shorter-lived than the snapshot and acquisition intervals. It also has an OS implementation limitation, i.e., UDP flows cannot be obtained from the proc file system.

**Streaming method** Microscope [19] and Weave Scope [30] intercept system calls or kernel functions regarding TCP/UDP socket operations in the Linux kernel with Linux Kernel Probes (Kprobes) [21] and tracepoints [9]. We call these methods as streaming methods. After obtaining the flows from the arguments and returning values of the intercepted system calls and kernel functions, the flows are immediately transferred to the user space. Since this method intercepts the TCP/UDP path in the kernel, there is no loss of flow. However, as the number of TCP connections per unit of time and number of messages for UDP transmission and reception increases, the number of flows to be transferred increases. Thus, the CPU usage with the methods increases.

**In-kernel flow-aggregation method** Datadog Network Performance Monitoring [8], and Neves et al.'s method [25] use the Linux eBPF to aggregate events with the same flow in the kernel within a certain period. We call these methods as in-kernel flow-aggregation methods. The aggregation reduces CPU usage for transferring the flows to the user space, as the nummber of TCP/UDP messages increases. However, the number of connections increases in a network service having a large number of short-lived TCP connections compared with a service in which persistent connections are used. The issue with these methods is that the transfer cost is proportional to the increase in the number of flows.

### 2.6 Motivation

A challenge common to each method is increased CPU usage or application processing overhead. A single TCP connection consists of multiple round trips of packets, and single or multiple transactions are issued over a single TCP connection. Therefore, socket-based methods are advantageous in terms of CPU usage.

Random sampling is a common solution to reduce tracing overhead, but sampling can result in missing dependencies. In batch processing, in particular, some clients communicate with the server only once a day or week. The probability of missing dependencies of network services that communicate rarely is higher than that of network services that communicate constantly. System administrators can exclude hosts with a large overhead from the tracing targets as a first-aid measure in actual operation. However, remembering the excluded hosts increases their cognitive load.

We need a socket-based low-overhead tracing method to obtain complete dependencies. To address the issue of the in-kernel flow-aggregation method on TCP short-lived connections, reducing the number of flows transferred to the user space is effective. To improve the in-kernel flow-aggregation method, the key to our proposed method is to bundle multiple flows into a single flow in the kernel. However, bundling all flows with a network service communicating with many other network services into a single flow can result in missing dependencies. Thus, our method should identify multiple destinations.

## 3. Proposed In-kernel Flow-bundling Method

Clients can connect with a server via many different short-lived ports with the same IP address to process network I/O concurrently in distributed applications. In a web application, for example, the web server can have multiple worker processes to process multiple HTTP requests concurrently, and each worker process can connect to the same database server. Considering multiple flows having different short-lived ports of the clients to the same listening port as a single flow does not result in missing dependencies.

### 3.1 Design of the Proposed Method

**Figure 2** shows an overview of our proposed method in a host. Network services in a user space on the host generally execute TCP/UDP communication through sockets in the kernel. The tracing process on the host tranfers a program for tracing flows to the kernel at boot time of the process. The procedure of the kernel program is as follows.

**Step 1** The kernel program intercepts the kernel-function calls corresponding to TCP connection establishment and TCP/UDP messages transmission/reception.

**Step 2** The kernel program takes the socket structure as input and executes the in-kernel flow-bundling shown in Algorithm 1. A hash table is created to bundle and store the flows at boot time of the tracing process. Each time a kernel function to be intercepted is called, the kernel program stores the intercepted flows in the hash table. The following tuple is a key in the hash table: <source address (saddr), destination address (daddr), listening port (lport), flow direction, IP protocol>.

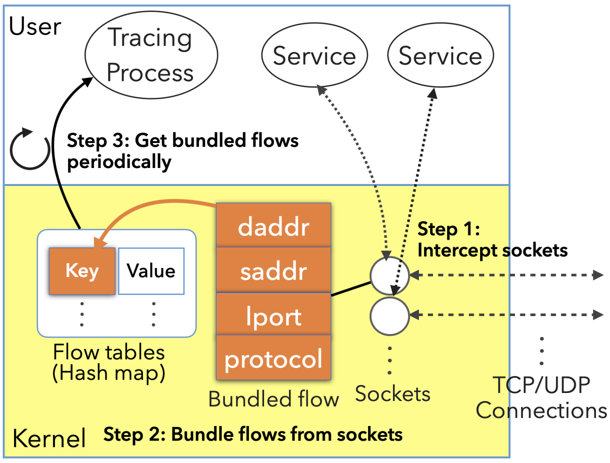


**Fig. 2** Overview of tracing with in-kernel flow-bundling.

Algorithm 1 measures the network throughput as the weight of the dependency graph. Intercepting each TCP message sending/receiving function can obtain statistics such as the message length required for building a weighted dependency graph as described in Section 2.4. The kernel program stores the intercepted message length in a hash table as statistical data to calculate the network throughput.

The procedure for determining the direction of the flow described in Section 2.2 is as follows. Since UDP is a non-connection protocol, there is no procedure to establish a connection like TCP's connect and accept. The listening side must specify the listening port with the bind() system call at startup. The kernel program is attached to the bind(), and the program obtains the port number from the bind() argument and stores it into a table. It looks up the source port in the flow obtained from the UDP message send/receive functions (PROBE__ UDP_SENDMSG, PROBE__UDP_RECVMSG) in the table of the bind() ports. The result of looking up leads to whether the source port is the listening port. If the source port is the listening port, the flow has the incoming direction, otherwise the flow is in the outgoing direction.

For TCP, if the function where the kernel program is called is a function called from the connect system call (PROBE__ TCP_CONNECT), the flow is in the outgoing direction. If it

**Algorithm 1** Algorithm of in-kernel flow-bundling

```
Input: Socket structure $S$, listening ports P
Output: Dump all bundled flows on hash table for storing bundled flows H
new H
function GET_LISTENING_PORT_AND_DIRECTION($S$)
    if P.lookup($S.sport$) then
        return $S.sport$, INCOMING
    else
        return $S.dport$, OUTGOING
    end if
end function
function INSERT_FLOW($S$,$proto$,$msglen$)
    $lport$, $dir \leftarrow$ GET_LISTENING_PORT_AND_DIRECTION($S$)
    $key \leftarrow \{S.saddr, S.daddr, lport, dir, proto\}$
    $stats \leftarrow$ H[$key$]
    if $stats == NULL$ then
        Initialize $stats$
        $stats.msglen \leftarrow msglen$
        H.insert($key$, $stats$)
    else
        $stats.msglen += msglen$
        H.update($key$,$stats$)
    end if
end function
function PROBE__TCP_CONNECT($S$)
    INSERT_FLOW($S$, TCP, 0)
end function
function PROBE__TCP_ACCEPT($S$)
    INSERT_FLOW($S$, TCP, 0)
end function
function PROBE__TCP_SENDMSG($S$,$msglen$)
    INSERT_FLOW($S$, TCP, $msglen$)
end function
function PROBE__TCP_RECVMSG($S$,$msglen$)
    INSERT_FLOW($S$, TCP, $msglen$)
end function
function PROBE__UDP_SENDMSG($S$,$msglen$)
    INSERT_FLOW($S$, UDP, msglen)
end function
function PROBE__UDP_RECVMSG($S$, $msglen$)
    INSERT_FLOW($S$, UDP, msglen)
end function
```

is a function called from the accept system call (PROBE__TCP_ACCEPT), the flow is in the incoming direction. The TCP message sending/receiving functions (PROBE__TCP_SENDMSG, PROBE__TCP_RECVMSG) also cannot identify the direction from the function name because TCP sends/receives messages in both directions. As with UDP, the bind() port identifies the TCP flow direction.

**Step 3** The tracing process placed in the user space obtains a list of bundled flows from the hash table. It also deletes the obtained flows from the hash table. The process is repeated at regular time intervals.

### 3.2 Implementation

We implemented our method using the eBPF assuming Linux (Linux Kernel version 5.6 or later), which is widely used in cloud applications.

We used eBPF to run the kernel program to bundle flows in the kernel. eBPF users can write kernel programs in the constrained C language and execute the bytecode compiled using the C compiler Clang [18] in a sandbox within the kernel.

To intercept the kernel function and system calls, we used Kprobes, a dynamic tracing technology for the Linux kernel. Kprobes can set a breakpoint at addresses in the kernel code and execute pre-defined handlers at the breakpoints. Kprobes and eBPF can attach an eBPF program to a breakpoint of a kernel function. To access sockets, our method in this implementation attaches kernel functions that take a *strcut sock* structure, or a *struct sk_buff* structure as an argument or return value. For TCP, our method attaches the kernel programs to the *tcp_v4_connect* function called from the connect() system call and *inet_csk_accept* function called from the accept() system call, *tcp_sendmsg* function for sending TCP messages, and *tcp_cleanup_rbuf* function called for receiving TCP messages. For UDP, our method also attaches the kernel programs to *ip_send_skb* function called from the sendmsg() system call and *skb_consume_udp* function called from the recvmsg() system call.

To store the bundled flows, our method uses eBPF maps, a generic data structure that allows data to be shared between eBPF programs or between kernel space and user space processes. The eBPF maps support multiple types of data structures, such as hash tables and arrays. Our method also uses eBPF maps to store the listening ports bound by the bind().

To obtain the bundled flows, our method uses the *bpf_mao_lookup_and_delete_batch()* system call to atomically retrieve and delete multiple bundled flows in the table. The system call is repeated at 1 s intervals to keep acquiring the latest converged flows.

We published this implementation as a library in the Go language [*1]. Our library is intended to be embedded in a network-dependency visualization system such as Weave Scope [30].

## 4. Experiments

To confirm the effectiveness of our method, we exprimentally evaluated its CPU overhead and latency overhead.

### 4.1 Experiment Setup

**Machine specifications** We prepared one virtual machine for each client and server in Sakura Cloud [*2]. The hardware specifications of the virtual machines are Intel Xeon Gold 6212U 2.40 GHz 6-core CPU, 16 GiB memory. The OS of each machine is Ubuntu 20.10 Kernel 5.8.0, and the network bandwidth between virtual machines is 1 Gbps.

**Load generation** We implemented connperf [*3], a load generation tool for TCP/UDP, to pseudo-generate the load of the application. It runs in either client mode or server mode and supports both short-lived and persistent connection methods of TCP, on the basis of the description in Section 2.3. The connperf communication over TCP and UDP is a simple echo client and server, where the client sends a message to the server and the server sends back the received message to the client.

**Comparison methods** Current socket-based methods were compared with the proposed method. The snapshot polling method takes snapshots of sockets in the kernel at regular intervals through Netlink [24], a mechanism for communicating messages between the kernel space and the user space, and Process Filesystem (procfs). The streaming method attaches to *tcp_connect_v4* function and *inet_csk_accept* function in TCP to obtain only the flows generated at the time of connection establishment. The in-kernel flow-aggregation method is based on the implementation described in Section 3.2 and includes the both source and destination ports as keys in the hash table for storing flows.

**Evaluation metrics** The proposed method requires small overhead of CPU usage and application latency. CPU overhead is measured as CPU utilization of the tracing process. Latency overhead is the sum of execution times of all eBPF programs executed during a round trip of connperf. The execution time of an eBPF program is obtained from the eBPF statistics maintained by the Linux kernel [16]. The CPU utilization and execution time of the eBPF programs are the arithmetic mean values when the tracing process is executed for 30 s.

**Parameters** The polling-interval set for each method to be compared with the proposed method was set to 1 s. The message length of connperf was fixed to 64 bytes.

To enable the reproduction of the experiments, we opened the programs to automate the experimental procedure available in the public repository [*4].

### 4.2 Experimental Results

#### 4.2.1 CPU Overhead

**Figure 3** compares CPU utilization per CPU core over varying round trips per second of TCP short-lived connections, persistent connections and round trips per second of UDP. We varied the load parameters between 5k and 35k on the basis of the C10K problem [17], which is a scalability problem when the number of simultaneous connections is on the order of 1k.

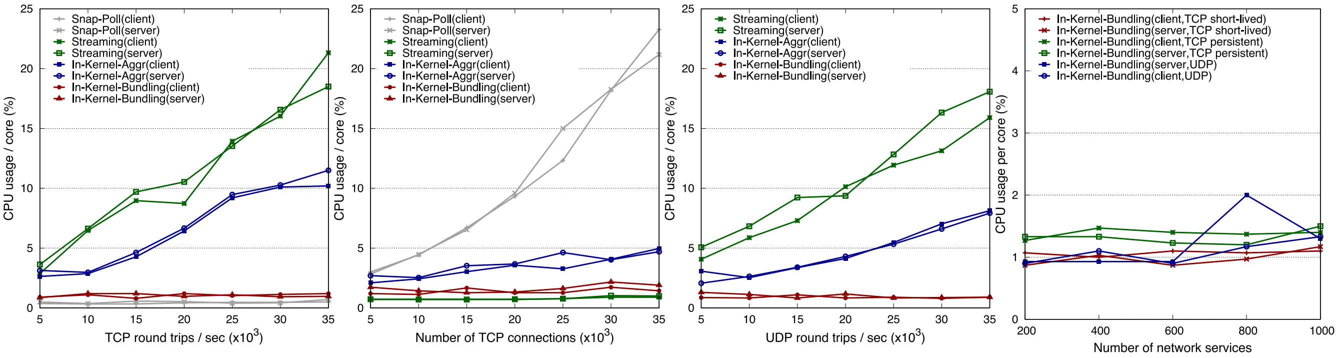
**Short-lived TCP connections** Figure 3 (a) shows that the CPU utilization of the proposed method remained below 1.2%.

[*1] https://github.com/yuuki/go-conntracer-bpf

[*2] https://cloud.sakura.ad.jp

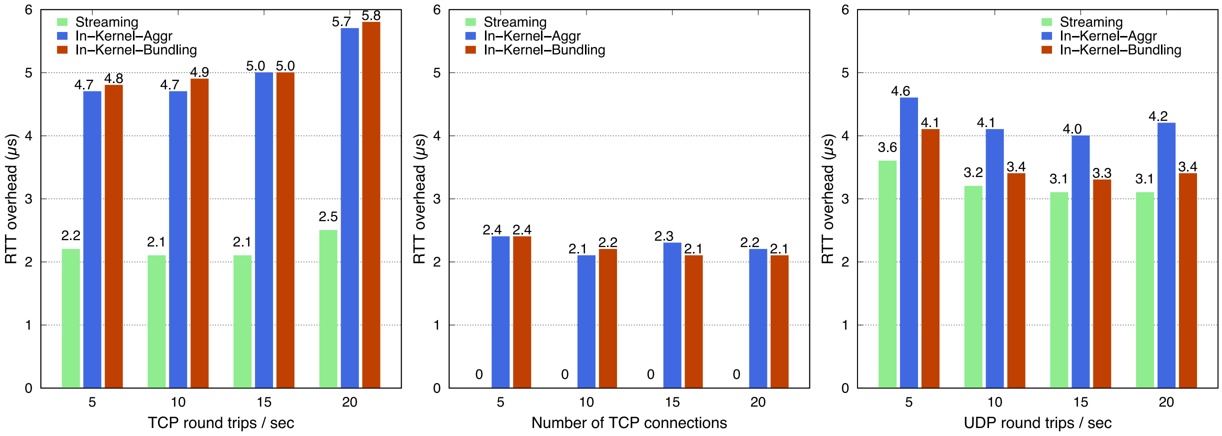
[*3] https://github.com/yuuki/connperf

[*4] https://github.com/yuuki/shawk-experiments



**(a)** Short-lived TCP connections **(b)** Persistent TCP connections **(c)** UDP **(d)** Multiple network services

**Fig. 3** CPU overhead.



**(a)** Short-lived TCP connections **(b)** Persistent TCP connections **(c)** UDP

**Fig. 4** Latency overhead.

The CPU utilization of the streaming method increased from 2.9 to 21.3% in proportion to the increase in the number of round trips. The CPU utilization of the in-kernel flow-aggregation method increased from 2.6 to 11.5%. The CPU utilization of the snapshot polling method was less than 1%, the lowest among the methods. Since the snapshot polling method cannot trace all of the short-lived connections, the number of obtained connections was small, resulting in a low CPU utilization.

**Persistent TCP connections** Figure 3 (b) shows that the CPU utilization of the proposed method remained below 2.2%. That of the snapshot polling method ranges from 3 to 23.3% and that of the in-kernel flow-aggregation ranged from 2.1 to 5.0%. The CPU utilization of the snapshot polling method increased as the number of persistent connections increased because the number of scanning connections increased. The CPU utilization of the streaming method was lower than that of the other methods because this method traces the persistent connections only once, immediately after the start of measurement.

**UDP** Figure 3 (c) shows the same trend in change as that of TCP for short-lived connections. This is reasonable because the UDP processing in connperf is similar to that of a short-lived connection excluding the connection establishment. Due to the limitation of Linux procfs, the snapshot polling method cannot scan UDP; therefore it was not considered for this metric.

With the proposed method, as the number of source or destination network services increased, the number of flows transferred to the user space increased. We define the rate of the number of bundled flows ($B$) to the total number of unique flows ($T$) as a bundling rate ($R$) ($R = 1 - B/T$). To measure the effect of the decreased bundling rate, we emulate the communication with multiple network services by placing a connperf process on a Docker container.

Figure 3 (d) shows the CPU utilization of the proposed method when the number of network services was varied from 200 to 1k. The total number of round trips per second and that of connections were fixed at 10k. The total round trips and connections were distributed to the network services equally. The bundling rate was varied from 0.98 to 0.9 in theory. The CPU utilization of the proposed method remained less than 2% regardless of the number of network services.

### 4.2.2 Latency Overhead

We measured the execution time of the eBPF program for the same environment and parameter variations as in the CPU utilization measurement. **Figure 4** shows the arithmetic mean of the sum of the execution times of all the eBPF programs executed during the round trip. This experiment did not involve measuring the latency of the snapshot polling method since it does not intervene in the communication path as other methods do.

Figure 4 shows that the maximum execution time was 6 μs for each measurement. For TCP short-lived connections, the latency overhead of the proposed method was 0.4–4% higher than the in-kernel flow-aggregation method and 54–58% higher than the streaming method. For TCP persistent connections, the overhead was −7.0–0.7% higher than the in-kernel flow-aggregation method. For UDP, the overhead was 14–25% lower than the in-kernel flow-aggregation method and 8–12% higher than the streaming method. The streaming method had a lower overhead for TCP short-lived connections and zero overhead for TCP persistent connections compared with the other methods. This is because the streaming method only traces the flows during TCP connection establishment.

## 5. Discussions

### 5.1 Evaluating the Overheads

**CPU** The proposed method is the most efficient of the socket-based methods for discovering complete dependencies. This is because the method bundles all the flows into a single flow no matter how much the flows increased. The proposed method has sufficiently low CPU utilization when the number of bundled flows is less than 1k. It can maintain low CPU utilization even when the number of communicating targets increases. This means, for example, that even if 1k network services are deployed under a load-balancer service, the flows generated by the services can be traced with a low CPU utilization. However, in Fig. 3 (d), the flow-bundling rate should approach 0 as the number of network services increases from 1k to 10k. When the flow-bundling rate becomes lower, the performance difference between the in-kernel flow-bundling and the in-kernel flow-aggregation becomes small.

**Latency** The RTT by Internet Control Message Protocol (ICMP) ping between the virtual machines was 300 µs. The round trip overhead was 1/50th the RTT since 6 µs for the proposed method. This is a sufficiently small overhead for the total latency including the network transfer.

### 5.2 Evaluating the Accuracies

To discuss the accuracies of dependency discovery with the proposed method, we take the case where multiple network services with different IP addresses are provided through the same IP address and the case where multiple network services are provided with a single IP address.

**Identifiability of destination servers behind a relay** When a client communicates with multiple servers via the same IP address, depending on the relay method, socket-based methods may not be able to identify multiple servers individually. There are two types of network relays used in the cloud: TCP/UDP session terminating and packet forwarding [15].

TCP/UDP session terminating places a relay network service between a client and servers. Since the relay network service creates sockets, socket-based methods can discover dependencies between clients and servers that communicate with the relay network service, respectively. The proposed method can identify multiple servers with different IP addresses via the relay network service.

Packet forwarding rewrites the packet header with techniques such as Network Address Translation (NAT), and thus the clients and the servers do not know each other's local IP addresses. Because packet forwarding does not create network sockets for relaying, socket-based methods cannot discover relay dependencies. This is an issue common to socket-based methods, regardless of whether flows are bundled or not. To solve this issue, matching the forwarding history with the flows traced by socket-based methods is required, either by tracing packets before and after forwarding, or by looking up the NAT table. The proposed method can identify multiple servers with different IP addresses by unbundling the bundled flows in the user space. The flow bundling multiple servers can be unbundled into the bundled flows per server by matching the forwarding history with the bundled flows because the forwarding history has the relation with the destination IP address and port from the clients and the forwarding destinations.

**Identifiability of source clients in front of a relay** When multiple clients communicate with a single server via the same IP address, identifiability of relay source clients is an equivalent issue about identifiability of relay destination servers.

### 5.3 Limitations

**Server identifiability** When multiple network services are provided on a single server with the same listening port, the proposed method cannot distinguish these network services because of each network service being bundled by the listening port. However, at least in Linux, multiple network services cannot listen on the same listening port of the same IP address. The exception is that network services enable the Linux socket option SO_REUSEPORT [*5] to allow multiple different network services to use the same port. SO_REUSEPORT is designed to improve load balancing on threads by allowing multiple threads in a single network service to each listen on the same port. Even if SO_REUSEPORT is enabled, we believe that multiple network services on a single server are rarely bundled into a single flow.

**Client identifiability** The proposed method cannot distinguish between different clients with the same IP address because it lacks the short-lived ports on the client. Adding the process name of the network service to the uniqueness identifier solves the issue, but it does not address the case of duplicate process names. However, container-based virtualization, which is currently in widespread use, does not recommend to run multiple network services within a single container [10]. The cases in which we want to identify processes with the same process name on the same host individually will become rare in the future.

### 5.4 Timeliness of Dependency Discovery

The snapshot polling and in-kernel flow-aggregation and proposed methods delay the discovery of dependencies by the interval since these methods acquire flows at regular time intervals. However, since system administrators are expected to visually check for dependencies, a delay small enough to be unrecognizable by humans is not a problem. The most common interval for data acquisition is 15 s, followed by 30 s according to a survey on Prometheus usage by the developers of Prometheus [29], a typical cloud monitoring tool. The minimum value of the interval is 1 s. Based on this survey, setting a 1-s interval, as mentioned in

[*5] socket(7), https://man7.org/linux/man-pages/man7/socket.7.html

Section 4 is sufficient for practical use.

## 6. Conclusion

We proposed an in-kernel flow-bundling method to bundle TCP/UDP flows in a kernel to discover the dependencies among network services. The proposed method can trace flows with low overhead for applications using either short-lived or persistent connections. In our experiments, we confirmed that the proposed method can maintain CPU overhead of less than 2.2% even when communicating to less than 1k network services. The latency overhead to the application is 6 µs per round trip at most.

For future work, we will first implement our method to support tracing a containerized network service. We will then develop a visualization system for network dependencies that applies our method. To reduce the setup and management effort of the visualization system, we will design a decentralized architecture without maintaining a central database.

### References

[1] Why persistent connections are bad (online), available from ⟨https://meta.wikimedia.org/wiki/Why_persistent_connections_are_bad⟩ (accessed 2021-04-26).

[2] Alvidrez, M.: Embracing Risk, *Site Reliability Engineering: How Google Runs Production Systems*, chapter 3, pp.25–36, O'Reilly Media, Inc. (2016).

[3] Bahl, P., Chandra, R., Greenberg, A., Kandula, S., Maltz, D.A. and Zhang, M.: Towards Highly Reliable Enterprise Network Services via Inference of Multi-Level Dependencies, *ACM SIGCOMM Computer Communication Review*, Vol.37, No.4, pp.13–24 (2007).

[4] Calavera, D. and Fontana, L.: *Linux Observability with BPF: Advanced Programming for Performance Analysis and Networking*, O'Reilly Media, Inc. (2019).

[5] Chen, P., Qi, Y., Zheng, P. and Hou, D.: CauseInfer: Automatic and Distributed Performance Diagnosis with Hierarchical Causality Graph in Large Distributed Systems, *IEEE Conference on Computer Communications* (*INFOCOM*), pp.1887–1895 (2014).

[6] Chen, X., Zhang, M., Mao, Z.M. and Bahl, P.: Automating Network Application Dependency Discovery: Experiences, Limitations, and New Solutions, *USENIX Symposium on Operating Systems Design and Implementation* (*OSDI*), pp.117–130 (2008).

[7] Clawson, J.K.: Service Dependency Analysis via TCP/UDP Port Tracing, Master's thesis, Brigham Young University – Provo (2015).

[8] Datadog, Inc.: Datadog Network Performance Monitoring (online), available from ⟨https://docs.datadoghq.com/network_monitoring/performance/⟩ (accessed 2021-03-04).

[9] Desnoyers, M.: Using the Linux Kernel Tracepoints (online), available from ⟨https://www.kernel.org/doc/Documentation/trace/tracepoints.txt⟩ (accessed 2021-05-07).

[10] Docker Inc.: Run multiple services in a container (online), available from ⟨https://docs.docker.com/config/containers/multi-service_container/⟩ (accessed 2021-05-06).

[11] Dragoni, N., Giallorenzo, S., Lafuente, A.L., Mazzara, M., Montesi, F., Mustafin, R. and Safina, L.: Microservices: Yesterday, Today, and Tomorrow, *Present and Ulterior Software Engineering*, pp.195–216, Springer (2017).

[12] Fonseca, R., Porter, G., Katz, R.H. and Shenker, S.: X-Trace: A Pervasive Network Tracing Framework, *USENIX Conference on Networked Systems Design & Implementation* (*NSDI*), p.20 (2007).

[13] Gregg, B.: *BPF Performance Tools*, Addison-Wesley Professional (2019).

[14] Han, J., Haihong, E., Le, G. and Du, J.: Survey on NoSQL database, *Sixth International Conference on Pervasive Computing and Applications* (*ICPCA*), pp.363–366 (2011).

[15] K. Gilly et al.: An Up-To-Date Survey in Web Load Balancing, *World Wide Web*, Vol.14, No.2, pp.105–131 (2011).

[16] Kahle, B.: How and When You Should Measure CPU Overhead of eBPF Programs, *eBPF Summit 2020* (2020).

[17] Kegel, D.: The C10K problem (online), available from ⟨http://www.kegel.com/c10k.html⟩ (accessed 2021-03-19).

[18] Lattner, C.: LLVM and Clang: Next Generation Compiler Technology, *The BSD conference*, Vol.5 (2008).

[19] Lin, J., Chen, P. and Zheng, Z.: Microscope: Pinpoint Performance Issues with Causal Graphs in Micro-Service Environments, *International Conference on Service-Oriented Computing* (*ICSOC*), pp.3–20 (2018).

[20] Liu, X., Heo, J. and Sha, L.: Modeling 3-tiered Web Applications, *IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pp.307–310 (2005).

[21] Mavinakayanahalli, A., Panchamukhi, P., Keniston, J., Keshavamurthy, A. and Hiramatsu, M.: Probing the Guts of Kprobes, *Linux Symposium*, Vol.6, p.5 (2006).

[22] Moon, Y., Lee, S., Jamshed, M.A. and Park, K.: AccelTCP: Accelerating Network Applications with Stateful TCP Offloading, *USENIX Symposium on Networked Systems Design and Implementation* (*NSDI*), pp.77–92 (2020).

[23] Natarajan, A., Ning, P., Liu, Y., Jajodia, S. and Hutchinson, S.E.: NSDMiner: Automated Discovery of Network Service Dependencies, *IEEE International Conference on Computer Communications* (*INFOCOM*), pp.2507–2515 (2012).

[24] Neira-Ayuso, P., Gasca, R.M. and Lefevre, L.: Communicating between the Kernel and User-Space in Linux using Netlink Sockets, *Software: Practice and Experience*, Vol.40, No.9, pp.797–810 (2010).

[25] Neves, F., Vilaça, R. and Pereira, J.: Black-box inter-application traffic monitoring for adaptive container placement, *Annual ACM Symposium on Applied Computing* (*SAC*), pp.259–266 (2020).

[26] Sigelman, B.H., Barroso, L.A., Burrows, M., Stephenson, P., Plakal, M., Beaver, D., Jaspan, S. and Shanbhag, C.: Dapper, a Large-Scale Distributed Systems Tracing Infrastructure, Technical Report, Google (2010).

[27] The OpenTelemetry Authors: OpenTelemetry (online), available from ⟨https://opentelemetry.io/⟩ (accessed 2021-03-04).

[28] Tsubouchi, Y., Furukawa, M. and Matsumoto, R.: Transtracer: Socket-Based Tracing of Network Dependencies Among Processes in Distributed Applications, *The 1st IEEE International COMPSAC Workshop on Advanced IoT Computing* (*AIOT*), pp.1206–1211 (2020).

[29] Turnbull, J.: *Monitoring with Prometheus*, Turnbull Press (2018).

[30] Weaveworks Ltd.: Weave Scope (online), available from ⟨https://github.com/weaveworks/scope⟩ (accessed 2021-03-04).

[31] Zand, A., Vigna, G., Kemmerer, R. and Kruegel, C.: Rippler: Delay Injection for Service Dependency Detection, *IEEE International Conference on Computer Communications* (*INFOCOM*), pp.2157–2165 (2014).

**Yuuki Tsubouchi** decided not to finish his M.S. program at Osaka University, Osaka, Japan, in 2013 and became a site reliability engineer at Hatena Co., Ltd until 2018. Since 2018, he has been a researcher at SAKURA internet Research Center, SAKURA internet Inc. He is currently a Ph.D. candidate in informatics at Kyoto University, Kyoto, Japan. His research interests include site reliability engineering for cloud computing. He is a member of IEEE, ACM, and IPSJ.

**Masahiro Furukawa** is a site reliability engineer at Hatena Co., Ltd.

**Ryosuke Matsumoto** received his Ph.D. in informatics from Kyoto University, Kyoto, Japan, in 2017. From 2015 to 2018, he worked for Pepabo Researchand Development Institute, GMO Pepabo, Inc., as chief engineer and a chief researcher. Since 2018, he has been a senior researcher at SAKURA internet Research Center, SAKURA internet Inc. His research interests include OS, middleware, Internet operation technology, and security. He is a member of IEEE, ACM, and IPSJ.