Transtracer: Socket-Based Tracing of Network Dependencies among Processes in Distributed Applications

Yuuki Tsubouchi
SAKURA internet Inc.
Email: y-tsubouchi@sakura.ad.jp

Masahiro Furukawa
Hatena Co., Ltd.
Email: masayoshi@hatena.ne.jp

Ryosuke Matsumoto
SAKURA internet Inc.
Email: r-matsumoto@sakura.ad.jp

***Abstract*—Distributed applications in web services have become increasingly complex in response to various user demands. Consequently, system administrators have difficulty understanding inter-process dependencies in distributed applications. When parts of the system are changed or augmented, they cannot identify the area of influence by the change, which might engender a more damaging outage than expected. Therefore, they must trace dependencies automatically among unknown processes. An earlier method discovered the dependency by detecting the transport connection using the Linux packet filter on the hosts at ends of the network connection. However, the extra delay to the application traffic increases because of the additional processing inherent in the packet processing in the Linux kernel.**

As described herein, we propose an architecture of monitoring network sockets, which are endpoints of TCP connections, to trace the dependency. As long as applications use the TCP protocol stack in the Linux kernel, the dependencies are discovered by our architecture. Therefore, monitoring processing only reads the connection information from network sockets. The processing is independent of the application communication. Therefore, the monitoring does not affect the network delay of the applications. Our experiments confirmed that our architecture reduced the delay overhead by 13–20 % and the resource load by 43.5 % compared to earlier reported methods.

***Index Terms*—Distributed Tracing; Linux; Monitoring; Observability; Site Reliability Engineering**

1. Introduction

The number of accesses from users has been increasing because of the widespread use of web services. Web service providers have been providing services for more than 10 years since web services first became indispensable to people's lives. A single service provider has come to provide various services such as social networks, electronic commerce, and sound and video streaming to respond to various demands by users. When some functions of a certain web service are accessed by other web services, there are cases in which functions are shared by mutual connection via a network. For these reasons, distributed applications that form the foundation of web services have become complex.

Distributed applications for web services are usually built by network communications among OS processes. Network dependencies have become complex along with increasing complexity of distributed applications. System administrators have difficulty understanding network dependency processes because of their complexity. If an error such as a failure or performance degradation occurs in specific processes, then it might propagate other processes that depend on processes through the network. If system administrators add changes to the systems without knowing the network dependencies, then failures might occur throughout a range that is larger than their expectation. Therefore, system administrators must identify the extent to which changes are affected by investigating the dependency. Automatically tracing dependencies unknown to system administrators must reduce efforts for such investigations.

A method [6] reported earlier employs a network packet filter to discover transport connections. This method has no false positive because it discovers the connection that actually occurred. As long as applications use the transport connection supported by Linux kernel, the dependency can be discovered. This method has no false positives. Therefore, it can trace exhaustive dependencies without the need to change the application code. Tracing the dependencies of unknown processes is suitable because of its exhaustivity. However, intervention in packet processing by a packet filter in the Linux kernel adds extra delay to the application processing.

As described herein, we propose an architecture to trace dependencies exhaustively among unknown processes for Linux, an OS that is widely used for servers. This architecture monitors network sockets containing connection information, which are the termination points of TCP connections. Socket monitoring is independent of packet processing in the network stack because a process for monitoring sockets only reads the connection information included by sockets. Therefore, this architecture does not affect the delay of the applications. The number of sockets per unit time does not exceed the number of packets per unit time because a connection includes multiple packets. A socket is opened using a connection. Therefore, socket monitoring is more resource-efficient than packet filtering. However, this architecture does not trace the dependencies built by network protocols other than the transport layer in

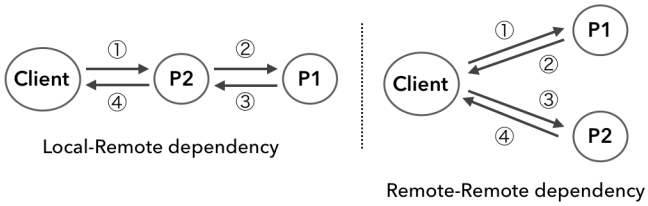

Figure 1: Local–remote dependence and remote–remote dependence.

the OSI reference model: it only traces a TCP connection.

The remainder of this paper is organized as follows. Section 2 provides background and related works. Section 3 presents details of our architecture. Section 4 presents experimentally obtained results of performance measurement. Section 5 concludes this paper.

2. Background

2.1. Definition of dependencies

As described in this paper, we adopt the same definition of dependency among processes as that used for a study reported by Zand [15]. A process P_2 depends on process P_1 if the delay, degradation, or failure in process P_1 directly or indirectly causes delay, degradation, or failure in process P_2. Based on this definition, a study by Chen et al. [5] classified process dependencies as local–remote or remote–remote.

Figure 1 shows each dependence. The number in the figure 1 represents the order of request and response. First, process P_2 has a local–remote dependency on process P_1 if process P_2 must contact process P_1 during request processing from a client. Then, process P_2 has a remote–remote dependency on process P_1, which is on remote–remote if a remote client must access to process P_1 first to access process P_2.

2.2. Related works

Approaches of three types have been proposed for tracing dependencies in distributed applications: packet-based approaches, connection-based approaches, and request-based approaches.

2.2.1. Packet-based approach. Packet-based approaches collect network packets from existing traffic and discover dependencies by analyzing information such as the source and destination hosts and ports on the packet header, and the packet transmission and reception timestamps. It estimates the dependencies by finding a pattern that correlates with traffic characteristics among processes by particularly addressing a time difference in the traffic flow among processes.

A salient advantage of the packet-based approach is that no change to existing applications is necessary. Furthermore, this approach requires no installation of additional software for the end host. The only change is that a network switch must be changed so that packets can be monitored with a switch.

However, if all packets are monitored, then the cost of packet analysis processing increases considerably when the packet flow rate is large. In general, when sampling packets the cost of packet analysis is reduced by sampling packets, but the dependencies included in small traffic might be missed.

The degree of correlation is calculated when a correlated pattern is discovered. Therefore, the presence or absence of a dependency is expressed as a probability. This approach has false positives that indicate incorrect dependencies even though they are not actually mutually dependent. In addition, system administrators must tune the threshold because the dependencies are determined based on a preset threshold for the degree of correlation.

Sherlock [1], Orion [5] and NSDMiner [10] use only packet information of existing traffic. Macroscope [12] combines network packets collected by the host or network device with transport connection information related to the end host. Rippler [15] injects delays into network traffic and inspects the propagation of the delays. Thereby, it reduces false positives.

2.2.2. Connection-based approach. The connection-based approach traces dependencies by obtaining transport connection information related to the host that terminates the connection. Unlike the packet-based approach, this approach does not identify correlated patterns, but discovers dependencies based on the actual connection. Consequently, this approach has no false positives.

Additional effort is required of system administrators because one must deploy additional software on end hosts to acquire transport connection information.

A study reported by Clawson [6] has presented a connection-based approach that uses Linux packet filters to discover dependencies by monitoring transport connections that terminate on the host. As long as an application uses a transport protocol supported by Linux, it can detect dependencies. Using a packet filter entails the shortcoming of adding extra delay to application processing because it intervenes in packet processing in the Linux kernel.

2.2.3. Request-based approach. The request-based approach traces which path in the system an application layer request follows. An identifier is assigned to each request and is embedded in the communication content. It is then propagated to subsequent processes. Therefore, this approach can trace, depending on the identifier, the system process through which the request was processed.

The salient advantage of the request-based approach is that it has low false positives because the dependence is found based on the actual connection, as in the connection-based approach. Furthermore, it is possible to trace information according to the processing contents of the application and the application layer protocol.

However, the request-based approach embeds an identifier in the request, which adds extra delay to the application response time. By sampling the embedding of identifiers on a request basis, the delay can be reduced, but false negatives are high for dependency detection. Furthermore, embedding the identifier takes some time to change the application or to place software on the communication path.

Pinpoint [4], Magpie [2], X-Trace [8], and Dapper [13] change the application code and add processing to embed the request identifier. Service mesh [9] assigns and embeds an identifier to the request without changing the application by placing proxies called sidecar proxies in microservices [7] or other process units and communicating with other services via the proxy.

3. Proposed Method

As described in this paper, our purpose is dependency tracing to identify the scope of change when system administrators make changes to components in complex distributed systems that include unknown processes. Even if they change the code or configuration of a specific request processing of an application running on a server, the application memory space is shared. Therefore, the minimum unit for tracing dependencies is a process on the host because effects of the changes might propagate if it is a process within the same process.

Based on characteristics of each existing study, an architecture must be found with low overhead and both true positives and negatives. This paper describes our proposed Transtracer, a socket-based approach: monitoring network sockets containing the connection information in the Linux kernel.

Socket monitoring only reads the connection information that is already held by the socket. Therefore, the monitoring processing is independent of the application communication process. Because socket monitoring does not add extra processing to the packets in application communication path, it reduces delay overhead compared to that by the packet filter.

3.1. Details of the proposed method

Figure 2 presents how to retrieve socket information for TCP connections. When the Tracer process runs on the host, the Tracer process queries the Linux kernel and obtains a snapshot of the active TCP connection status from the socket corresponding to each connection. At the same time, the Tracer process acquires the process information corresponding to each connection. Then it links each connection and each process.

Transtracer must poll, acquiring snapshots of connection status periodically, to continue to detect and acquire new connections. However, polling delays new connection detection by the polling time interval. As described in this paper, the polling interval is set in seconds. The delay time for detection is also in seconds. Even if a new connection can be detected in a time shorter than one second, system

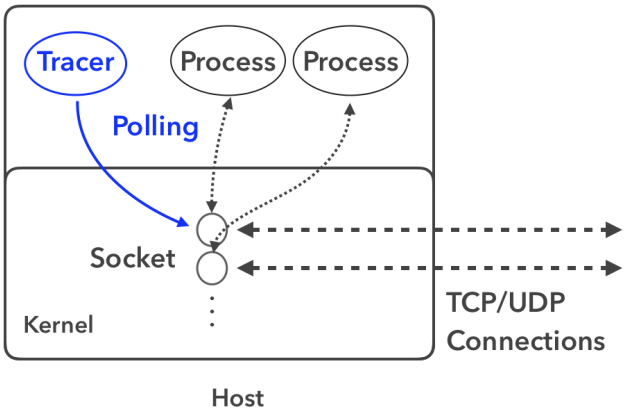

Figure 2: Retrieving connection information from sockets.

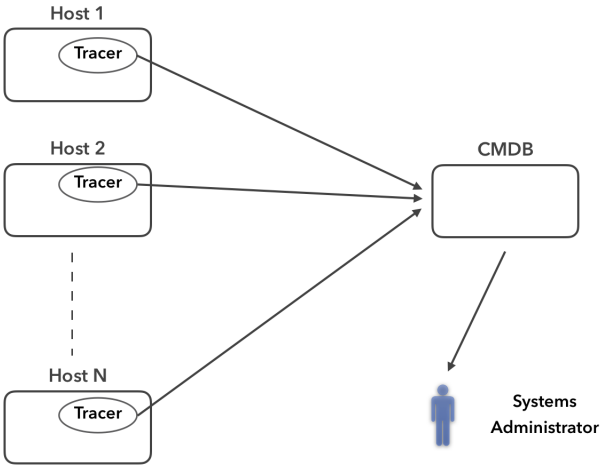

Figure 3: System configuration.

administrators cannot recognize it. For that reason, polling time by units of seconds is sufficient.

Polling might miss short-lived connections within the polling time interval. Actually, decreasing the polling interval leads to an increase of the hardware resource consumption per unit time, but it allows detection of shorter-lived connections. The polling interval is assumed to be in seconds. Therefore, the lifetime of the detectable connection is assumed to be in seconds.

Figure 3 presents the system configuration for matching the connection information related to multiple hosts and for creating a dependency graph. Tracer running on each host sends connection information to the central Connection Management DataBase (CMDB). The system administrator or each host queries the CMDB using information that uniquely identifies the target host or process and the time range as parameters. Thereby, it obtains the dependency associated with the target. Dependencies can be expressed as a graph structure for each process connected to the network, assuming that the process is a node and that the network connection is an edge. Details of the data structure using the relational database are described in section 3.2.

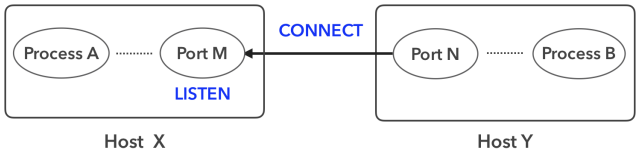

Figure 4: Recognition of connection direction.

Figure 4 portrays a method for identifying dependency direction. A TCP connection is divided into the side of requesting a connection and the side of listening on a connection as follows. Dependency direction is determined. If process A on host X is listening on port M and process B on host Y connects to port M on host X, then process B will fail to connect without process A. Thereby, process B depends on process A.

If all TCP connection events that can be acquired on the host are collected, then the number of connection information increases and the number of records on the CMDB increases. To reduce the number of records, Transtracer reduces redundant connection information. Each process might connect to another process using an ephemeral port, which is a random source port assigned by the OS. Therefore, a process listening on a specific port might be connected from a client of a specific process using multiple random source ports. However, dependency detection does not need an ephemeral port as unique connection information because a Tracer process aggregates ephemeral ports to which the same listening port is connected into a single flow, and sends it to CMDB.

3.2. Implementation of the proposed method

We implement the Tracer process on the premise of Linux (Linux Kernel 4.15) because Linux is used widely as an OS for servers. We implement the CMDB on PostgreSQL (version 11.3) because PostgreSQL is a relational database that is used widely for database management systems and because PostgreSQL has an easy to manage IP address because PostgreSQL has network address types as data types.

The inet_diag module that monitors the transport protocol socket is used to acquire the TCP socket information. The inet_diag module can obtain socket information such as the destination IP address, destination port number, source IP address, source port number, connection status, and socket inode number. The interface for accessing the inet_diag module includes the file system format Process Filesystem (procfs) and Netlink [11], which is a mechanism for message communication between the kernel space and user land in Linux. Our implementation uses not procfs but Netlink, which is accessible at higher speeds because parsing file contents is not necessary, unlike procfs.

The socket information from the inet_diag module does not include information about the process which owns the socket. Our implementation compares the inode number of the socket information with the inode number of the process.

TABLE 1: Experiment environment

Role	Item	Specification
Client	CPU	Intel Xeon CPU E5-2650 v3 2.30 GHz 2core
	Memory	1 GBytes
	Benchmarker	wrk 4.1.0-4
Server	CPU	Intel Xeon CPU E5-2650 v3 2.30 GHz 4core
	Memory	1 GBytes
	HTTP Server	nginx 1.17.3
CMDB	CPU	Intel Xeon CPU E5-2650 v3 2.30 GHz 1core
	Memory	1 GBytes
	Database	PostgreSQL 11.3

If they match, it then links the socket and process. The inode number of a process is obtainable from the file under /proc/[pid]/[fd]/ on procfs.

Our implementation stores dependency data in the PostgreSQL server, which is used as CMDB. A table for storing the node (processes table) and a table for managing the edge (flows table) are required because a dependency is expressed as a directed graph configured with a process as a node and a connection as an edge. However, because of aggregation of connections, the active open side node which has aggregated ephemeral ports has no single port number, although the passive open side node has a listening port number. Therefore, because representation of the active open side and the passive open side with the same schema is difficult, the active open side node and the passive open side node are separated as different tables (actibe_nodes and passive_nodes tables).

We developed the implementation above under the name Transtracer [14]. Transtracer includes two commands: *ttracerd* runs as a daemon on the host, collects connection information of sockets and saves in the CMDB; *ttctl* is used to query the CMDB and obtain dependencies. Separating the command on the writing side and the reading side in this way specifies the argument options and avoids confusion of the user. A Transtracer user sets up the PostgreSQL server as the CMDB, starts up ttracerd on all hosts, and then uses ttctl to query the CMDB after accumulating dependencies in the CMDB.

4. Performance Evaluation

To confirm the Transtracer architecture's effectiveness, we evaluated the delay overhead and the resource load that Transtracer imposes by the implementation presented in the 3.2 section.

4.1. Experiment environment

Table 1 presents the experiment environment. We constructed each role of the experiment environment one by one on virtual servers of Sakura Cloud[1]. The OS of each role is Ubuntu 18.04.2 Kernel 4.15.0. The bandwidth among virtual servers is 1 Gbps. We ran the ttracerd process on Server, and benchmarked nginx on Server by wrk[2], an HTTP benchmarker, on Client.

1. https://cloud.sakura.ad.jp
2. https://github.com/wg/wrk

We configured the benchmarking environment of Clawson's study [6] as follows, which is an early method of a connection-based approach, for comparison with Transtracer. In the environment, TCP connection logs detected by iptables, a Linux packet filter, on the Server were streaming to systemd-journald, the log management process adopted in the default of Ubuntu 18.04. We performed the same benchmark by wrk.

We classified a packet filter method for detecting connections into two methods: NEW filter and the established (ESTB) filter. The NEW filter method leaves logs only when a new connection is made from a client by iptables. NEW filter method has a small log volume, but reused connections cannot be detected later. The ESTB filter method keeps logs of all packets that flow from the already established connection from Client. The ESTB filter method has a larger log volume than the NEW filter method and has no detection omission. The experiment undertaken for Clawson's study adopted ESTB filter method and sampled packets at only 20 packets per minute. Sampling might not detect connections with low packet rates. Therefore, we set it to invalidate sampling.

The configurations common to each subsequent experiment are organized. nginx only returns a statically placed 612-byte HTML file. The number of nginx worker processes was set to be 2 out of 4 cores so that only nginx would not use up the CPU cores. The thread count of wrk was set to 1; wrk ran a benchmark of 60 s. To keep the number of connections during one benchmark as fixed, we enabled HTTP Keep-Alive on both nginx and wrk so that the value of the connection timeout was set to a value larger than the benchmark time. To measure the CPU utilization of each process, we used the Linux pidstat command and set the measurement interval specified in pidstat to the same value as the polling interval. Unless otherwise specified, we set the polling interval when the ttracerd process obtains connection information from the socket to 1 s. The resource load values measured hereinafter are the average values of five consecutive measurements during one benchmark.

4.2. Experiments

First, we evaluated the delay overhead that Transtracer allocates to the application on the server being traced. We graphed the average response time of each of the four conditions: the state without tracing (normal), the state where the ttracerd process is started (ttracerd), the ESTB filter, and the NEW filter when the number of simultaneous connections of wrk was increased from 5,000 to 20,000. Figure 5 portrays the results of this experiment. Transtracer was 13–20 % smaller than the value of ESTB filter and 5.8–16.2 % smaller than the value of NEW filter for every number of connections. Results show that the response time of Transtracer was 1.7–13.4 % longer than in the normal state. Therefore, Transtracer can reduce the delay overhead of the response time compared to earlier methods.

Second, we evaluated how much overhead the resource load would have on the servers to be traced if not using

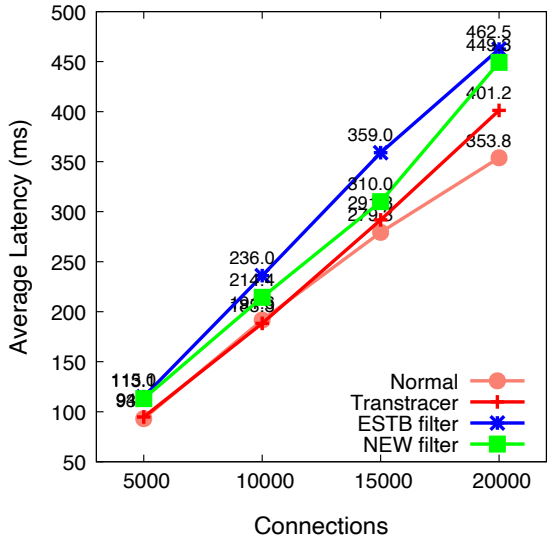

Figure 5: Response time.

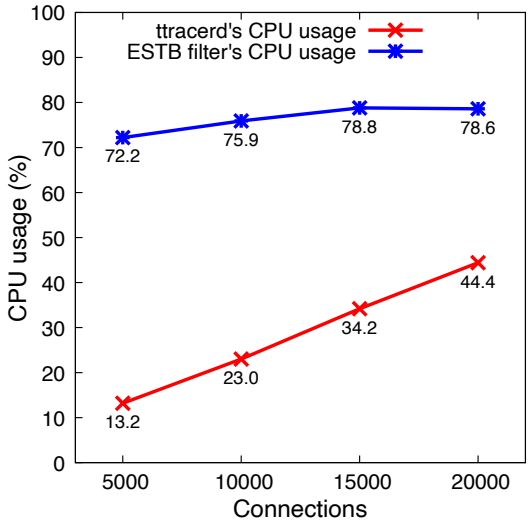

Figure 6: CPU usage.

Transtracer to trace dependencies. We graphed the change of CPU usage rate of the ttracerd process and the ESTB filter. The change of connection information acquisition time was graphed when the number of concurrent connections of wrk was increased from 5,000 to 20,000. Figure 6 presents the results of this experiment. The ESTB filter method has a CPU utilization of more than 70 % for every number of connections, whereas the ttracer process CPU utilization is 44.4 %, even at 20,000 connections. Results demonstrate that Transtracer reduces the CPU load by 43.5 % compared to the ESTB filter method.

The NEW filter is superior to other methods in terms of CPU utilization in an environment where connections are reused. That is true because the NEW filter, which matches only new connections, uses the CPU only when the

connection is established at the start of the benchmark and each CPU load of the other methods is applied constantly during the benchmark. However, comparison with different measurements which measure the average value of CPU utilization during the benchmark and the method of measuring the instantaneous CPU utilization only at the start of the benchmark is difficult. Therefore, Figure 6 does not display the CPU utilization of the NEW filter.

These results suggest that the Transtracer architecture functions effectively because the response delay overhead and the resource load overhead are less than those of earlier methods. Even in a high-load environment with numerous connections, Transtracer is useful with low overhead. Therefore, Transtracer can achieve exhaustive dependencies without compromising the operational environment of the real environment, such as not tracing the dependencies only on high-load hosts. However, short-lived connections lasting only seconds might be missed because of polling. In web services, processes might reuse connections to reduce processing costs associated with each connection because of the use of HTTP/2.

5. Conclusion

As described in this paper, we proposed a novel architecture, Transtracer, a network-dependency tracing program among processes unknown to system administrators by monitoring sockets, which are connection termination points on Linux, in distributed applications. Transtracer can trace exhaustive dependencies without affecting the communication delay of applications while reducing the resource consumption load on target hosts. Results of the experiment confirmed that the response delay overhead was reduced by 13–20 %. The resource load was reduced by 43.5 % compared to methods reported earlier.

Future works are described below. First, by detecting dependencies reliably, even for short-lived connections, we will implement socket monitoring by streaming connection events from sockets with Linux extended Berkley Packet Filter (eBPF). Second, we plan to support container virtualization environments. Finally, to reduce the cost of CMDB setup, we consider a method by which the Tracer process on each host distributes and stores connection information related to localhost.

References

[1] Bahl P, Chandra R, Greenberg A, Kandula S, Maltz DA, 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.

[2] Barham P, Isaacs R, Mortier R, and Narayanan D, "Magpie: Online Modelling and Performance-aware Systems," presented at the 17th Workshop on Hot Topics in Operating Systems (HotOS), 2003, pp. 85–90.

[4] Chen MY, Kiciman E, Fratkin E, Fox A, and Brewer E, "Pinpoint: Problem Determination in Large, Dynamic Internet Services," presented at the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 2002, pp. 595–604.

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

[6] Clawson JK, "Service Dependency Analysis via TCP/UDP Port Tracing," Master's thesis, Brigham Young University – Provo, 2015.

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

[8] Fonseca R, Porter G, Katz RH, and Shenker S, "X-Trace: A Pervasive Network Tracing Framework," presented at the USENIX Conference on Networked Systems Design & Implementation (NSDI), 2007, pp. 20–20.

[9] Li W, Lemieux Y, Gao J, Zhao Z, and Han Y, "Service Mesh: Challenges, State of the Art, and Future Research Opportunities," presented at the IEEE International Conference on Service-Oriented System Engineering (SOSE), 2019, pp. 122–1225.

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

[11] Neira-Ayuso P, Gasca RM, 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.

[12] Popa L, Chun BG, Stoica I, Chandrashekar J, and Taft N, "Macroscope: End-Point Approach to Networked Application Dependency Discovery," presented at the International Conference on Emerging Networking Experiments and Technologies (CoNEXT), 2009, pp. 229–240.

[13] Sigelman BH, Barroso LA, Burrows M, Stephenson P, Plakal M, Beaver D, Jaspan S, and Shanbhag C, "Dapper, a Large-Scale Distributed Systems Tracing Infrastructure," Google, Tech. Rep., 2010.

[14] *Transtracer*, https://github.com/yuuki/transtracer v0.1.1.

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