Socker: Network-application Co-programming with Socket Tracing

Dong Guo
Tongji University

Shuhe Wang
Tsinghua University

Y. Richard Yang
Yale/Tongji University

ABSTRACT

The emerging applications and networks are putting forward more and more demands for bi-directional awareness between applications and networks, and how to integrate applications and networks has been studied a lot in recent years. In this paper, we observe that socket is the waist between applications and networks, thus we explore the possibility of integrating applications and networks at the socket layer for the first time ever. We implement a first-ever socket level network-application integration framework Socker based on eBPF. By associating sockets with network control functions, programmers can realize flexible routing control based on the application logic, as well as dynamic application logic adjustment based on the network states. In our preliminary evaluation setting, the result shows that the application built on Socker achieves 28.5% request time reduction compared with the traditional socket based implementation on average.

CCS CONCEPTS

• **Networks** → **Programming interfaces**;

KEYWORDS

Software-Defined Networks; Socket; Network-Application Integration

ACM Reference Format:
Dong Guo, Shuhe Wang, and Y. Richard Yang. 2021. Socker: Network-application Co-programming with Socket Tracing. In *ACM SIGCOMM 2021 Workshop on Network-Application Integration (NAI '21), August 27, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3472727.3472799

1 INTRODUCTION

Today's distributed applications become more and more customized. To be more competitive in the application market, they require high performance with various performance metrics. Similar to applications running on standalone machines, which optimize themselves from using the details of underlying operating system, it is a reasonable analogy that distributed applications could also improve performance from knowing the control logic of the underlying network. On the other hand, current networks are also becoming increasingly "intelligent". SDN paradigm and NFV-enabled network management and orchestration [2] enable network operators to define sophisticated policies to run networks in a flexible and dynamic way. Combined with the increased programmability in network, it also seems possible to embed application specific requirements in network control logic to provide differentiated services to the applications.

Therefore, the coupling of applications and network control has become popular over the past few years. On the one side, great effort has been put in making the network aware of the application's requirements. For example, Google's SDN networks [4, 11] have already implemented control functions to accumulate application data and make TE decisions on the data. On the other side, certain applications also have techniques to adapt themselves to the network status, which is influenced by network control. DASH-based applications [10] adjust their bitrates based on current bandwidth. And in cloud gaming [12], ROI detection algorithms are based on one-side delay. What's more, there are also studies that combine applications and network control in a closed loop. For example in Minerva [8], the congestion control algorithm is based on user QoE, and rate selection on the other hand is according to network bandwidth.

However, all the above work assume application's core function logic (referred to as AF) and network control function logic (referred to as NF) are separately located in different contexts. The internal details of AF and NF are implicit to each other. To have close cooperation, they must additionally summarize their information into sketches and exchange the sketches with each other. This will bring a painful dilemma as small sketches lead to loss of information, and the cooperation will be less efficient, while large sketches could cause large overhead in making the sketches and passing the sketches, as well as expose too much information which may raise security issues for both parties. Besides, they all just work for one certain type of applications or on certain type of network control. There lacks a general framework for joint programming of applications and network control.

We propose a different idea from all these previous work: *jointly locate AF and NF which controls corresponding AF's flows under the same end host context.* This way, applications and network control can directly exchange information without any information loss, so the overall performance will be greatly improved. Also, since they are located under the same context at end hosts, the security concern that applications could leak important information to the outside no longer exists. NF could be the intermediate point for its corresponding AF to communicate network policies with the outside network, while keeping AF's application details inside. Finally, this proposal depicts a unified framework for both AF and NF inside the distributed system. In the future when the network as a service becomes popular, it helps application users to better understand and manage the whole system.

NAI '21, August 27, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-8633-3/21/08.
https://doi.org/10.1145/3472727.3472799

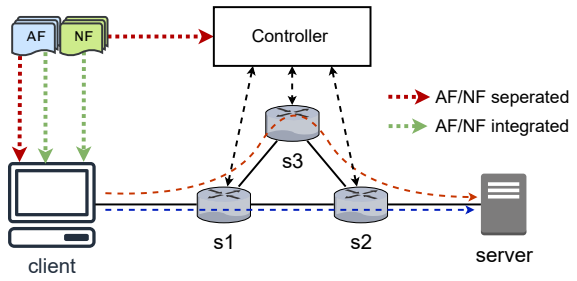

Figure 1: An motivating case

Promising as it sounds, there still exist challenges to bind AF and NF under the same end host context:

Complex dependency: Separate AF and NF codes are transformed into one tightly coupled context. But all the complicated branches and conditions in original codes make it hard to detect all the points where AF and NF have interactions. An AF may generate multiple flows at different times under different situations, and an NF may have multiple rules that may be enabled upon different events. The complexity to statically analyze which NF rules are responsible for which AF flows is high.

Quick synchronization: Previously, most AF and NF independently run in their own context, and periodically pull information sketches from the other. When there are abrupt changes that happen in the middle of two pulls, they cannot react to synchronize the states of each other in time.

We notice that AF and NF both interact with sockets. AF call sockets to send/receive flows while NF set up socket parameters to express control logic. Therefore, we propose Socker, a socket level framework, to solve these challenges. To resolve **complex dependency** of AF and NF, **dynamic flow tracking** is used to trace underlying sockets, and dynamically bind AF and NF. For **Quick synchronization** between AF and NF, **protocol independent reaction** is used for quick response to changes at runtime. In our preliminary evaluation setting, the result shows that the application built on Socker achieves 28.5% request time reduction compared with the traditional socket based application on average.

In summary, we make the following contributions:

- We motivate our design and explain why it is better compared with strawman solutions. (§2).
- We propose the design of Socker. (§3).
- We implement a prototype of Socker and conduct a preliminary evaluation. (§4, §5).
- We summarize prior work on coupling of application and network (§6).

2 MOTIVATION

2.1 Motivating case

We start with a simple setting to show it is necessary to locate AF and NF together. The client runs an application that uploads a certain amount of files to the server. They are connected with a small network with 3 switch nodes, as in Figure 1. The path s1-s2 has shorter latency but smaller bandwidth, while s1-s3-s2 is instead the opposite. AF here is to choose a compress level to compress

```
compressLevel = 0
nab.set('mimeList', ['text/plain'])

@nab.sub('mimeType', 'mimeList', 'topo')
def NF(mimeType, mimeList, topo):
  if mimeType in mimeList:
    path = maxBWPath(topo)
  else:
    path = shortestPath(topo)
  nab.pub('path', path)
  return path

@nab.sub('path', 'filesize')
def AF(path, filesize):
  bw = getBW(path)
  if (filesize / bw) > QOS_THRESHOLD:
    compressLevel = computeCL(filesize, bw)
  else:
    compressLevel = 0

def file_upload(file):
  nab.set('filesize', getSize(file))
  nab.set('mimeType', getMime(file))
  s = socker(nf=NF)
  s.connect((SERVER, PORT))
  if compressLevel > 0:
    file = gzip.compress(file, compressLevel)
  s.send(file)
  s.close()
```

Figure 2: The code snippet of a file upload program with dynamic gzip compress level using Socker

the files. The higher the level is, the smaller the transmitted files would be, and of course, the longer the compression time would be. NF here is to select a path from the client to the server for file uploading. The goal is to finish uploading as soon as possible.

The total uploading time consists of the compression time and the transmission time. Lower compression level means shorter compression time, but also larger file size in transmission. Besides, network path bandwidth and latency also influence the transmission time. Therefore AF and NF should cooperate wisely to decide compression level and path in a joint way.

When AF and NF are separate, as shown with red arrows in Figure 1, they may not be able to exchange information efficiently. In this case, ideally with no congestion, AF will choose a low compression level and NF will choose s1-s2 to speed up uploading. But if AF suddenly starts to send a large file without notifying NF to switch path in time, s1-s2 will soon be overloaded. On the other hand, if s1-s3-s2 is the backup path for multiple AF, and all their short paths receive congestion suddenly, NF will reroute all the flows through s1-s3-s2 then. Without notifying AF to raise compression level to decrease sending rates, s1-s3-s2 will soon receive huge accumulated volume of flows and also get congested. To solve such a mess, users should write an additional protocol to exchange information between AF and NF, and rewrite their codes to plugin the protocol, which is arduous.

On the contrary, if we integrate AF and NF in the same context at end hosts, as shown with green arrows in Figure 1. Specifically, the network control programs (NF) are moved to the end hosts, AF and NF can respond to any state changes of each other in a quick and reactive way, thus an end host becomes a "controller", but can only control the flow originated from itself. As shown in Figure 2, all we need to do is just add a few annotations with shareable variables to NF and AF. The AF determines the file compressing level based on the network states, and the NF computes the routing path (*maxBWPath* or *shortestPath*) based on the file type from AF states. This is simple to use without modifying the original codes and adding extra communication protocol.

2.2 Strawman solutions

Next we discuss the design choice when binding AF and NF at end host. As mentioned in section 1, to handle **complex dependency** and **quick synchronization** of AF and NF, the key challenge is to handle communication efficiently. Similar to inter-process communication, AF and NF should exchange information through certain shared objects. For strawman solutions, we have:

User space implementation: shared objects can be created as global variables in user space. This method is easy to implement and user space are flexible to support. However, AF doesn't have visibility into the kernel space, so it doesn't know the underlying network details, i.e. socket feature and status, of a flow it generated. Then, NF is unable to uniquely handle this flow as its network feature is not written by AF to the shared objects.

Kernel space implementation: eBPF [3] offers a special kind of data structure called eBPF maps, which are globally accessible among kernel space and user space. By virtue of eBPF maps, we can associate AF with the socket it calls at runtime, and store the information into eBPF maps so NF can check and control uniquely with the matching socket. However, eBPF maps essentially take up a limited region of kernel memory, and the map types are constrained by kernel, in addition, the maps do not support pub/sub APIs to perform automatic re-execution. So this solution is not suitable for our system.

Our design combine the two strawman solutions. For AF, the shared objects in user space could store application-level information; and eBPF maps could store network-level information, which is relatively small as only key-value pairs of application and socket matching relation need recording. We explain this design in details in next section.

3 SYSTEM DESIGN

In this section, we first present our observations driving the design, then we propose our design of Socker, and we dive into the detail by describing the case of TCP connection in Socker.

3.1 Design Observations

Programmers write AF to create sockets and NF to control the network. A socket can bind one NF to specify the desired control for the data written into the socket (see example in Figure 2). However, the eventual flow created by a socket can only be determined at runtime (e.g., the socket() parameters are variables or ephemeral ports allocation). Meanwhile, the execution result (e.g., a route) of

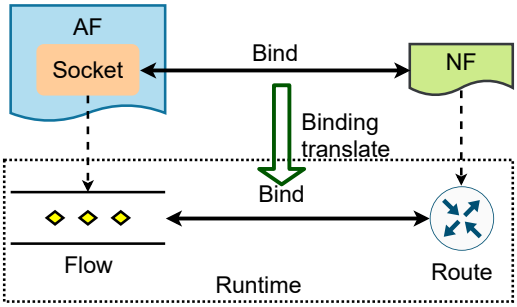

Figure 3: The AF/NF binding to flow/route binding

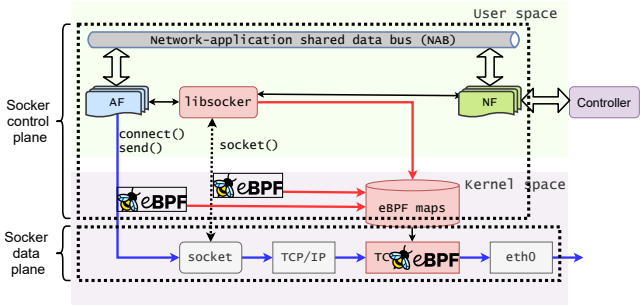

Figure 4: The overall architecture of Socker

NF is also unknown before running. In high-level abstraction, what Socker does is realizing the binding translation from AF/NF binding at programming to the flow/route binding at runtime, as shown in Figure 3. To this end, we have the following observations to drive the design of Socker.

The communication between AF and NF should be protocol independent. When we design the communication mechanics of AF and NF, a natural way we thought is by direct function call between them. However, this leads to a strong coupling between AF and NF in the perspective of the protocol between AF and NF, any change with the parameter leads to a lot of changes of the invokers. Also, it is hard to realize automatic re-execution of AF and NF when one of the variables state inside AF/NF changed. Thus, we adopt the observable design pattern between AF and NF, the shared objects are stored in a stand-alone program-level data bus equipped with pub/sub APIs.

The flow created by a socket must be tracked in the kernel at runtime to get the ephemeral port. Consider the creation of a TCP flow, before invoking the *connect()* system call, the local port of the socket is not determined. However, the *connect()* system call leads to TCP three-way handshake in the kernel, thus a set of packets already be sent after *connect()* system call returns back, leading to the out of control of handshake packets. So we must track the flow of a socket at kernel space, to this end, we choose to use eBPF toolset to trace the system calls in the kernel.

3.2 Overall Architecture

The overall architecture of Socker consists of the control plane and the data plane as shown in Figure 4. The red arrows indicate the control paths and the blue arrows are the data paths.

Socker Control Plane. The control plane of Socker is responsible for three tasks.

(1) Providing a shared data layer for AF and NF and trigger re-execution automatically.
(2) Keeping track of the context information (i.e., user space sockets) of the application.
(3) Synchronizing the desired network states (e.g., the desired path for a flow) into the data plane.

These are achieved by an *network-application shared data bus* (NAB), a language-dependent high-level library *libsocker*, and a set of eBPF programs. In order to achieve interaction between AF and NF and realize automatic re-execution when the state changes, we design the NAB as a data bus, thus AF and NF can share their states in a protocol-independent manner. Also, our NAB adopts the observable design pattern with pub/sub APIs that automatically register the AF/NF as observers to their dependent states, thus any changes to the states can trigger the re-execution of AF/NF. The network states is pulled by the NF from the controller and published to NAB. The *libsocker* wraps the original user level socket functions and a set of eBPF programs attached to system calls. Applications use *libsocker* to create sockets, thus the process runtime information (e.g., process id, socket file descriptor) can be captured by Socker. Also, the eBPF programs track the kernel functions (e.g., sock_alloc_file(2)/tcp_connect(2)) related to the sockets created by applications, and we store the tracked states in kernel eBPF maps. The eBPF programs are attached to socket system calls, we track the arguments passed to the system calls and store the information to eBPF maps.

In addition, the synchronization between the NF and the network policy requires certain mechanics to allow the hosts to inject network configurations into the data path. For example, the NF communicates with the network controller using a dedicated channel. Since the source routing technique can achieve end-host defined routing without the engagement of the controller, which can greatly simplify our setting, thus we assume the network supports source routing, and we describe the architecture of Socker based on the source routing setting.

Socker Data Plane. The Socker data plane is responsible for applying the desired actions returned by NF to the outbound packets, this is done by attaching an eBPF program to the egress traffic control (TC). For example, in a source routing network, if the NF is a routing control function and returns a path for the flow, the eBPF will encapsulate the packets of the flow before emitting to the network interfaces.

Socker eBPF Maps. The main goal of the eBPF maps in Socker is to store the mapping from the flow created by user space sockets to the execution results of NF. However, there is no straightforward way to link them (e.g., user space cannot directly get the kernel socket structure address). To this end, we design four basic eBPF maps of type BPF_MAP_TYPE_HASH as shown in Table 1, and use the inode number of the kernel socket structure as a bridge to link user space sockets and kernel space socket, because the user space

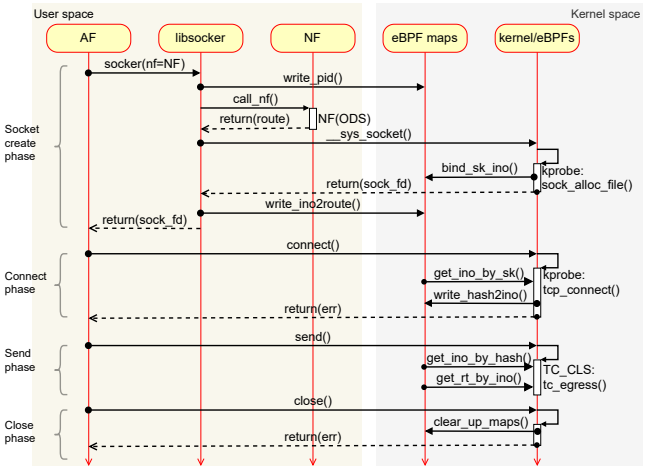

Figure 5: The workflow of Socker for TCP

program can read the file link on the Linux */proc* file system and get the inode number of the corresponding kernel socket inode.

Socker eBPF programs. Socker contains a set of eBPF programs attached to system calls to keep track of the sockets and TCP/UDP connections created by applications, in addition, we attach an eBPF program to TC egress to perform the packet control.

- **kprobe:sock_alloc_file.** Upon a socket create system call is invoked, the kernel will construct a kernel socket structure associated with the file structure. We attach a kprobe eBPF function to the sock_alloc_file to obtain the mapping from kernel socket structure and its inode number and store the mapping to the sk2ino map. When the socket(2) system call returns back to user program, the *libsocker* queries the inode number from the file system directory (*/proc/pid/fs/fd*) to get the kernel socket inode number, then executes the NF to get the desired route and stores the mapping from inode to route to ino2route map.
- **kprobe:tcp_connect.** We attach a kprobe eBPF program to tcp_connect function in kernel. We create the hook here because upon the tcp_connect function is invoked, the local port of the socket is already determined but TCP SYN packets is not emitted. That makes us able to control the SYN packets. We compute the hash of the 5-tuple, and lookup sk2ino to get the inode number of the current socket, then store the mapping to hash2ino map.
- **SCHED_CLS:tc_egress.** We attach eBPF program to the egress of TC, the program computes the hash of 5-tuple from the *__sk_buff*, and lookup the hash2ino to get the kernel socket inode number, then lookup the ino2route map to get the desired route for the current packet. Before emitting to the network, the packet will be processed properly to fit the network infrastructure (e.g., encapsulate the packet with MPLS stack for MPLS networks).

3.3 Socker Workflow

To illustrate how the design of Socker realizes the joint control of the application and the network, we use the TCP connection

Table 1: The eBPF maps in Socker

Name	Key	Value	Description
pids	u32	u32	Stores the set of user space process ids for eBPF codes to filter the sockets that needs to be controlled. It also helps to improve the performance by filtering processes since the kprobe in kernel tracing system intercepts system calls from all user space processes.
sk2ino	struct sock*	u64	Contains the mapping from the pointer of kernel socket structure to the inode number of the kernel socket file descriptor.
hash2ino	u32	u64	Stores the mapping from the hash of 5-tuple to the socket inode number.
ino2route	u64	struct route	Stores the mapping from kernel socket inode number to the computation result of NF.

workflow in Socker as a case to describe the process as shown in Figure 5, where we divide the lifetime of a TCP socket into four phases. For UDP sockets that does not invoke the *connect()* system call, we perform a similar procedure of TCP *connect()* system call in the *send()* system call of the UDP socket.

Socket create phase. The application calls the *socker()* function from *libsocker* to create a socket along with a network function pointer as one of the parameters. The *libsocker* writes the current process id to the *pids* eBPF map, and invokes the NF to get the result. Meanwhile, *libsocker* invokes the socket system call, we trace the execution of the kernel function *sock_alloc_file*, and store the mapping from kernel socket structure to the inode number representing the socket file descriptor. After the socket system call returns, *libsocker* queries the link information from the user space socket file descriptor to get the inode number of the kernel socket. Finally, *libsocker* stores the mapping from the inode number to the result of NF into the *ino2route* eBPF map and returns to the application.

Connect phase. When the application emits a connect system call, we track the kernel function *tcp_connect()* before the SYN packets are sent. The eBPF program looks up the *sk2ino* eBPF map to get the kernel socket inode number, and computes the hash of 5-tuples, then stores the mapping from the hash to the inode number into the *hash2ino* eBPF map. Note that for UDP sockets, the above process is moved to the send phase.

Send phase. After TCP connection established, the application sends packets to socket via socket writing system calls (e.g., *send*/*sendto()*). the eBPF program attached to the egress of traffic control (TC) gets the socket buffer before emitting the packet to network interfaces. Upon a packet reaches to the TC egress, the eBPF program compute the hash of 5-tuple from the socket buffer, and looks up the *hash2ino* map to get the kernel socket inode number, then looks up *ino2route* to get the desired route for current packet. Finally, rewrite the packet (e.g., encapsulate with MPLS) before emitting it to the network interface.

Close phase. When the socket is explicitly closed by the application or closed by errors, Socker cleans up all related entries in the eBPF maps.

4 IMPLEMENTATION

We implemented a prototype Socker based on BCC [3](v0.20.0). For the *libsocker*, we implemented a Python socket wrapper library. The NAB is realized by a python class that follows the observable design pattern, we expose the subscribe API by python function annotation, which will trigger the wrapped function when the subscribed data changed, and the data is passed to the wrapped function as the parameters (see Figure 2 as an example). All the code is public already.

5 EVALUATION

In this section, we give a preliminary evaluation of Socker on a simulated network, our goal is to show the benefit when using Socker to do socket programming against the traditional socket programming.

5.1 Settings

The evaluation is conducted on a Ubuntu20.04.1 desktop (Linux kernel 5.8.0 with eBPF kernel modules enabled) with an Intel i7 CPU (2.2GHz) and 16GB memory. We built a network consists three switches and two hosts as shown in Figure 6 using mininet. The network is configured to support MPLS routing and the bandwidth allocated for each link is marked on the figure. The server runs a simple TCP server for data receiving, and the client runs the program using Socker in Figure 2. The evaluation is based on a static network view, the dynamic states rely on a network monitor (e.g., at controller). The timely information from the monitor can be pushed to the NFs of end hosts, then the states propagate. We use the *enwik8* (100MB text file) dataset from the Large Text Compression Benchmark [7] as an input of the program.

We wrote another normal socket program to do the same task as in Figure 2, we run the two programs with different compress levels to get the overall performance.

5.2 Results

The results are shown by Figure 7. We can see that under different compress levels, Socker always outperforms the norm socket program, and achieved over 28.5% upload time reduction on average.

The reason for our gain is that Socker allows the application to choose the better routes for typical flows, in our case, the large file upload flow uses a route (i.e., maxBWPath) with higher bandwidth (slightly longer latency) compared to the default route (i.e., shortestPath). Also, the application can adjust its compress level according to the bandwidth of the route and the QoS requirements which is the task of *computeCL* function in Figure 2, the design of the *computeCL* function is out of our scope, however, we evaluate all the possible compress level (from 1-9) of Gzip and the result shows that Socker can gain in all cases.

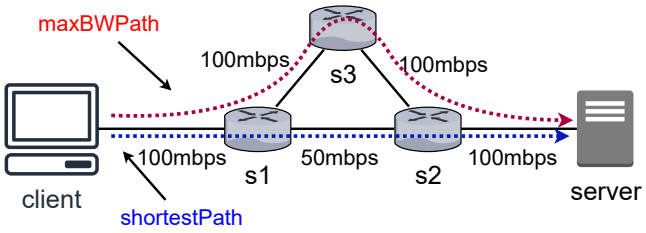

Figure 6: The topology for evaluation.

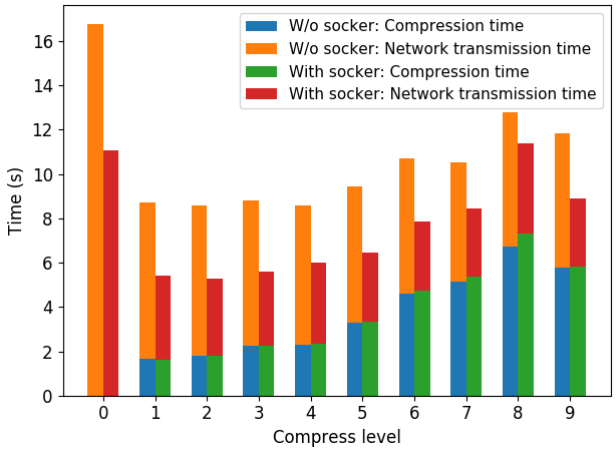

Figure 7: The compression and network transmission time under different compress levels.

6 RELATED WORK

Application-assisted network control: There are many researches that enable network control to function with information of applications. Socket Intents [9] augments the socket interface to express application's demands. Google's SDN WAN [4] and edge network [11] use local controllers to collect application states, and hand them over to the central controller to decide traffic allocation. For specific types of applications like video, there are also studies [1, 8, 13] to combine application-level metrics such as QoE to guide transport-level control.

Network-instructed application: Applications could also improve performance by combining network status. DASH-based algorithms [10] adjust bit rate based on estimation of network bandwidth, which is computed from real-time network states. In areas of machine learning and cloud gaming [12], there are also applications that choose between different algorithms based on network latency and bandwidth. One drawback for these applications is that applications could only indirectly "guess" network control from data plane statistics. This may lead to inaccurate and untimely inference when the network state is unstable.

In-network computation: Another interesting and related topic is to run applications in the network. As programmability of in-network nodes increases, applications could offload part of their logic to the network to do some in-flight computation. This way, the average round trip latency and computation at the servers could both be largely reduced. Current studies [5, 6] take advantage of programmable switches, and mostly implement simple and accumulative operations and fast key-value stores on the switches.

Socker's ultimate goal is to provide a unified framework for all the AF and NF,and implement them across the whole distributed system including network nodes. Socker will be able to provide a high level programming model for in-network applications to interact with network control.

7 CONCLUSION

In this paper, we present Socker, a first-ever socket level network-application integration framework. We introduce the system design of Socker in detail, and go over the process of TCP connection in Socker. Finally, we conduct a preliminary evaluation of our prototype implementation to show the effect of Socker.

REFERENCES

[1] Sadjad Fouladi, John Emmons, Emre Orbay, Catherine Wu, Riad S Wahby, and Keith Winstein. 2018. Salsify: Low-latency network video through tighter integration between a video codec and a transport protocol. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*. 267–282.

[2] Kai Gao, Taishi Nojima, and Y Richard Yang. 2018. Trident: toward a unified sdn programming framework with automatic updates. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. 386–401.

[3] iovisor. 2021. BCC: Tools for BPF-based Linux IO analysis, networking, monitoring, and more. https://github.com/iovisor/bcc. (2021).

[4] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, et al. 2013. B4: Experience with a globally-deployed software defined WAN. *ACM SIGCOMM Computer Communication Review* 43, 4 (2013), 3–14.

[5] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. 2018. Netchain: Scale-free sub-rtt coordination. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*. 35–49.

[6] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 121–136.

[7] Matt Mahoney. 2021. Large Text Compression Benchmark. http://mattmahoney.net/dc/text.html. (2021).

[8] Vikram Nathan, Vibhaalakshmi Sivaraman, Ravichandra Addanki, Mehrdad Khani, Prateesh Goyal, and Mohammad Alizadeh. 2019. End-to-end transport for video QoE fairness. In *Proceedings of the ACM Special Interest Group on Data Communication*. 408–423.

[9] Philipp S Schmidt, Theresa Enghardt, Ramin Khalili, and Anja Feldmann. 2013. Socket intents: Leveraging application awareness for multi-access connectivity. In *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*. 295–300.

[10] Iraj Sodagar. 2011. The mpeg-dash standard for multimedia streaming over the internet. *IEEE multimedia* 18, 4 (2011), 62–67.

[11] Kok-Kiong Yap, Murtaza Motiwala, Jeremy Rahe, Steve Padgett, Matthew Holliman, Gary Baldus, Marcus Hines, Taeeun Kim, Ashok Narayanan, Ankur Jain, et al. 2017. Taking the edge off with espresso: Scale, reliability and programmability for global internet peering. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. 432–445.

[12] Yunfei Zhang, Gang Li, Chunshan Xiong, Yixue Lei, Wei Huang, Yunbo Han, Anwar Walid, Y Richard Yang, and Zhi-Li Zhang. 2020. MoWIE: Toward Systematic, Adaptive Network Information Exposure as an Enabling Technique for Cloud-Based Applications over 5G and Beyond. In *Proceedings of the Workshop on Network Application Integration/CoDesign*. 20–27.

[13] Anfu Zhou, Huanhuan Zhang, Guangyuan Su, Leilei Wu, Ruoxuan Ma, Zhen Meng, Xinyu Zhang, Xiufeng Xie, Huadong Ma, and Xiaojiang Chen. 2019. Learning to coordinate video codec with transport protocol for mobile video telephony. In *The 25th Annual International Conference on Mobile Computing and Networking*. 1–16.