DETER: Deterministic TCP Replay for Performance Diagnosis

Yuliang Li
Harvard University

Rui Miao
Alibaba Group

Mohammad Alizadeh
Massachusetts Institute of Technology

Minlan Yu
Harvard University

Abstract

TCP performance problems are notoriously difficult to diagnose because subtle differences in TCP parameters and features may lead to completely different performance. The gold standard for diagnosis is to collect packet traces and trace TCP executions. However, it is not easy to use these tools in large-scale data centers where many TCP connections interact with each other. In this paper, we introduce DETER, a deterministic TCP replay tool, which runs lightweight recording all the time at all the hosts and then replays selected collections where operators can collect packet traces and trace TCP executions for diagnosis. The key challenge for deterministic TCP replay is the butterfly effect—a small timing variation causes a chain reaction between TCP and the network that drives the system to a completely different state in the replay. To eliminate the butterfly effect, we propose to replay individual TCP connection separately and capture all the interactions between a connection with the applications and the network. We show that DETER has low recording overhead and can help diagnose many TCP performance problems such as long latency related to receive buffer shrinking, zero windows, late fast retransmission, frequent retransmission timeout, and problems related to the switch shared buffer.

1 Introduction

Modern data center applications increasingly rely on high throughput and low latency TCP performance. Yet, these applications often experience TCP performance problems that are hard to diagnose. This is because the TCP stack is a complex system that involves many heuristics to deal with network conditions and application behaviors, and it has many variations that optimize for different traffic scenarios and application objectives.

As a result, there is simply no single best setting for all scenarios. Researchers invent more than two TCP variations every year and there are already tens of congestion control algorithms to choose in Linux. TCP in Linux 4.4 has 63 parameters to configure, some of which are less known to normal application developers, such as early retransmission flag and TCP low latency flag which provides options for optimizing specific traffic settings. Other parameters are hard to configure even for TCP experts, as they have to run TCP multiple times to fully understand the influences of different parameter settings and the interactions of various TCP features. For example, thin-dupACK dynamically changes the threshold of the number of dupACK for fast retransmission based on the size of the current transfer. TSO window divisor affects the Nagle test for TSO, which decides how many packets to wait in order to form a larger packet.

Moreover, TCP is under continuous, error-prone development. There are 16 bugs identified in Linux TCP [25] in just July and August of 2018. As an example, one bug is related to DCTCP, where the DCTCP CC's ACK generation conflicts with the basic TCP framework's ACK generation, resulting in some packets never being acknowledged [19].

Many misconfigurations and bugs are hard to diagnose because they are sporadic and intermittent. However, they are still sufficient to degrade application performance, especially in data centers where large scale distributed systems often involve thousands of requests to fulfill a task [48, 39], because a single long latency may delay the entire task [32, 42].

Although diagnosing TCP performance problems is notoriously hard, the gold standard tools are still the same as what have been used for tens of years: capturing packet traces [18] and tracing TCP executions [13, 1]. While these tools are useful for diagnosing individual connections, using them in large-scale data center environments is hard, because there are millions of flows from hundreds of thousands of hosts interfering with each other constantly. Collecting packets and tracing TCP executions at all hosts and switches takes large quantities of storage, computing, and bandwidth resources. TCP counters [58, 28] are useful lightweight tools in production, but they are not detailed enough to diagnose the complex settings and interactions mentioned above (see more examples of complex TCP performance problems in §5).

A common way to debug complex large-scale systems is deterministic replay [52, 40, 33, 49, 27, 37]. Deterministic replay is proven to be an effective tool for developers to recreate performance problems, identify their root causes, and uncover many long-standing bugs in popular software. It would be ideal if we can deterministically replay TCP (i.e., deterministically re-execute the TCP code).

However, deterministically replaying a large network of TCP connections is difficult because TCP is a tightly coupled system with multiple interacting parties: applications, the network, other TCP connections traversing through a common switch, and the kernel at hosts.

In particular, the closed-loop nature of TCP creates a *butterfly effect*, where even small timing variations (e.g., clock drifts) between the runtime and the replay can drive the system to an entirely different state. Better time synchronization cannot solve this problem: even a nanosecond of timing variation leads to completely different TCP behaviors (§2.2). This is because small timing variations at hosts can cause different packet arriving orders at switches and therefore different packet drops. The differences in packet drops cause different TCP behaviors (e.g., congestion control) in turn, leading to different traffic rates from TCP senders and causing more differences in switch behaviors such as packet drops. Such butterfly effect propagates to many flows in the entire network after many rounds.

To eliminate the butterfly effect, we propose DETER, a DEterministic TCP Replay system, which breaks the closed loop interactions by replaying each TCP connection separately. We identify the minimal set of signals that capture all the interactions between a TCP connection with the application and the network, and record these signals at hosts in a lightweight manner. Specifically, DETER captures application socket calls and any impact on packets (e.g., if they are dropped or marked ECN) in the runtime. In the replay, we no longer need switches because all their actions to packets have been recorded and can be simply replayed. Since all the switch actions are deterministically replayed, we break the butterfly effect. We also isolate the TCP connection with other connections in the network because they only interact through switch actions.

The next question is how to deterministically replay an individual TCP connection. Although we already capture the interactions with the application and the network, there are still non-determinisms in the kernel at hosts. We design a customized solution for TCP which captures TCP-kernel interactions such as the kernel calling TCP handler functions, TCP reading kernel variables, and locks in thread scheduling. Note that we do not need to capture every packet, as the sender and receiver can generate packets and ACK for each other. The size of our total recording is just 2.1~3.1% of the size of fully compressed packet traces.

Since the recording is lightweight, DETER can run at all times for every connection on each host. Upon observing a performance problem, we can use DETER to zoom into any TCP connection, deterministically replay its exact same execution, capture packet traces, and examine TCP state during the execution—all after the fact. We can also iteratively debug the same performance problem instance multiple times to collect different levels of detail each time.

Once we have the packet traces for each connection using DETER, we can also replay network queues in a physical network, emulator, or simulator as long as the setup has the same topology, routing, buffer size, and switch queuing algorithms as the runtime. During the replay, we push all the outgoing packets for all the senders into the network based on their timestamps. We also introduce a heuristic that significantly improves the accuracy of replaying packet drops.

We demonstrate the benefits of DETER by showing how we diagnose TCP performance problems in a Spark application with 6.2K connections, tail latency problems in an empirical web search workload with incasts, and example performance problems in a local testbed. With DETER, we can also diagnose a wide range of performance problems that require tracing the TCP execution, such as long latency related to receive buffer shrinking, zero windows, late fast retransmission, frequent retransmission timeout, and problems related to the switch shared buffer. The main limitation of DETER is that it requires recording at both the sender and the receiver of a connection and therefore cannot work when we do not have access to both ends.

2 Diagnosis Example and Challenges

We use a diagnosis example to demonstrate the benefits of deterministic replay. We then use the example to show the key challenge to enable the deterministic replay—the butterfly effect. Even a nanosecond of sending timing variation leads to completely different TCP behaviors.

2.1 A Diagnosis Example

We use an example to show how DETER helps diagnose TCP performance problems. We run a network with two senders (A and B) and one receiver, which are connected to a single switch and 10 Gbps links between them. Each sender sends two long flows of 20 MB each. 30 ms after the long flow starts, sender A sends a short flow of 30 KB to the same receiver. In one run, the short flow takes 49 ms to complete, which is two orders of magnitude higher than its expected completion time. In comparison, the RTO is just 16 ms.

Usually, people diagnose a problem by reproducing it. However, this problem is very hard to reproduce (shown in §2.2). If we cannot reproduce a problem, we have to rely on the information captured online, such as the TCP counters that data centers usually continuously monitor [58, 28]. Unfortunately, TCP counters are not enough for diagnosing this problem. The counter for retransmission timeout is two, but twice the RTO (2*16 ms) is still much less than 49 ms.

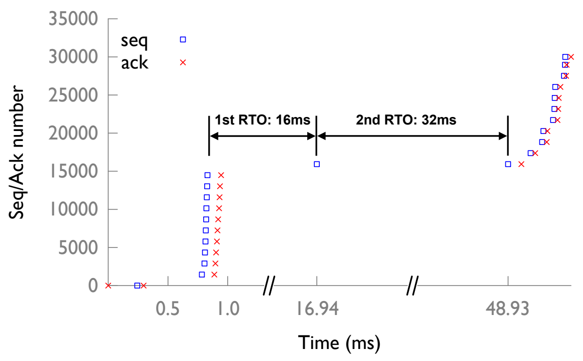

Figure 1: Receiver side Seq and Ack number of the short flow that experiences 49 ms FCT.

With DETER, we can deterministically replay the connections using the lightweight data recorded in the runtime (Table 1). During the replay, we capture the packet trace at the receiver side for the short flow (Figure 1). The trace shows that the second timeout is 32 ms. This is because the two timeouts are consecutive and thus trigger exponential backoff. The trace also shows the reason why the sender experiences the second timeout: the receiver receives the first retransmitted packet at 16.94 ms, but it does not send an ACK. Without the ACK, the sender has to retransmit again at 48.93 ms.

Why does the receiver not send an ACK for the first retransmitted packet? DETER allows us to replay multiple times, in order to collect more data and iteratively diagnose the problem. We replay again and use Ftrace [1] to get the function call graph on the processing of the first retransmitted packet. It shows that TCP enters the delayed ACK function, which means TCP decides to delay the ACK for the first retransmitted packet. The delayed ACK timeout is 40ms (which is a hardcoded value in the kernel and not configurable), which is longer than 2*RTO, so the second retransmission triggers first.

The root cause of this problem is that delayed ACK is very risky in the presence of RTO, because after RTO the sender can only send one packet. Ideally, the receiver needs a way to identify retransmissions (e.g., the sender marks the retransmitted packets), so it does not delay the ACK for them. As a workaround today, reducing the delayed ACK timeout can mitigate the problem.

2.2 Butterfly Effect

While deterministic TCP replay is a powerful tool for diagnosing TCP performance problems, it is not easy to ensure determinism. For the above example, if we simply replay with the same socket calls at the same times as the runtime, we cannot reproduce the problem.[1] Figure 2 shows that when we replay 100 times, the short flow always has way less than 49 ms flow completion times (FCT). In the production

[1]We synchronize the clocks among the senders and receivers to 100s of nanoseconds precision by PTP (Precision Time Protocol [2]).

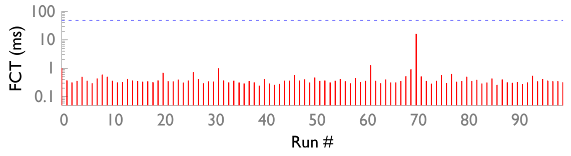

Figure 2: FCT of the short flow across 100 attempts of replay with socket calls. The blue dashed-line is 49 ms.

where there are more flows and more dynamic traffic than our testbed, it is more difficult to reproduce the same problem.

The key challenge for the deterministic replay is the butterfly effect. Packet sending times at hosts often have microsecond-level variation between the replay and the runtime. This is caused by the inherent host non-determinisms, such as the clock drift, context switching, kernel scheduling, and cache state [42].

The small variation gets amplified by the butterfly effect—the closed loop interactions between switches and TCP. A small packet sending time variation may change the order of packets from different hosts at a switch, which causes *switch action variations*—the switch may drop or mark ECN on a different set of packets. This starts the butterfly effect in the closed loop between switches and TCP: *Switch action variations* cause *TCP behavior variations* (e.g., TCP changing congestion window size differently). TCP behavior variations change its flow sending rates, which affect the queue lengths at all the switches the flow traverses ever since and lead to more switch action variations. Such a chain reaction between switches and TCP affects more and more flows all over the network in multiple rounds.

One may expect that reducing the sending time variation (e.g., better clock synchronization, more deterministic packet processing time) can improve the replay accuracy. However, our experiment shows that even a nanosecond of variation can lead to completely different packet-level behaviors.

We run an ns3 simulation [15] to control the sending time variation. We use the same topology and traffic as in §2.1. For the runtime, we set the host packet processing delay to 10 us, the same as what we measure in the testbed. The short flow incurs a long flow completion time because of the correlated RTO and delayed ACK. We then replay the experiment with the same socket calls and timings. To simulate different levels of sending time variation, we simulate a normal distribution of host packet processing delay with the same mean delay of 10 us but with a standard deviation ranging from 0 to 1000 ns. For each level, we replay 100 times.

Figure 3 shows the percentage of replays that reproduce the correlated RTO and ACK delay on the short flow. Once the sending time variation exceeds zero, even just 1 ns, the probability of reproducing the same problem suddenly drops.

This is because with a non-zero sending time variation, there is always a chance that a switch takes different actions on a packet between the runtime and the replay. Smaller timing variation can only delay the appearance of different ac-

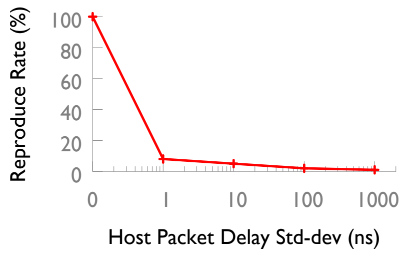

Figure 3: The rate of reproducing the correlated RTO and ACK delay.

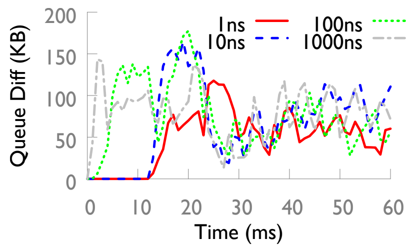

Figure 4: The time series of queue length difference.

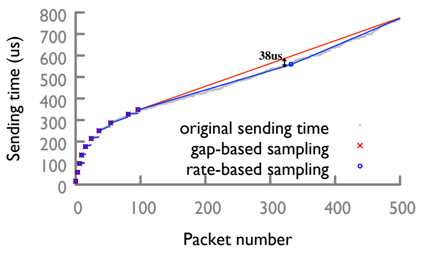

Figure 5: Inference error of sampling approaches. (The lines indicate the inferred sending time of each packet. The end points of the lines are sampled sending times.)

tions, but cannot prevent it. Once the switch takes a different action, the butterfly effect starts, causing a chain reaction of changing sending rates and queue lengths. The chain reaction persists regardless of the level of the sending time variation.

Figure 4 illustrates this. We show the time series of queue length difference between runtime and replay experienced by each packet. For each level of sending time variation, we show a typical one of the 100 replays[2]. For 1 ns variations, although the queue length difference starts later than with higher variations, once the difference starts at 12 ms, it never goes down to 0.

This result indicates that we cannot simply rely on reducing the sending time variation. This motivates our DETER design, which decouples the TCP and the network so that switch action variations cannot affect TCP.

3 DETER Design

In this section, we discuss DETER design with four key ideas: first, we break the butterfly effect by replaying individual TCP connections separately and record TCP's interactions with the application and the network. Second, to deterministically replay each TCP connection, we record all the non-determinisms that happen in the interactions between TCP and the kernel. Third, we introduce a rate-based sampling solution to reduce the overhead of recoding packet sending times. Finally, with the packet traces of all the connections, we show how to replay switch queuing behaviors.

3.1 Breaking the Butterfly Effect

We break the closed loop between TCP and the switches by replaying individual connections separately. We identify the minimal set of signals that capture all the interactions of a TCP connection with the application and the network.

TCP interacts with applications through socket calls. DETER captures all socket calls and its input arguments such as the number of read/recv bytes and socket flags.

[2]Although we cannot show all 100 replays here, we inspect each of them, and they have similar trend.

TCP interacts with the network through packets. TCP sends packets into the network and receives packets from the network. We do not need to record most incoming packets because we replay the sender and the receiver of a connection together and they can automatically generate packets for each other. We only need to record how the switches inside the network change the packet stream such as dropping packets or marking them with ECN bits. At the receiver, we detect packet drops by checking if the IP_ID fields are continuous and ECN by checking the ECN bits (see Section 4 for details) and record them there.

Note that for a TCP connection, it does not matter which switch drops or marks the packets. Only the final changes to the packets matter. So in the replay, we no longer need switches because their actions to packets have been recorded and we can just replay them. Since the switch actions are deterministically replayed, we break the butterfly effect.

A TCP connection interacts with other connections when they share switch resources in the network and cause different switch actions[3]. Since we recorded switch actions, we also isolate the interactions among TCP connections.

In summary, in the runtime, we record socket calls and switch changes to packets at all the hosts. Users can specify which connections to replay. To replay a connection, we set up a simple two-host testbed that runs as a sender and a receiver for every single connection without involving any switches. We run a socket call generator to generate socket calls at the right time and run a packet corrector to inject actions on packets before they arrive at the TCP sender and receiver. We can easily parallelize the replay of multiple connections because we replay each connection independently.

3.2 Handling Non-determinisms in the Kernel

The next question is how to deterministically replay a single TCP connection. It is complex to replay a general system [49, 27], which requires record and replay lots of non-determinisms. We use the knowledge of TCP to design a customized replay for TCP, which is lightweight. Specifically, besides the interaction with the application and the network,

[3]We discussion TCP connections on the same host in the next subsection.

TCP also has three non-determinisms from interacting with the kernel: the kernel may call TCP handler functions, the TCP may read kernel variables, and there is thread scheduling.
(1) TCP handler function calls from kernel: The kernel may call some TCP handler functions. For example, the OS timer may call TCP timeout handler. The kernel may also call resume transmission handler, which sends more packets in the send buffer. We need to record them.
(2) Reading kernel variables: TCP reads a few variables that are updated by other kernel programs (or hardware), such as memory pressure indicator, the jiffies (a low-resolution clock), the mstamp (a microsecond-resolution clock), and the send queue byte count. We should record the return value of each read.
(3) Thread scheduling: TCP works in a multi-threaded environment. Different threads, such as applications, NIC interrupts, and OS timers, access the shared socket variables by calling TCP handler functions. For example, an application thread calls a socket call handler to copy data into the socket send buffer; a NIC interrupt may call the TCP receive packet handler to frees up some space of the send buffer; OS timer may call the timeout handler to send a pending packet in the send buffer. It is important to ensure the order of different threads accessing the same variable. Fortunately, TCP uses a single socket lock to ensure that only one thread can access all the shared variables at a time. Thus, we just record the order of lock acquisition of different threads by giving a sequence number for each lock acquisition.

In the replay, we run the same TCP stack with the same TCP configuration as the runtime. In addition to the socket call generator and the packet corrector, we also generate handler calls from the kernel based on the recorded logs. We feed in the recorded kernel variables when TCP reads them. We also enforce the order of lock acquisition of different threads (see §4 for more details).

3.3 Sampling Packet Sending Times

So far we have ensured the ordering of TCP behaviors (e.g., the sequence of packets, state updates, loss detections, timeouts). One remaining question is how to replay packet sending times accurately. Recording the sending times for all the packets takes high storage overhead. To reduce the overhead, we choose to sample packets, record the sending times for sampled packets, and infer the times for the other packets. The question is how to select the samples in real-time while bounding the inference error within a given threshold *th*.
Strawman solution: gap-based sampling. TCP usually sends packets in bursts. So intuitively for each burst, we can keep the sending time of the first packet and the burst length. Assuming all the packets in the same burst follow the same sending rate, we can then infer the sending times of all the unsampled packets. We can identify packets in the same bursts if their interarrival time is below a threshold.

We perform a simple experiment to show that this approach has an unbounded error. We send two flows from two senders through a shared 10 Gbps link. The second flow starts 500us after the first flow. Figure 5 shows the packet sending time series of the first flow. All the packets from the 96-th to the 499-th are in the same burst (i.e., no gap of packet sending time), but the rate changes. As a result, the inferred sending time of the 323-th packet is 38 us later than the actual time.
Our solution: Rate-based sampling. Gap-based sampling fails to sample packets when the packet rate changes. Therefore, instead of recording the burst length, we propose to record the packet rate. When the inferred sending time based on the recorded packet rate is wrong (i.e., the difference with the actual time is above the threshold *th*), we sample a new packet. We set *th* to 5 us by default.

Specifically, in the runtime, we follow Algorithm 1. s is the previous sampled packet and p is the new packet. Given the sending time of s ($s.time$) and a packet rate r, we can infer the sending time of p ($p.time$). In reverse, to ensure that our inferred sending time of p falls in the range of $[p.time - th, p.time + th]$, we must ensure our recorded packet rate r falls in the range of $p\_range = [\frac{p.index - s.index}{p.time + th - s.time}, \frac{p.index - s.index}{p.time - th - s.time}]$ (Line 2). Thus, we compare the recorded rate range rec_range and p_range. If they overlap, it means we can find a rate, in the intersection of rec_range and p_range, that can be used to infer a bounded sending time for both p and all the previous packets between s and p. Thus, we do not need to sample p (Line 4). Otherwise, if the two ranges do not overlap, we sample p, record a rate in rec_range, and reset rec_range (Line 6-7).

DETER can generate the full packet trace for each connection, by combining the recorded (inferred) sending times with the packets generated by the replay of TCP execution.

Algorithm 1 DETER Sampling sending time. *p.index* is its index within its 5-tuple flow, and *p.time* is its sending time.

```
1: procedure SAMPLE($p$: a new packet)
2:     $p\_range = [\frac{p.index - s.index}{p.time + th - s.time}, \frac{p.index - s.index}{p.time - th - s.time}]$
3:     if $p\_range \cap rec\_range \neq \emptyset$ then
4:         $rec\_range = p\_range \cap rec\_range$
5:     else
6:         record($s.index$, $s.time$, $rec\_range.mid$)
7:         $s = p$; $rec\_range = [-\infty, \infty]$
```

3.4 Replaying Switch Queues

Because we can get all the packets, their sizes, and sending times for each connection in the network (§3.2 and §3.3), we can use them to replay switch queues in simulators (e.g., ns3 [15]) by pushing all the packets at the right time into the network. Replaying switch queues can help us understand the interaction between different connections at the switches (e.g., which flows contend for the queues).

The simulator needs the same topology and switch data plane (e.g., forwarding tables, buffer sharing policies, switching delay, and link propagation delay) as the runtime. Today, many vendors build high-fidelity simulators for their own devices [5, 21, 11]. One can also choose to replay switch queues in a physical network if available. Replaying switch queues also requires that the hosts during the runtime have microsecond-level synchronization, so that the relative packet sending time error across hosts are small. Clock synchronization in data centers is moving towards sub-microseconds level [2, 22, 34].

Replaying the exact queueing behavior is both impractical and unnecessary. It requires recording the exact order of enqueue and dequeue, which is too heavy for the runtime. On the other hand, it is often good enough to show the contending flows and their occupancies with high accuracy.

Thus, we opt for a simple design that can achieve high accuracy. We simply push all the packets into the network at the right time. It can achieve high accuracy because the switch queue occupancy is a continuous function with respect to packet sending times. Since the difference in packet sending times between the runtime and the replay is bounded, the difference of switch queue occupancy is also bounded. Specifically, suppose a packet's arrival time at a port differs by k packets transmission time, and the fan-in of that port is f, the queue difference is at most $(f-1)k$. k is small because our sampling bounds the sending time error to 5 us, and there are limited hops to amplify it. f is also small because the destinations of flows traversing a switch are random[4]. Even if f is large, such as during incast, the queue occupancy is also large, so the difference is a small fraction of the queue.

However, one exception is packet drops. Because dropping packets or not is a binary decision (not a continuous function), even if a microsecond level difference can cause different drops. Specifically, a runtime dropped packet may get through, which we call a *false-accept*. It also occupies some free space in the queue, leaving less space for later packets that should be in the queue, so one of the later packets may get mistakenly dropped, which we call a *false-drop*.

We propose to reduce the probability of false-accepts and false-drops by letting the hosts tag *should-be-dropped* packets. In this way, we ensure that the switches only drop packets with tags (for eliminating false-accept) and always deliver packets without tags (for eliminating false-drop).

The key challenge is how to know which switch to drop the tagged packets. Since the switch queue occupancy is a continuous function, it has bounded differences with respect to the sending time difference. We propose to decide whether to drop packets at a switch based on the switch's queue occupancy upon packet arrivals. That is, when a *should-be-dropped* packet arrives at a switch, and the queue occupancy is above a threshold (e.g., > queue max length - 5 MTU), the switch drops the packet.

[4]In theory, the fan-in is within 4 for 99.7% of the time for a 64-port switch with random traffic.

Type	Data recorded
Interaction w/ network	losses, ECN, reordering
Interaction w/ applications	socket calls
Handler call from kernel	Timeout handler, resume transmission handler, packet receive handler
Kernel variables	Infrequently updated variables, e.g., jiffies, memory pressure indicator
	Influence of frequently updated variables e.g., RACK loss detection
Order of lock acquisitions	sequence number of lock acquisitions for diff. threads
Timestamp samples	Sampled packet sending time (time, index, rate)

Table 1: Runtime recorded data.

When a packet only experiences one congested switch on its path, which is the most common case, our solution works well. In the rare case when there are multiple congestion spots on the path, DETER may drop the packet at a wrong location. Our evaluation shows that this heuristic reduces the error of dropping packets[5] from 58.3% to 2.87%.

4 Implementation

In this section, we discuss the implementation details of DETER. We just need 139 extra lines of code in the Linux kernel. Then we accomplish the record and replay with two kernel modules and two userspace programs (3000 lines of C and C++ code in total).

Runtime recording. For each connection, we first record its configurations, and then record the data listed in Table 1 during its runtime. We note that the configurations of connections on the same server are mostly the same, so we only record the parameters that differ from the default values. Our current prototype starts the recording after the connection is successfully built[6]. We now discuss the runtime recording.

Interaction with the network: This includes packet drops, ECN, and packet reordering. In our design, we use the IP_ID field to detect packet drops: Linux sends packets of each connection with consecutive IP_ID values, so the receiver can check if there are gaps in the series of incoming packets to detect drops (Similarly, the sender can detect drops in the incoming ACKs)[7]. On other platforms that do not have the consecutive IP_ID feature, we use LossRadar[44] to detect drops, which only takes O(#loss) space. The host also checks the ECN of the IP header of each incoming packet, and record

[5]Percentage of false-accept, false-drop, and drop at wrong location in all drops.

[6]Record and replay for connection setup is not very different. The only difference is detecting the drop of the first packet (SYN and SYN-ACK). This can be solved by recording the IP_ID of all SYN packets at both sender and receiver, which just adds 8 bytes for each connection.

[7]This is different from TCP's drop detection: TCP sender does not distinguish drop of a data packet or its ACK. We must distinguish them because both the sender and the receiver must replay accurately.

1 bit (CE) for it. Sometimes there may be packet reordering, which we can detect also using the IP_ID field.

Recording the interactions with the network is lightweight. In data centers, the packet drop rate is just 10^{-5} to 10^{-4} [51, 36]. For ECN we just need 1 bit per packet. Reordering is rare, so it does not cost much. We instrument the TCP receive packet handler to record them.

Socket calls from the application: We hook the TCP socket call handler functions to record the #bytes and flag, so that we do not need to change the application.

We can reduce the storage overhead of socket calls a lot. We find that there are often identical socket calls. For example, distributed files systems break large files into fixed-size chunks, so most of the send and receive sizes are the same. Thus, we store all the common patterns of socket calls (the common #bytes and flag pairs) for different applications, and only record the pattern numbers in the runtime. DETER associates connections to applications via their TCP port numbers.

Other TCP handler calls from the kernel: We hook the timeout handlers and the resume transmission handler, and record them when they get called.

Kernel variables read by TCP: We record the memory pressure indicator and jiffies with low overhead because their values change infrequently. The memory pressure indicator is very rarely set, and the jiffies increments by 1 every 4 ms. So we just maintain the values of the last read and only record the reads that return a new value.

The mstamp and the send queue byte count are updated frequently. We reduce the overhead by recording their *influences* instead of their values. Specifically, the variables influence the TCP executions by serving as the metrics of if-conditions in TCP. For example, TCP uses the mstamp to detect losses (RACK [16]). We just need to record the loss detection result, rather than the actual value of the clock. We identify and record all the if-conditions they affect (1 bit for each), which relates to loss detection, cwnd reset, TCP segmentation offload, and TCP small queue. Moreover, most of the if-conditions have a dominant result (e.g., loss detection mostly return false), so we reduce the overhead further by only recording when they have the uncommon result.

We use a special reader function to record these values. For example, in the TCP code, we replace `a=jiffies` with `a=reader(jiffies)` to record the value of jiffies and replace `if (mstamp>b)` with `if (reader(mstamp>b))` to record the influence of mstamp. The reader function simply records the value passed to it and returns this value.

Lock acquisition: We instrument TCP's lock acquisition function to record which thread calls this function, so we know the order of lock acquisition by different threads. We also optimize the overhead. Specifically, one thread may acquire the lock many times consecutively. For example, NIC interrupt acquires one lock for each incoming packet, so there are often tens of lock acquisition by NIC interrupt in a row. Therefore, we record the number of consecutive lock acquisitions, instead of recording them individually.

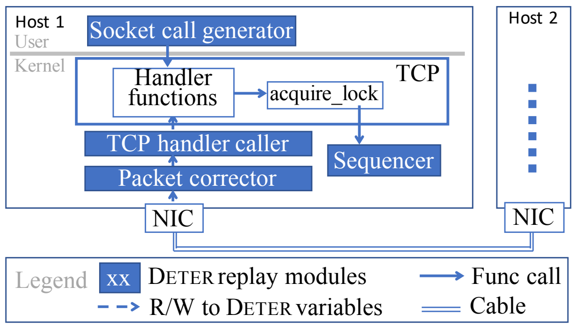

Figure 6: Replay implementation in DETER

Sampled sending times: To get the most accurate timestamps, we sample and record the sending times in the NIC driver, just before TCP pushes packets into the NIC ring buffer.

Replay. We now discuss the replay.

Replay TCP stacks. Figure 6 shows the replay setup. We implement the *packet corrector* with NetFilter [7]. It injects drops and CE bits to the incoming packets[8]. We also enforce the reordering here.

To replay the socket calls, we implement a *socket call generator* in the user space to inject socket calls from the applications according to the log.

We also implement a *TCP handler caller*, which is a kernel thread that calls TCP handler functions according to the log. The handler functions include the packet receive handler, the timeout handler, and the resume transmission handler. When calling the packet receive handler, it gets a packet from the packet corrector as an argument to the handler.

To enforce the order of different threads acquiring the lock, we implement a *sequencer*. It knows the order of different threads acquiring the lock based on the log. We instrument the lock acquisition function to check with the sequencer before it actually acquires the lock. If the current thread is not the next to acquire the lock, it waits for other threads until itself is the next to acquire the lock.

We reuse the reader function that we introduced before to feed the recorded kernel variables or their influences. During the replay, the reader function reads the log and return the corresponding value.

Replay sending and receiving timestamps. We only record packet sending times for replay. We then infer receiving times from sending times: for the received packet which triggers a new packet to send, we can estimate its receiving time as the sending time of the new packet minus the average packet processing time, which is measured separately. For the received packet that does not trigger a new packet, its gap with the previous received packet is close to their sending time gap,

[8]We require no packet drops before packets entering the packet corrector, so we must make sure no packets get unexpectedly dropped in the queues on the hosts (e.g., NIC ring buffer, softirq queue, qdisc) during the replay. We can set the sizes of these queues large enough to avoid unexpected drops.

because they experience similar network conditions. Note that only the sending times affect the switch queue replay, but not the receiving times.

Switch queue replay. We run Precision Time Protocol [2] in our testbed. We implement the switch queue replay in both testbed and simulation. For the testbed, we implement a DPDK packet generator that reads the packet trace, tags packets, and sends packets to the NIC at the right time. We use a NetFPGA-based switch to implement the drop accuracy improvement (§3.4). It is also implementable in P4 [23]. We also implement the replay in a packet-level simulation in ns3 [15], with the same topology, link delay and bandwidth, switch queueing algorithm, and routing state as the testbed.

5 Evaluation

In this section, we demonstrate the benefits of deterministic replay in DETER by showing how we diagnose TCP performance problems in a Spark application with 6.2K connections, the tail latency problem in an empirical web search workload with incasts, and example problems in a local testbed.

We also measure the CPU and storage overhead of DETER recording and the accuracy of DETER replay. Our evaluation shows that DETER only uses 2.1~3.1% compared to fully compressed packet traces and requires 0.094%-1.49% of CPU overhead. DETER also fully replays the sequences of packets at hosts and replays switch queues with lower than 1 MTU differences on average.

5.1 Diagnosis in Spark

Evaluation setting. We run a TeraSort job in Spark [24] that sorts 200 GB data on 20 servers connected with 10Gbps network in Amazon EC2 [20]. We use 4 executors (i.e., 4 cores) and 20GB memory on each server. The NIC MTU is 1500B. We enable TCP segmentation offload, and disable generic receive offload[9]. We run DETER on all servers to record data for all connections during the runtime and also run Tcpdump [18] to collect the packet traces as the groundtruth.

Replay accuracy. We use DETER to replay each connection and run Tcpdump during the replay. We compare the packet traces we collected during runtime and replay. The sequence of packets are exactly the same (we have a one-to-one mapping of TCP headers). The sending time differences between packets are lower than 5 us.

Diagnosis. We can use DETER to identify and diagnose tail latency problem in Spark. We define each flow as all the packets belonging to the same Spark message. Spark usually sends one large message with multiple socket calls. So if a socket call starts after all the previous packets are acknowledged, we treat the socket call as a new message. Otherwise, we treat it as part of the previous message.

[9]We have not implemented replay for it, but it is not hard (§7).

We find that the tail latency of flows from HDFS are mostly caused by receiver limit, because their receive windows frequently reach zero.

The 99.9 percentile latency for flows between Spark workers experience a variety of problems as summarized in Table 2. For flows shorter than 1MB, their tail latency are mostly caused by packet drops (RTO or fast retransmission (FR)). For flows longer than 10MB, their tail latency are mostly caused by receive window frequently reaching zero (Rwnd=0).

The flows in the range [100KB,1MB] are of particular interests, because most of their tail latencies (18 out of 24) are caused by multiple delayed ACKs. We show the sender side packet trace for one of them in Figure 7; others have similar patterns. The sender frequently gets blocked after sending a burst of packets, until around 40 ms later when the ACK comes back. Such burst-40ms-ACK pattern repeats multiple times and causes excessive delay. This is out of our expectation, because the receiver should acknowledge every two data packets.

So we use DETER to replay again, and use TCP Probe to print the variables that decide whether to delay the ACK. We find that TCP explicitly delays the ACK because the free space in the receive buffer is shrinking. This suggests that the root cause is the application not reading the data in the receive buffer in time. So we replay again and confirms that the receiver application is slow in issuing receive socket calls. Our guess is that the application is busy with processing data, so the CPU is the bottleneck in this case.

DETER helps us to effectively diagnose the problems caused by the network (e.g., RTO, fast retransmission). In addition, it also helps us identify problems caused by applications. This is helpful because in data centers it is often unclear where the performance bottleneck is, and blaming the network is often the first reaction [28]. Unlike previous systems that infer the bottleneck [58, 28], DETER helps us quantify the duration of different bottlenecks without instrumenting the applications.

Overhead. DETER records a total of 200.6 MB data in the runtime. For comparison, Tcpdump uses 22.4 GB to record only the IP and TCP headers and timestamps and 6.5 GB after applying the state-of-the-art compression solution [38]. DETER storage is only 3.1% of compressed packet traces.

If we keep using DETER to monitor a data center that continuously runs such Spark jobs, DETER storage overhead translates to 2.8 GB/host/day. We can delete the data every day if we do not see performance problems.

We also use Linux perf [8] to evaluate the CPU overhead of DETER recording. DETER uses 0.094% of total CPU time.

5.2 Diagnosis in Data Center Workload

Evaluation setting. We now generate TCP tail latency problems using empirical workloads modeled after traffic patterns that have been observed in production datacenters. We run

Flow size (MB)	<0.1	[0.1, 1]	[1, 10]	>10
RTO	8	3	4	0
FR	74	0	0	0
Delayed ACK	0	0	18	0
Rwnd=0	0	0	1	1
Slow start	0	0	1	0

Table 2: Reasons for 99.9-th percentile latency for flows of different sizes in Spark.

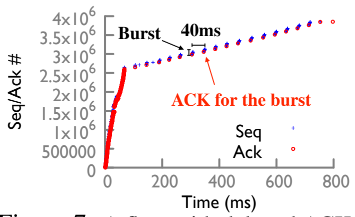

Figure 7: A flow with delayed ACKs.

Flow size (MB)	<0.1	[0.1,1]	[1,10]	>10
Congestion	149	35	25	2
Late FR	29	27	0	0
ACK drops	0	2	0	0
Tail drops	4	1	0	0
RTO	2	1	2	0

Table 3: Reasons for 99.9-th percentile latency for flows of different sizes in data center workload.

a client-server RPC call software [6] in the same 20-node Amazon EC2 testbed. The clients set up a persistent TCP connection to each server, and request flows according to Poisson process from a random server. We set the flow sizes following the distribution observed in a production data center running web search applications [26]. We also add incast traffic pattern, by having the client simultaneously request 10 random servers, so the 10 servers respond synchronously causing incast. We set the average request rate to have an 80% network load, and 20% of the load is incast traffic. We generate a total of 280K requests over 380 persistent connections. All 20 nodes run both client and server.

Similar to the Spark program, we use Tcpdump to collect traces at both the runtime and the replay and show that DETER can provide deterministic replay for all the connections.

Diagnosing tail latency. In Table 3, we classifies the root causes into five categories: congestion (i.e., low throughput), the fast retransmission happens very late (late FR), ACK drops (so the sender gets stuck), tail drops (so the packets at the end of a flow get dropped), and RTO.

We analyze the short flows (100KB-1MB) with latency above 99.9-th percentile as an example. At the 99.9-th percentile, flows experience 173.8 slow down of completion time compared to the case of running the flow alone. We make the following interesting observations:

RTO is not the main root cause of tail latency. A widely discussed reason for tail latency is RTO [29, 54]. But actually RTO is rare in this experiment. The reason is that when there are multiple requests in the same connection, later requests can help recover the packet losses of previous requests, so TCP loss recovery is effective in this scenario.

Fast retransmission (FR) is delayed for 10s of milliseconds. When these flows experience loss, the senders start FR after 10s of duplicate ACKs (dupACKs). This is unexpected because the normal threshold for FR is 3 dupACKs. And this is bad because short flows usually do not have so many dupACKs. In fact, most (22 out of 27) of these flows do not have enough dupACKs on their own; their FR starts 10s of milliseconds later when another request in the same connection starts and triggers more dupACKs.

With DETER, we can replay repeatedly and gain more insight into the problem. To understand why it requires so many dupACKs for FR, we replay the connection of the flow that experiences late FR with the highest slow down. We use TCP Probe [13] to print out the threshold for dupACKs (tp->reordering) on every ACK's arrival during the replay. We find that this threshold starts at 3, but later increases (and never decreases), so when the flow that experience late FR arrive, the threshold is 45. We search in the TCP code, and find the threshold only increases when TCP detects reordering. So we replay again and print out the ACKs when the threshold increases, and find that they do reflect reordering.

A quick fix is to set the upper bound of this threshold (net.ipv4.tcp_max_reordering) lower, but it risks spurious retransmission in the presence of reordering. A potential optimization to TCP may be regularly reducing the threshold.

Overhead DETER records a total of 103.8 MB, which is 2.1% of compressed packet traces. (Tcpdump records 16.8 GB, or 4.9 GB with compression.)

The CPU overhead is 1.49%. The overhead is higher than in Spark, because the client-server software only uses CPU to send and receive data, without any data processing. In fact, it spends 99.78% of its CPU time in the networking stack (including DETER). So 1.49% is very close to the lower bound of DETER CPU overhead.

5.3 Diagnosing RTO in a Testbed

RTO usually has a large impact on the latency. However, there are many different causes of RTO, and often involves different parameters. In §2.1 we have shown one case. Here we show two other causes for RTO that we see in our testbed. In all cases, TCP counters can only be the first step–knowing that timeouts and packet losses happen. But it is very hard to realize the relationship between the timeout and other events. With DETER, we can replay the connection to get the packet traces and trace the TCP execution to dig out the root cause.

Evaluation setting. We use 3 hosts connected through a single switch via 10 Gbps links. We pick two of the hosts as senders and the rest one as the receiver. Each of the senders sends one long flow (10 MB) to the receiver. One of the senders also sends a short flow (10 KB) to the receiver.

Root cause 1: Not enough dupACKs. In this case, the short flow experiences RTO. We use DETER to replay the connections and capture the packet trace. The trace shows that the short flow sends 7 packets in the first round, and the 5-th packet gets dropped. Thus, although the 6-th and the 7-th packets trigger dupACKs, the number of dupACKs is not enough to trigger fast retransmission.

Root cause 2: Setting large TCP receive buffer size. The receive buffer size is a frequently tuned parameter for networks with different bandwidth-delay products. For example, an inter-data center connection with 100ms RTT and 1Gbps bandwidth need 12.5MB buffer size. Unfortunately, a large receive buffer can cause RTO issues. Here we show the diagnosis in an example with 10MB receive buffer.

We first replay and capture the packet trace. However, the time series of data packets and ACK packets shows a very different scenario. After a packet loss, there are more than 3 dupACKs, but the sender does not fast retransmits the lost packet. This is unexpected because just 3 dupACKs should trigger fast retransmission.

We first suspect that this may be the late FR case that we show in §5.2, so we replay again and print out dupACK threshold. But it shows that the threshold is 3.

To dig out the root cause, we replay again, and use Ftrace to get the function call graph of handling each ACK. Surprisingly, We find that TCP does not go to the dupACK branch. This means TCP even does not treat them as dupACKs. With the surprise in mind, we replay again and use TCP Probe to print the variables that are used to classify ACKs as duplicates. The `flag` variable reveals the reason: TCP does not treat the ACKs as duplicate because the flag's `WIN_UPDATE` bit is set [9]. This means each of these ACKs carries a different window size. We confirm this in the packet trace: each ACK carries a larger window size.

The direct cause for this problem is that the receive buffer size is very large. The receive window starts with a small size, and increases two MSS per received data packet until reaching configured receive buffer size. Thus, the window size keeps growing throughout the lifetime of this connection. However, this also suggests a potential optimization to TCP that it should have a smarter classification for dupACKs.

5.4 Evaluating Switch Queue Replay

Now we evaluate the accuracy of replaying switch queues in our testbed and simulation. We first run traffic in our testbed, and replay the queue to evaluate the accuracy. Then to understand how the switch queue replay works under more switches and more congestions, we run empirical traffic in a large scale simulation, and replay the queues.

5.4.1 Evaluation with Testbed

Evaluation setting: The testbed comprises 3 hosts. To get the groundtruth of the queue content, we use a NetFPGA switch and program it to send out the queue content through the unused port. The switch has a total of 393 KB buffer shared across 3 ports[10]. The MTU is 1500 B. The host clocks are synchronized with 100s of nanoseconds precision by Precision Time Protocol [2].

Because congestion is the most challenging scenario to replay, we set traffic to have severe congestion. We use 2 hosts as senders and the rest one as a receiver. Each of the two senders generates 2 long flows (10 MB each) to the receiver simultaneously. Each sender also generates 4 short flows (10 KB each) to the receiver, one every 5 ms. So there are a total of 4 long flows and 8 short flows.

[10]We use the commonly used dynamic threshold [31] with $\alpha = 4$.

During the runtime, we use DETER to collect data, and also collect the content of the congested queue. Then we first replay each connection to get the packet trace, and replay the queue. We replay the queue in both the original testbed, and in a simulation. The simulation has the same topology, and simulates the same link throughput, latency, and buffer setting as the NetFPGA switch.

Accuracy: The metric we use is *queue content difference*: the difference between the runtime queue q_{run} and the replay queue q_{rep} that each packet sees. Formally, we define $qdiff = \sum_{f \in q_{run} \cup q_{rep}} |f.size_{run} - f.size_{rep}|$, where $f.size_{run}$ means the bytes of flow f in the queue during the runtime and $f.size_{rep}$ is for the replay.

On average the queue content difference is 0.57 MTU in the testbed, and 1.0 MTU in the simulation. On the 99-th percentile, the difference is 4.83 MTU in the testbed, and 3.85 MTU in the simulation, both of which are very low compared to the buffer size. Replay in the testbed has a slightly higher tail difference because timing variations (e.g., thread scheduling) exist in the testbed, but not in the simulation.

5.4.2 Evaluation in Large Scale Simulation

Our testbed evaluation shows that the replay is effective for one switch. In production, there are more hosts, multiple layers of switches, and more congestions across the switches. So we use simulation to evaluate a larger scale network.

Evaluation setting: We run the simulation in ns3 [15], with 320 switches and 1024 hosts connected through a K=16 Fat-Tree with 10 Gbps links. Each switch has 2 MB buffer, shared by all its 16 ports[11]. To simulate the clock synchronization error, we add a delta to each host's clock, with a uniform distribution between 0 and 5 us[12].

The traffic includes both empirical background traffic that follows the flow size distribution of a web search workload [26], and incast traffic. The source and the destination of each background flow are chosen uniformly random. The flow arrival rate follows a Poisson process, and we vary the flow arrival rate to achieve different levels of traffic load, from 10% to 80%. We also generate the incast traffic by having the client simultaneously requests 40 servers, each of which sends back 250 KB response (10 MB total response size). We generate 2400 incast per-second.

To understand how the sampling affects the accuracy, we sample the sending times with different threshold of error: 2 us, 5 us, and 10 us. We then replay the queues.

Accuracy: Figure 8 shows the queue content difference of all queues in the network. The difference increases mildly

[11]For the buffer sharing policy, we use the commonly used dynamic threshold [31] with $\alpha = 4$.

[12]PTP in LAN can achieve sub-microsecond accuracy, and under 3.2 us in WAN most of the time [12]. More advanced clock synchronizations [41, 34] guarantee sub-microsecond accuracy. We choose 5 us to be conservative.

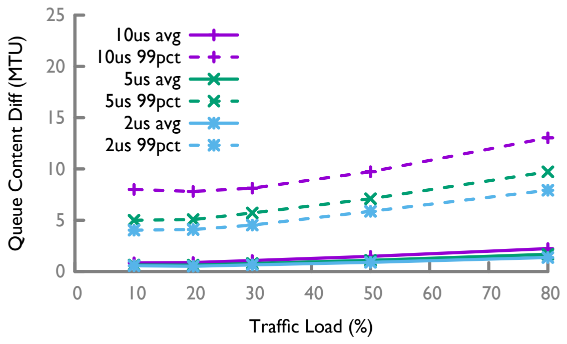

Figure 8: The queue content differences of replay in simulation.

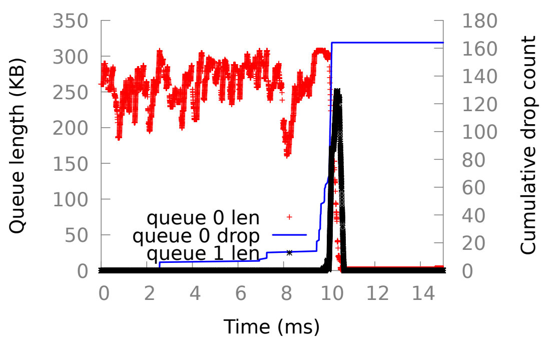

Figure 9: The lengths of two queues that share the buffer.

with higher load, both on average and at the 99-th percentile. For example, with 5 us threshold of error, at 30% load, the maximum load of most data centers in practice [50, 14], the differences are 0.78 MTU on average and 5.7 MTU at 99 percentile. At 80% load, an extremely high load, the differences are 1.7 MTU on average and 9.7 MTU at the tail. It also shows that a 5 us threshold achieves relatively good accuracy: it only increases less than 0.3 MTU (on average) and less than 1.8 MTU (at tail) difference compared to 2 us.

We also compare the packet drop error with and without the drop accuracy improvement. The drop error is $\frac{\#false\_accept+\#false\_drop+\#drop\_wrong\_location}{\#packets\_dropped\_in\_either\_runtime\_or\_replay}$. Our evaluation shows that the drop error reduces significantly. For example, for 5 us sampling threshold at 30% load, the error reduces from 58.3% to 2.87%.

The drop error is low under various loads, from 2.52% at the 10% load, to 3.81% at the 80% load. There is no false-drop, as the simulation can avoid this (§3.4). Most errors are false-accepts. Only less than 0.37% of the drops show up at wrong locations, which means we can trust the drops in the replay with high confidence, because only 0.37% of them give wrong locations. Since 80% load is extremely high and we also added incast traffic, we believe most data centers would not stress the network at this level, so we believe the drop error rate is low in general.

5.4.3 Diagnosing RTO Using Queue Information

Sometimes RTO can be caused by the queuing mechanisms of switches. We run the traffic in a 4 host (A, B, C, D) testbed. B and C respectively send 5 long flows (500MB each) to A. In the middle of the long flow transmission, A, C and D respectively send 5 short flows (100KB each) to B simultaneously. Two of the long flows from C to A experience RTO. Using DETER to replay them, we find that they both drop a whole window of packets, at the same time. But this time we cannot find any problem in the TCP stack. So we use the packet traces for all the connections to replay switch queues in an ns3 simulator.

During the replay, we collect all the enqueue and drop events at the switch. The packets are dropped at queue 0 of the switch. Figure 9 shows the length and the cumulative drop count of queue 0. At around 10 ms, there is a sudden increase in drops. Unexpectedly, the queue length is decreasing at the same time. We suspect that the switch buffer sharing [31] causes this problem, because the threshold of a queue decreases when the total buffer utilization of the switch grows.

So we replay again and monitor other queues of the switch. We find that a burst of packets builds up queue 1 at the time of queue 0 drops packets. This confirms our hypothesis.

This problem could also happen in data centers because most data center switches use shared memory across different queues. The threshold of any queue is proportional to the total free buffer size. If the switch buffer utilization suddenly increases, the threshold shrinks, which causes temporary blackhole at the almost-full queues (e.g., queue 0 in Figure 9). The sudden increase in switch buffer utilization can happen because of incast, which is common in data centers.

6 Related Work

Replay systems. There are many replay systems for kernel, multicore applications and distributed systems [52, 40, 33, 49, 27, 37, 17]. They record the input and interaction of the target of replay (a subset of components of the entire system) with the rest of the system to isolate the target, and then make sure the target itself replays accurately. There are two ways to directly adopt such replay techniques for TCP: (1) Replay each host's TCP stack separately. This means we should record every packet as they are the input to the stack, which is a significant overhead. (2) Replay the whole network altogether, including all connections and switches, which is very expensive and hard to get right as shown in §2.2. DETER customizes replay techniques for TCP: we replay each connection (a pair of TCP stacks), and only record the mutations to the packet stream in between (drop/ECN) to reduce the overhead of recording every packet, while avoiding replay the whole network together. We also introduce customized solutions to reduce the overhead of recording non-deterministic variables inside the TCP stack.

Monitoring tools in data centers. Per-packet monitoring tools [18, 10, 38] and TCP execution tracing tools [13] provide detailed information for diagnosis, but running them continuously is too expensive. To reduce overhead, people collect coarser-grained information such as TCP counters [58, 28] or per-flow stats on the host [53] or switches [4, 43]. There are also query systems (e.g., Everflow [59], Trumpet [46], Marple [47]) that allow operators to specify the packets and events to capture in a network. DETER is complementary to these works in that it enables deterministic replay for debugging the same performance problem iteratively. DETER requires low recording overhead at runtime and allow operators to use all kinds of monitoring tools during the replay.

Other network-related replay. OFRewind [57] replays the switch control plane, while DETER replays TCP and the switch data plane. Monkey [30] and Swing [55] are tools that synthesize testing traffic based on the runtime recorded traffic pattern, while DETER focus on replay for diagnosis.

7 Discussion

Extension to other network transport features: Here are a few examples of transport features that may affect the replay.

Generic receive offload (GRO): If GRO [3] is enabled, we also need to record the way it merges packets. It just requires recording the number of packets being merged into one segment, which is available in the skb metadata and just costs 6 bits per merge. Usually each merging contains 10s of packets, so the overhead is low. During the replay, the packet corrector should also merge the incoming packets as recorded.

Delay-based congestion control (CC): Our current prototype is based on loss-based CC. To extend our solution to delayed-based CC, we need to record the timestamps that used for updating CC states. We can compress them a lot, because consecutive timestamps differ by a few microseconds most of the time, so we just need a few bits to record the delta.

RED in switch: RED randomly drop packets. Replaying the queues and drops may have a large error in this case, but replaying TCP connections is not affected. This shows the benefit of our design decision: decoupling the replay of each individual connection, so that it does not depend on switches.

Use cases of DETER. DETER is designed for ease of use. The only requirement is that the user turns on DETER on both endpoints of the connection, which is often the case for network operators and cloud tenants. Internet application developers can also use DETER for performance testing. Data center network operators may also benefit from replaying the switch queues, because they may have the network topology and switch data plane simulators.

Host stack changes. If the host stack changes, DETER may need to change accordingly, but it is not hard. First, Linux already abstracts CC out of basic TCP framework, so changes to CC does not need to recode DETER in the basic framework, which contains most of the recording. Besides, we have principles for what to record and replay (Table 1 and §4), so it would be easy to identify the required changes to DETER.

We expect the recording overhead would not change much with stack changes, because most of the overhead comes from socket calls and lock acquisitions, both of which are not sensitive to stack changes: socket call is determined by the applications, and most lock acquisitions are for receiving packets whose amount is determined by traffic volumes. The overhead associated with kernel variables is very small with our technique of recording their updates or influences, and we believe this benefit remains in the future.

Generality to other transport protocols. We believe the replay technique is general across different protocols. Basically, what other transport protocols do are not very different from TCP: reads from/writes to applications, sends/receives packets, and possibly controls sending rate based on packet measurement. Similar to TCP, we just need to record the interaction with the application and the network, and then make sure we handle the concurrency inside the protocol.

Network failures. Network failures (e.g., routing fluctuations or blackholes) do not affect DETER replaying the connections, but do affect DETER replaying the switch queues which assumes that the routing states are stable. However, network failures are themselves bigger problems than the problems related to switch queueing, and there are many other works focus on addressing such issues [59, 36, 45, 56, 43]. DETER is complementary to these works, because it helps to understand how TCP reacts to such conditions.

Storage overhead of socket calls. Usually the number of socket calls is much smaller than the number of packets. Production data center survey [26, 35] shows that most network bytes are from large flows (>1 MB), which usually mean large send/receive sizes. Moreover, even if an application has many short messages, the developers tend to batch them into a large one to reduce the CPU overhead. If some network does only have applications that generate small socket calls, recording every socket calls may be high overhead.

8 Conclusion

DETER enables deterministic TCP replay, which can reproduce performance problems, provide packet traces and support tracing of TCP executions. DETER eliminates the butterfly effect by replaying individual TCP connections separately and capture all the interactions between a TCP connection with the application and the network in a lightweight fashion. We demonstrate that DETER is effective in diagnosing a variety of TCP performance problems.

9 Acknowledgement

We thank our shepherd Alex C. Snoeren and NSDI reviewers for their helpful feedback. We thank Wei Bai for providing diagnosis cases. We also thank Danyang Zhuo, Yurong Jiang, Sivaramakrishnan Ramanathan and Bradley McDanel for providing feedback. This paper is supported by the NSF grants CNS-1834263.

References

[1] ftrace, 2008. https://www.kernel.org/doc/Documentation/trace/ftrace.txt.

[2] IEEE Standard 1588-2008, 2008. http://ieeexplore.ieee.org/document/4579760/.

[3] Generic receive offload, 2009. https://lwn.net/Articles/358910/.

[4] NetFlow, 2009. http://www.cisco.com/go/netflow/.

[5] Broadcom moves from simulation to emulation with Mentor, 2014. https://www.electronicsweekly.com/uncategorised/broadcom-moves-from-simulation-to-emulation-with-mentor-2014-01/.

[6] Empirical Traffic Generator, 2014. https://github.com/datacenter/empirical-traffic-gen.

[7] NetFilter, 2014. http://www.netfilter.org/.

[8] Linux perf, 2015. https://perf.wiki.kernel.org/index.php/Main_Page.

[9] TCP window updates combined with dup acks sent in response to packet loss, 2015. https://www.ietf.org/mail-archive/web/tcpm/current/msg09480.html.

[10] In-band Network Telemetry, 2016. http://p4.org/p4/inband-network-telemetry.

[11] Cisco Packet Tracer, 2016. https://learningnetwork.cisco.com/docs/DOC-29644.

[12] IEEE 1588 PTP clock synchronization over a WAN backbone, 2016. https://www.endace.com/ptp-timing-whitepaper.pdf.

[13] TCP Probe, 2016. https://wiki.linuxfoundation.org/networking/tcpprobe.

[14] Microsoft Keynote at SIGCOMM 2017, 2017. http://conferences.sigcomm.org/sigcomm/2017/files/program-kbnets/keynote-2.pdf.

[15] Network Simulator 3, 2017. https://www.nsnam.org/.

[16] RACK: a time-based fast loss detection algorithm for TCP, 2017. https://tools.ietf.org/html/draft-ietf-tcpm-rack-02.

[17] Mozilla RR, 2017. https://rr-project.org/.

[18] Tcpdump, 2017. http://www.tcpdump.org/tcpdump_man.html.

[19] DCTCP Bug, 2018. https://github.com/torvalds/linux/commit/27cde44a259c380a3c09066fc4b42de7dde9b1ad.

[20] Amazon EC2, 2018. https://aws.amazon.com/ec2/.

[21] Boson NetSim, 2018. http://www.boson.com/netsim-cisco-network-simulator.

[22] Time Split to the Nanosecond Is Precisely What Wall Street Wants, 2018. https://www.nytimes.com/2018/06/29/technology/computer-networks-speed-nasdaq.html.

[23] P4 language, 2018. https://p4.org/.

[24] Spark TeraSort, 2018. https://github.com/ehiggs/spark-terasort.

[25] Linux TCP Github, 2019. https://github.com/torvalds/linux/tree/master/net/ipv4.

[26] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center TCP (DCTCP). In *SIGCOMM*, 2010.

[27] Gautam Altekar and Ion Stoica. Odr: Output-deterministic replay for multicore debugging. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, 2009.

[28] Behnaz Arzani, Selim Ciraci, Boon Thau Loo, Assaf Schuster, and Geoff Outhred. Taking the blame game out of data centers operations with netpoirot. In *Proceedings of the 2016 ACM SIGCOMM Conference*, 2016.

[29] Yanpei Chen, Rean Griffith, Junda Liu, Randy H. Katz, and Anthony D. Joseph. Understanding tcp incast throughput collapse in datacenter networks. In *Proceedings of the 1st ACM Workshop on Research on Enterprise Networking*, 2009.

[30] Yu-Chung Cheng, Urs Holzle, Neal Cardwell, Stefan Savage, and Geoffrey M. Voelker. Monkey see, monkey do: A tool for tcp tracing and replaying. In *Usenix*, 2004.

[31] Abhijit K. Choudhury and Ellen L. Hahne. Dynamic queue length thresholds for shared-memory packet switches. *IEEE/ACM Trans. Netw.*, 1998.

[32] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communication of the ACM*, 2013.

[33] Dennis Geels, Gautam Altekar, Scott Shenker, and Ion Stoica. Replay debugging for distributed applications. In *Proceedings of the Annual Conference on USENIX '06 Annual Technical Conference*, 2006.

[34] Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosenblum, and Amin Vahdat.

Exploiting a natural network effect for scalable, fine-grained clock synchronization. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, 2018.

[35] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. Vl2: A scalable and flexible data center network. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*, 2009.

[36] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, Zhi-Wei Lin, and Varugis Kurien. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *Proceedings of the 2015 ACM SIGCOMM Conference*, 2015.

[37] Zhenyu Guo, Xi Wang, Jian Tang, Xuezheng Liu, Zhilei Xu, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. R2: An application-level kernel for record and replay. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, 2008.

[38] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Mazières, and Nick McKeown. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *NSDI*, 2014.

[39] Virajith Jalaparti, Peter Bodik, Srikanth Kandula, Ishai Menache, Mikhail Rybalkin, and Chenyu Yan. Speeding up distributed request-response workflows. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, 2013.

[40] Samuel T. King, George W. Dunlap, and Peter M. Chen. Debugging operating systems with time-traveling virtual machines. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, 2005.

[41] Ki Suh Lee, Han Wang, Vishal Shrivastav, and Hakim Weatherspoon. Globally synchronized time via datacenter networks. In *Proceedings of the 2016 ACM SIGCOMM Conference*, 2016.

[42] Jialin Li, Naveen Kr. Sharma, Dan R. K. Ports, and Steven D. Gribble. Tales of the tail: Hardware, os, and application-level sources of tail latency. In *Proceedings of the ACM Symposium on Cloud Computing*, 2014.

[43] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. Flowradar: A better netflow for data centers. In *NSDI*, 2016.

[44] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. Lossradar: Fast detection of lost packets in data center networks. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies*, 2016.

[45] Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Jiaxin Cao, Sri Tallapragada, Nuno P. Lopes, Andrey Rybalchenko, Guohan Lu, and Lihua Yuan. Crystalnet: Faithfully emulating large production networks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017.

[46] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. Trumpet: Timely and precise triggers in data centers. In *Proceedings of the 2016 ACM SIGCOMM Conference*, 2016.

[47] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. Language-directed hardware design for network performance monitoring. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, 2017.

[48] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling memcache at facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, 2013.

[49] Soyeon Park, Yuanyuan Zhou, Weiwei Xiong, Zuoning Yin, Rini Kaushik, Kyu H. Lee, and Shan Lu. Pres: Probabilistic replay with execution sketching on multiprocessors. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, 2009.

[50] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. Inside the social network's (datacenter) network. In *SIGCOMM*, 2015.

[51] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. In *Proceedings of the 2015 ACM SIGCOMM Conference*, 2015.

[52] Sudarshan M. Srinivasan, Srikanth Kandula, Christopher R. Andrews, and Yuanyuan Zhou. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *USENIX Annual Technical Conference, General Track*, 2004.

[53] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. Simplifying datacenter network debugging with pathdump. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, 2016.

[54] Vijay Vasudevan, Amar Phanishayee, Hiral Shah, Elie Krevat, David G. Andersen, Gregory R. Ganger, Garth A. Gibson, and Brian Mueller. Safe and effective fine-grained tcp retransmissions for datacenter communication. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*, 2009.

[55] Kashi Venkatesh Vishwanath and Amin Vahdat. Realistic and responsive network traffic generation. In *Proceedings of the 2006 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, 2006.

[56] Xin Wu, Daniel Turner, George Chen, Dave Maltz, Xiaowei Yang, Lihua Yuan, and Ming Zhang. Netpilot: Automating datacenter network failure mitigation. In *Proceedings of the 2012 ACM SIGCOMM Conference*, 2012.

[57] Andreas Wundsam, Dan Levin, Srini Seetharaman, and Anja Feldmann. Ofrewind: Enabling record and replay troubleshooting for networks. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, 2011.

[58] Minlan Yu, Albert Greenberg, Dave Maltz, Jennifer Rexford, Lihua Yuan, Srikanth Kandula, and Changhoon Kim. Profiling network performance for multi-tier data center applications. In *NSDI*, 2011.

[59] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y. Zhao, and Haitao Zheng. Packet-level telemetry in large datacenter networks. In *SIGCOMM*, 2015.