DETER: Deterministic TCP Replay for Performance Diagnosis

Yuliang Li, Rui Miao, Mohammad Alizadeh, Minlan Yu
TCP performance diagnosis is important

- Apps are more distributed
- Increasingly rely on the TCP performance
- Tail latency is impactful
  - A single long latency slows down the entire task
- Need a diagnosis tool for TCP problems in large scale production networks
Why diagnosing TCP is hard?

- What I learned in the textbook
TCP is complex!

• Reality...
TCP is complex!

- Unexpected interactions between different components
TCP is complex!

• Unexpected interactions between diff components
TCP is complex!

- Unexpected interactions between different components
- 63 parameters in Linux TCP that tune the behaviors of different components
- Continuous error-prone development:
  - 16 bugs found in July & Aug of 2018 in Linux TCP
How do we diagnose TCP today?

Tcpdump
Detailed diagnosis is not scalable

Bandwidth

100Gbps

10,000x

10Mbps

# hosts

100,000s

10,000x

10s

Too much overhead!
Tension between more details and low overhead

- Existing tools cannot achieve both
- DETER solves it, by introducing replay
  - Lightweight recording during the runtime
  - Replay every detail

Runtime record = Data for diagnosis
Runtime record < Data for diagnosis

Details for diagnosis
Overhead
Lots of details, but high overhead
Low overhead, but miss lots of details
All details, low overhead
DETER overview

10.0.0.1:80 -> 20.0.0.1:1234 has long latency

Lightweight record
Run continuously
On all hosts

Deterministic replay
Capture packets/counters
Trace executions
Iterative diagnosis
Lightweight record         Deterministic replay
Intuition for being lightweight

**Lightweight record**

Record socket calls

**Deterministic replay**

Automatically generate packets
Non-deterministic interactions w/ many parties
Non-deterministic interactions w/ many parties

Key contribution:
• Identifying the minimum set of data that enables deterministic replay

Two challenges:
• Network wide: non-deterministic interactions across switches and TCP
• On host: non-determinisms within the kernel
Challenge 1: butterfly effect

• The closed loop between TCP and switches amplifies small noises
Challenge 1: butterfly effect

Sending time variation
- μs-level:
  - Clock drift, context switching, kernel scheduling, cache state

Switch action variation

TCP behavior variation

Runtime

Replay

Sending time variation

Switch action variation

TCP behavior variation
Challenge 1: butterfly effect

Sending time variation  
Switch action variation  
Butterfly effect  
TCP behavior variation

Runtime

Replay

Cong_win/=2

sock call → TCP

sock call → TCP

sock call → TCP

drop

Cong_win++

sock call → TCP

sock call → TCP

sock call → TCP

Cong_win/=2

sock call → TCP

sock call → TCP

enqueue
Challenge 1: butterfly effect

• To understand the impact of butterfly effect
• We try to replay a long latency problem in a 3-host testbed with 3 flows, by issuing the same set of socket calls as runtime
• Replay 100 times, but none of them reproduce the same problem.
Challenge 1: butterfly effect

- Run the same experiment in simulation, while controlling the sending time variation from 0 to 1000 ns.

What if we reduce it?

Reducing sending time variation **cannot** eliminate butterfly effect.

Even 1 ns variation still cause butterfly effect.
Challenge 1: butterfly effect

Sending time variation  \rightarrow  Switch action variation

TCP behavior variation

TCP

sock call

record & replay

TCP

sock call

record & replay
Challenge 1: butterfly effect

• Directly borrow classic kernel replay techniques?
Challenge 1: butterfly effect

- **Directly borrow classic kernel replay techniques?**
- **Solution:** record & replay **packet stream mutations**

**Runtime**

**Replay**

<table>
<thead>
<tr>
<th>Drop</th>
<th>Mark ECN</th>
<th>Reordering</th>
</tr>
</thead>
</table>

Packet stream mutations
Challenge 1: butterfly effect

- **Directly borrow classic kernel replay techniques?**
- **Solution:** record & replay **packet stream mutations**

**Runtime**

**Replay**

Drops, ECN, reordering, etc.

Drops, ECN, reordering, etc.
Challenge 1: butterfly effect

• Solution: record\&replay **packet stream mutations**
  + **Low overhead:**
    - Drop rate < $10^{-4}$;
    - ECN: 1 bit/packet;
    - Reordering is rare
  + **Replaying each TCP connection is independent**
    - Connections interact via drops and ECN, which we replay.
  + **Need no switches for replay**

Resource-efficient replay:
- Just need two hosts
Challenge 1: butterfly effect

• Solution: record & replay packet stream mutations

![Diagram showing the process of recording and replaying packet stream mutations with drop numbers and IP_ID consecutiveness.]
Challenge 2: non-determinisms within the kernel
Handling non-determinisms within the kernel

- Other handler function calls (e.g., OS timer calls timeout handler)
- Thread scheduling
  - Normal race conditions are expensive to record and replay
  - But TCP uses one lock per connection to prevent race conditions
  - So we record & replay the order of lock acquisitions of different threads
- Reading kernel variables (e.g., jiffies)

Runtime

- sock call
- Sockcall hdl
- Pkt hdl
- Timeout hdl
- Timer
- Read_jiffies
- kernel
- Record mutations
- DETER lock

Replay

- sock call
- Sockcall hdl
- Pkt hdl
- Timeout hdl
- Hdl caller
- Read_jiffies
- kernel
- Replay mutations
- DETER lock

Very few

10s of consecutive locks by the same thread, compress a lot

Value changes infrequently, only record new values

Correct input to TCP

28
Implementation

• Prototype in Linux 4.4
• Lightweight recorder (packet stream mutations, 3 types of kernel non-determinism)
  • Storage: 2.1%~3.1% compared to compressed packet header traces.
  • CPU: < 1.49%
• All data are recorded on end hosts.
• Just need 139 lines of changes to Linux TCP.
• Open source
An RTO problem in testbed

- Two senders to one receiver
  - 2 long flows (20MB) and 1 short flow (30KB)
- The short flow experiences 49 ms delay (2 orders of magnitude higher than expected)
  - In contrast, retransmission timeout (RTO) is 16ms
- TCP counters are not enough: they show 2 RTO, but $2 \times 16 < 49$. 
An RTO problem in testbed

Diagnosis Info:
- 2 RTO
- Exponential backoff

Why the receiver doesn’t ACK?
An RTO problem in testbed

Diagnosis Info:
- 2 RTO
- Exponential backoff
- Delayed ACK

DETER+Tcpdump
DETER+Ftrace

TCP expert may guess

Monitor function call graph (Ftrace)

Enters delayed ACK function
Case study in Spark

- Terasort 200 GB on 20 servers (4 cores each) on EC2, 6.2K connections
- Replay and collect trace for problematic flows

- The receiver explicitly delays the ACK, because the recv buffer is shrinking
- Caused by the slow receiver
Case study in RPC

• An RPC application running empirical DC traffic on 20 servers (4 cores each) on EC2, 280K requests

Late Fast Retransmission: fast retransmit after 10s of dupACKs.
- The threshold for dupACK increases, from 3 to 45.
- Due to reordering in the past

<table>
<thead>
<tr>
<th>Flow size (MB)</th>
<th>&lt;0.1</th>
<th>[0.1,1]</th>
<th>[1,10]</th>
<th>&gt;10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Congestion</td>
<td>149</td>
<td>35</td>
<td>25</td>
<td>2</td>
</tr>
<tr>
<td>Late FR</td>
<td>29</td>
<td>27</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ACK drops</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Tail drops</td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>RTO</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>
Other use cases

• We can diagnose many other problems in the TCP stack
  • RTO caused by diff reasons: small messages, misconfiguration of recv buf size
• We can also diagnose problems in the switches
  • Because we have traces, we can push packets into the network
  • In simulation (requires modeling switch data plane accurately)
  • Case study: A temporary blackhole caused by switch buffer sharing
Conclusion

• DETER enables deterministic TCP replay
  • Lightweight: always on during runtime
  • Detailed diagnosis during the replay

• Key challenge: butterfly effect
  • Record & replay packet stream mutations to break the closed loop between TCP and switches.