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

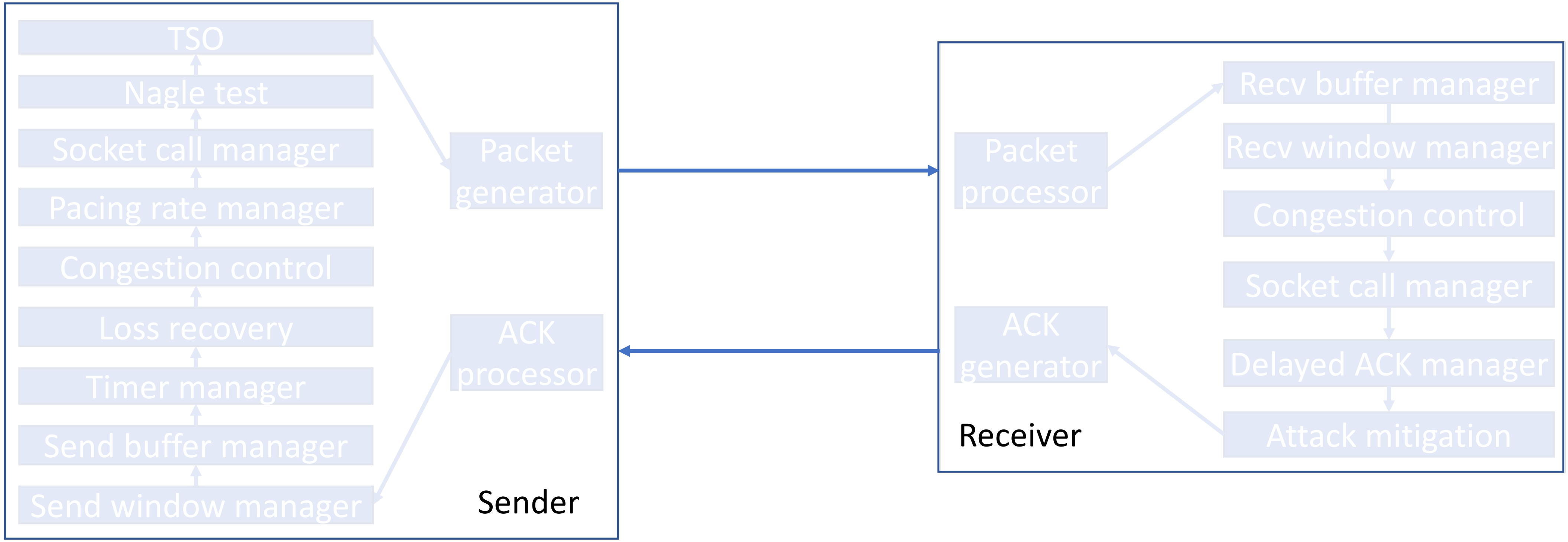
GRPC TensorFlow redis MEMCACHED APACHE Spark

Why diagnosing TCP is hard?

- What I learned in the textbook

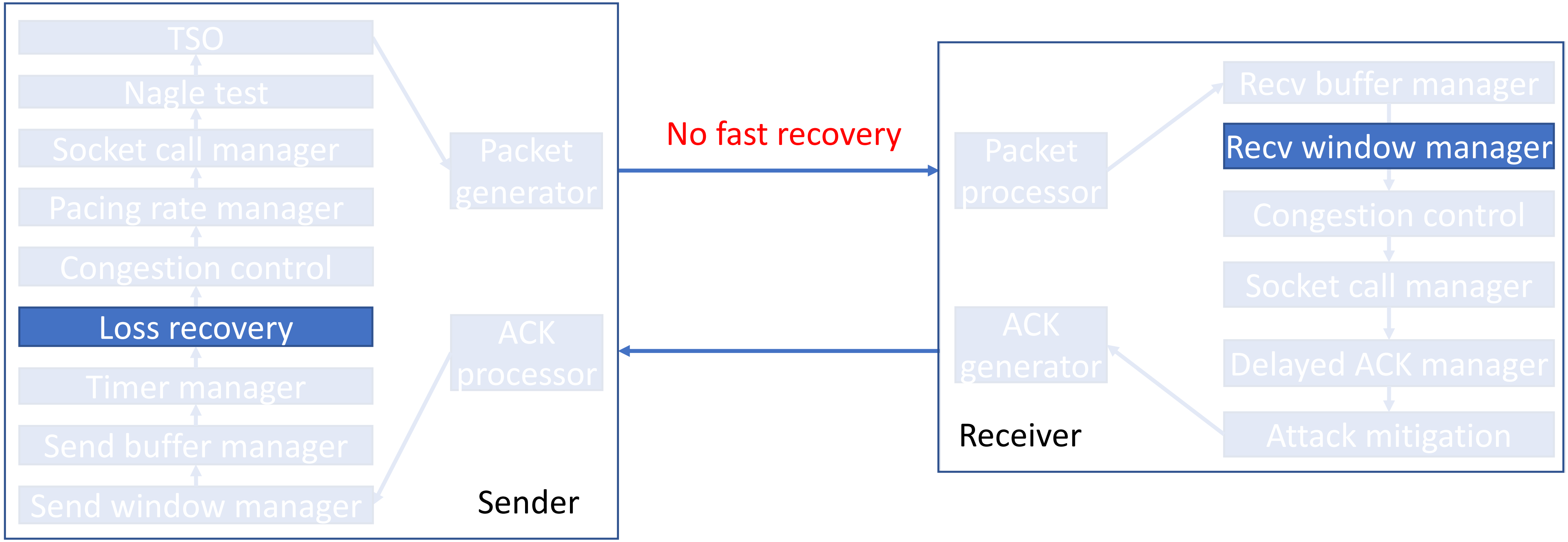
TCP is complex!

- Reality…



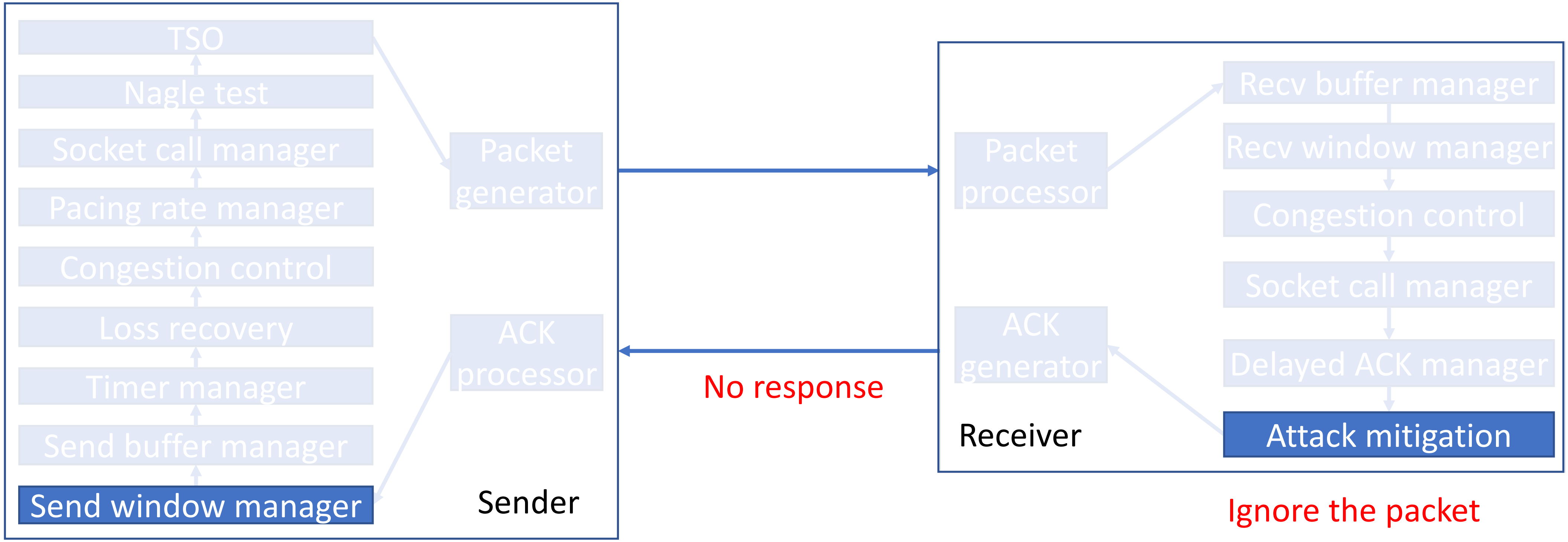
TCP is complex!

- Unexpected interactions between diff components



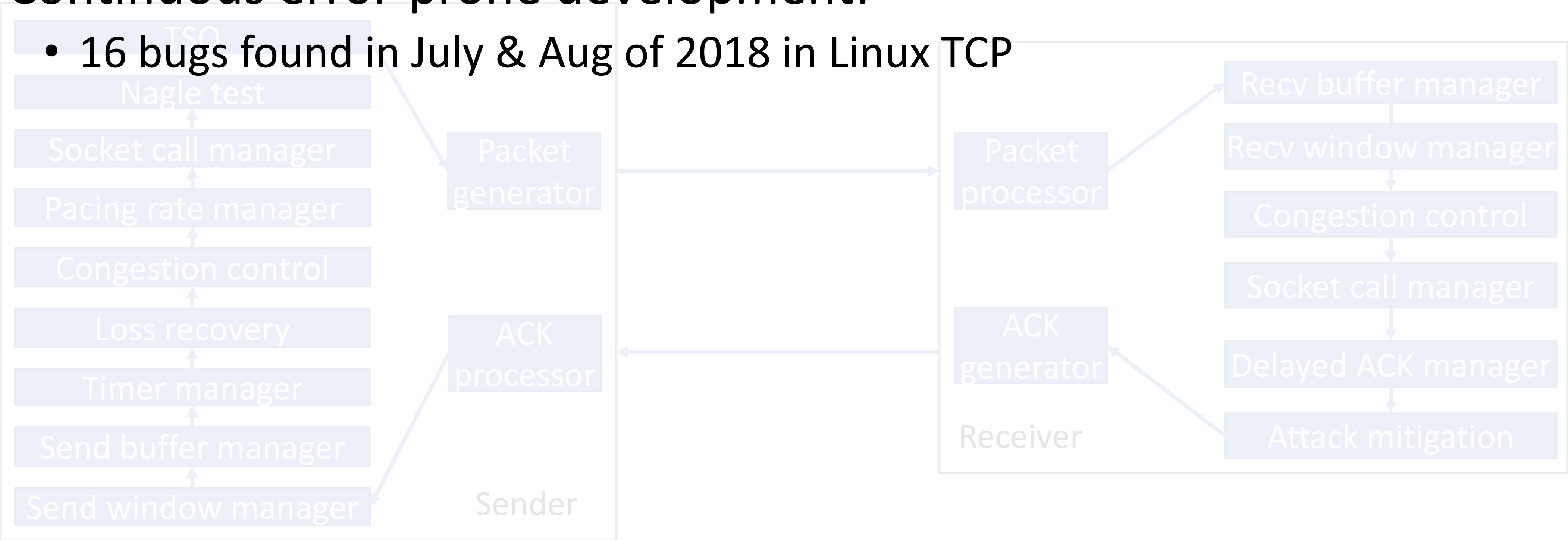
TCP is complex!

- Unexpected interactions between diff components



TCP is complex!

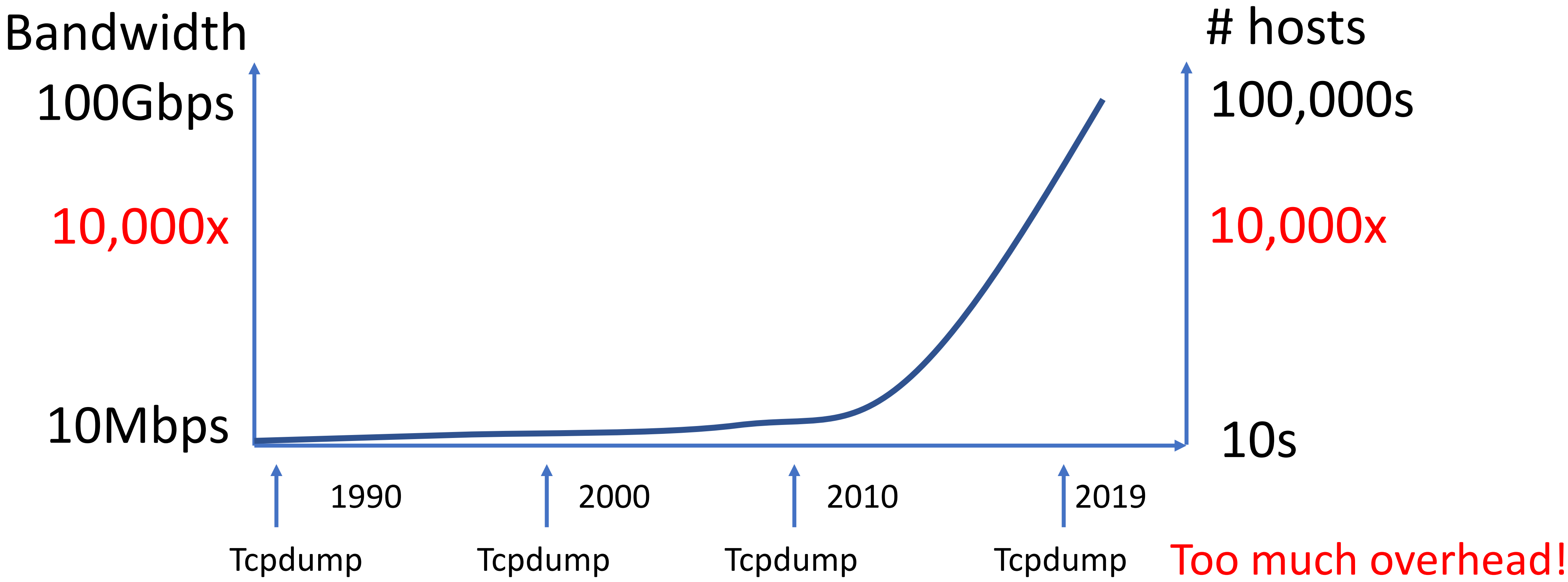
- Unexpected interactions between diff components
- 63 parameters in Linux TCP that tune the behaviors of diff components
- Continuous error-prone development:
  - 16 bugs found in July & Aug of 2018 in Linux TCP



How do we diagnose TCP today?

Tcpdump

Detailed diagnosis is not scalable

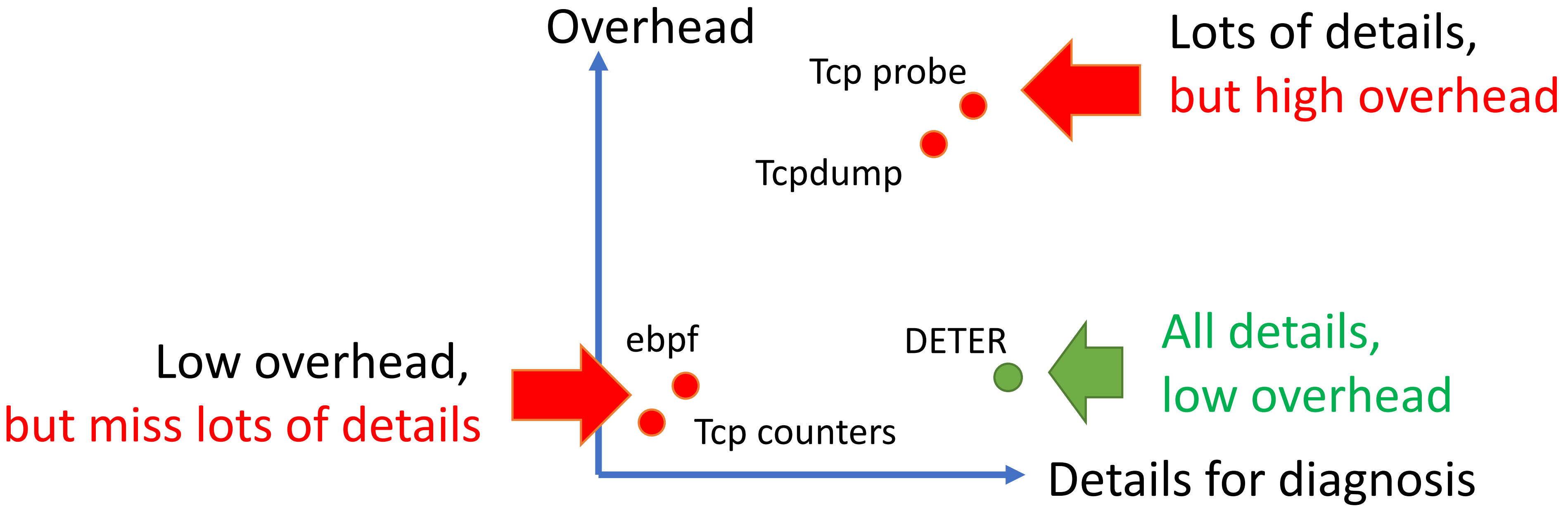


Tension between more details and low overhead

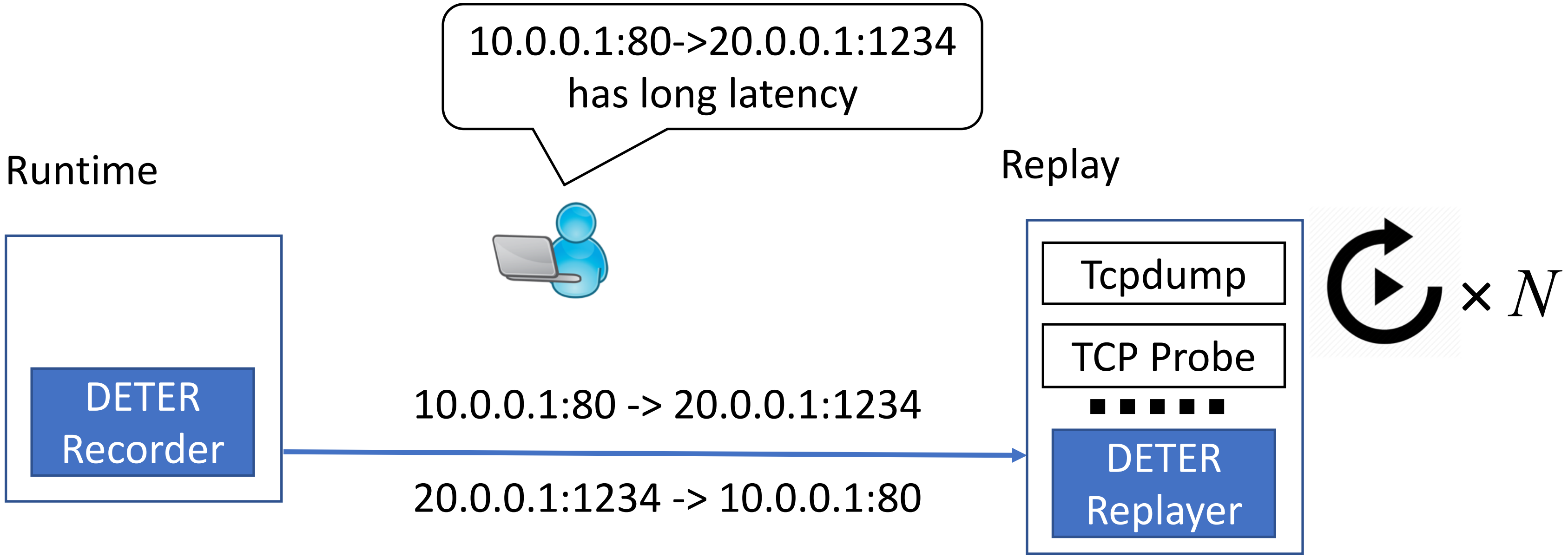
- Existing tools cannot achieve both
- DETER solves it, by introducing replay
  - Lightweight recording during the <u>runtime</u>
  - <u>Replay</u> every detail

Runtime record = Data for diagnosis

Runtime record < Data for diagnosis



DETER overview



Lightweight record

Run continuously

On all hosts

Deterministic replay

Capture packets/counters

Trace executions

Iterative diagnosis

Lightweight record

Deterministic replay

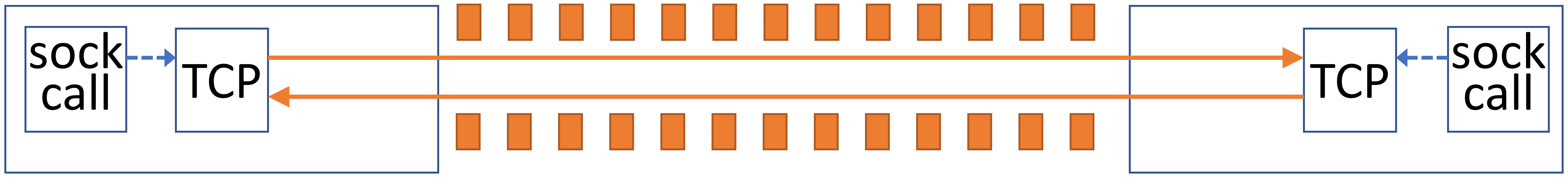
Intuition for being lightweight

Lightweight record

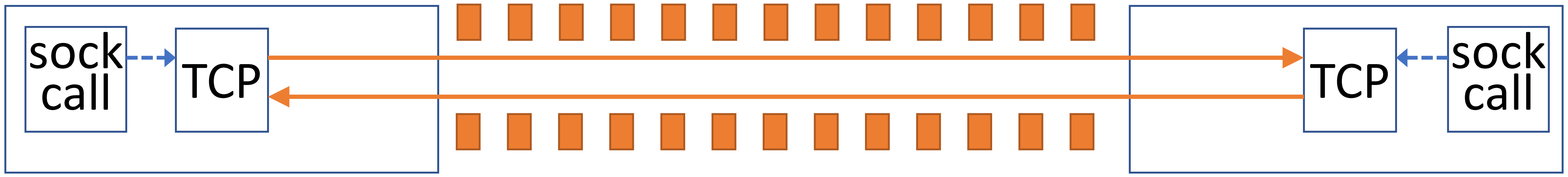
Deterministic replay FAIL!

Record socket calls

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

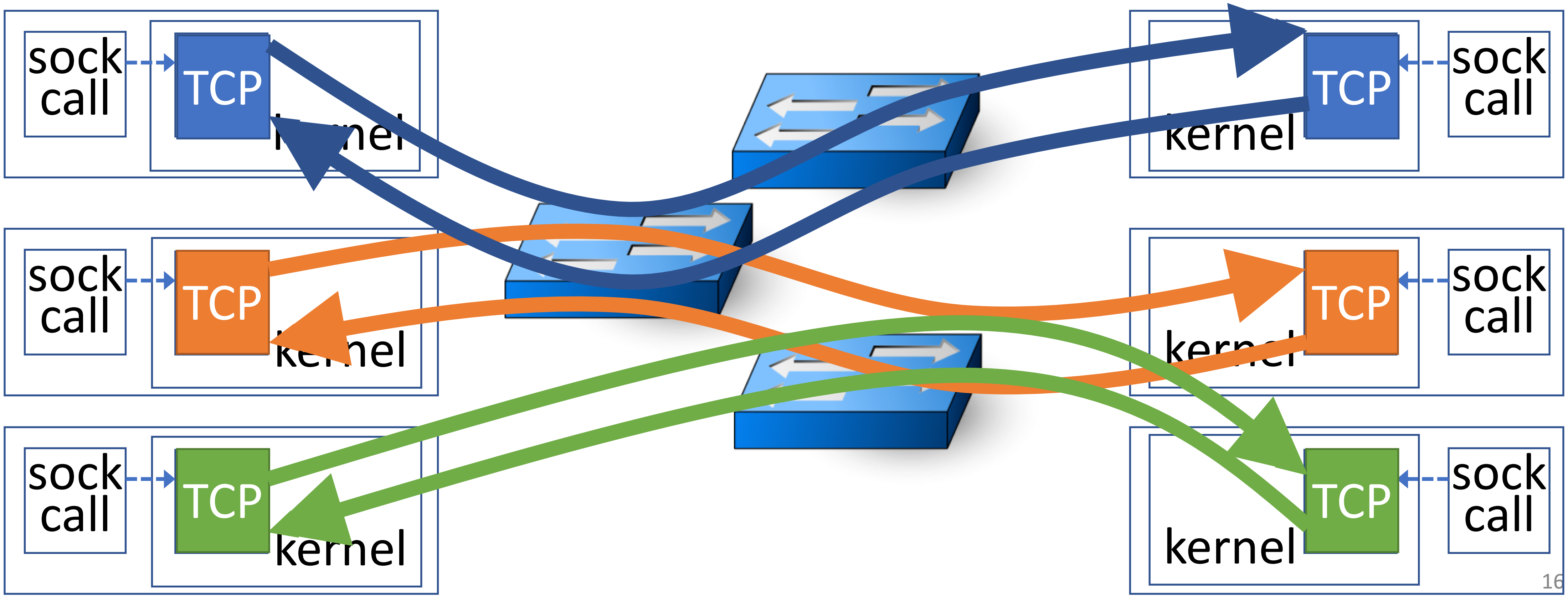
Two challenges:

- Network wide: non-deterministic interactions across switches and TCP
- On host: non-determinisms within the kernel

Butterfly effect

Challenge 1: butterfly effect

- The **closed loop** between TCP and switches amplifies small noises

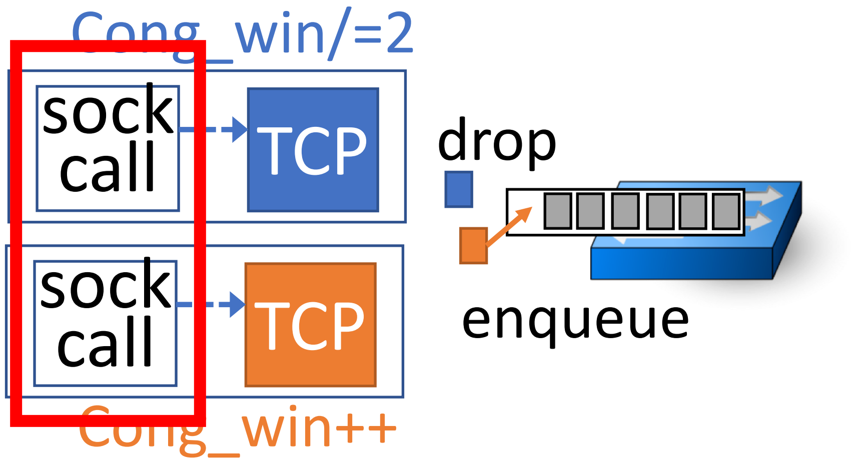


Challenge 1: butterfly effect

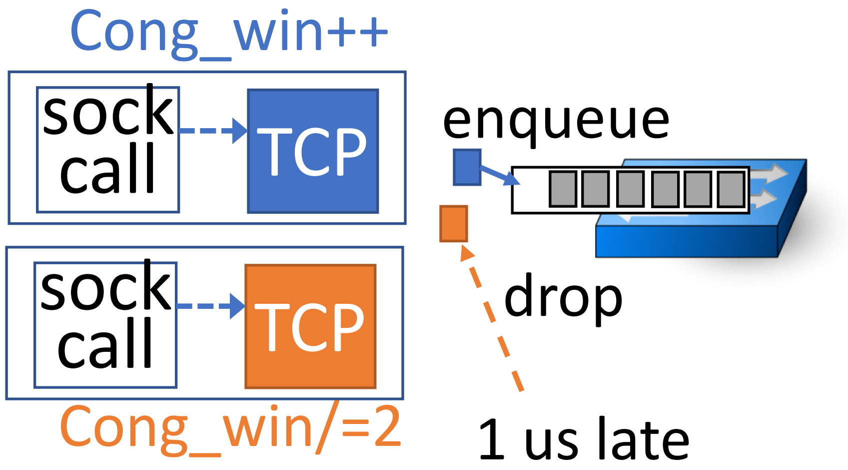
Sending time variation → Switch action variation → TCP behavior variation

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

Runtime



Replay



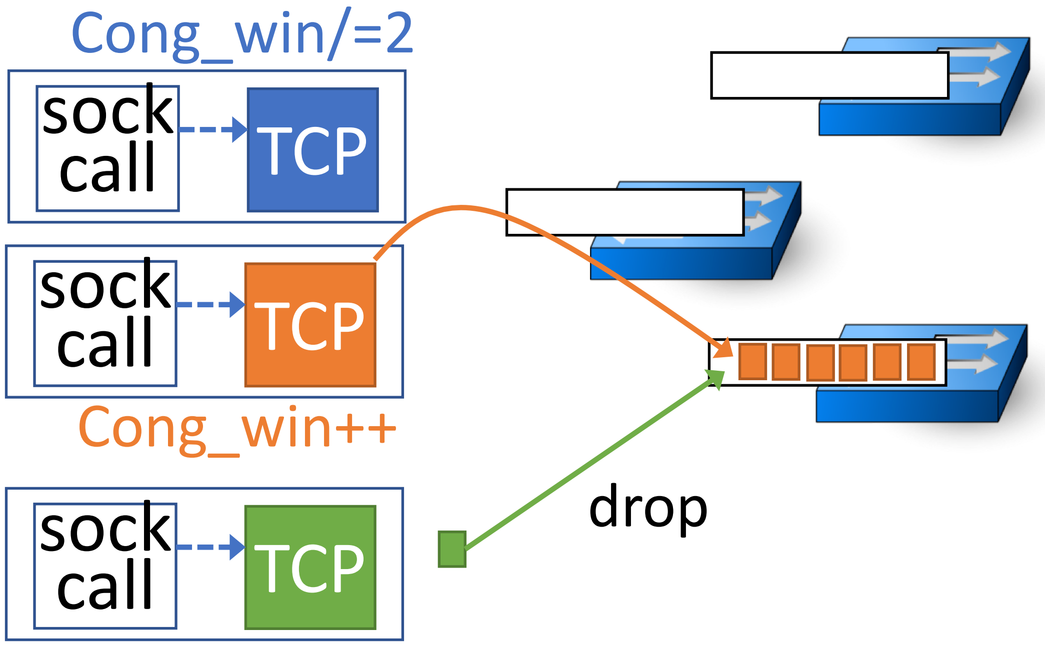
Challenge 1: butterfly effect

Sending time variation → Switch action variation

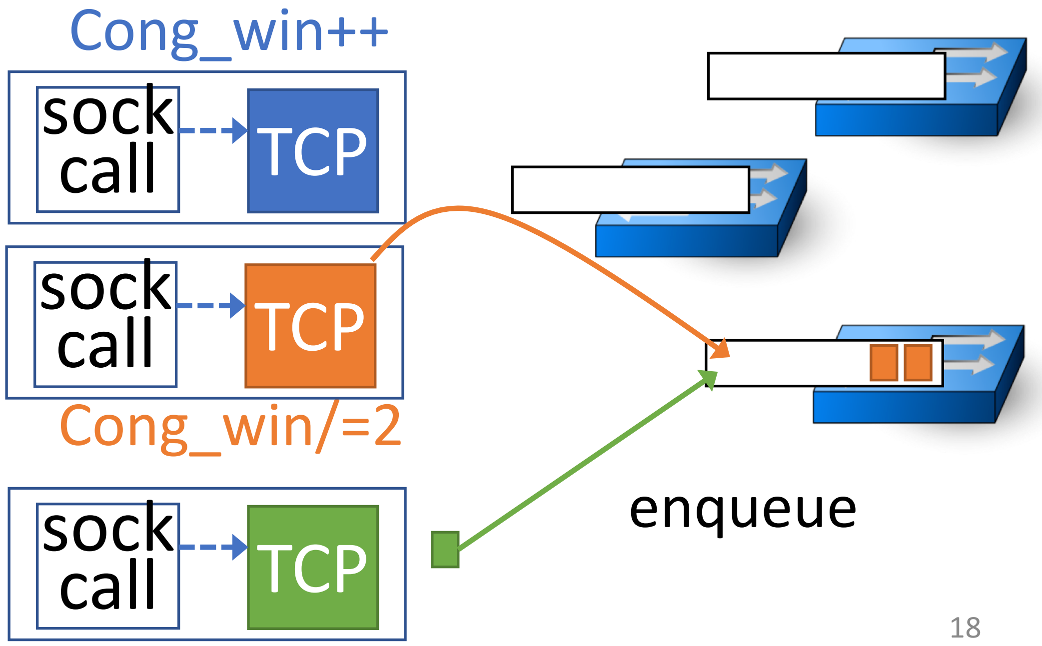
Butterfly effect

TCP behavior variation

Runtime



Replay



Challenge 1: butterfly effect

Sending time variation → Switch action variation

What if we reduce it?

Switch action variation

Butterfly effect

TCP behavior variation

- 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

Sending time variation → Switch action variation

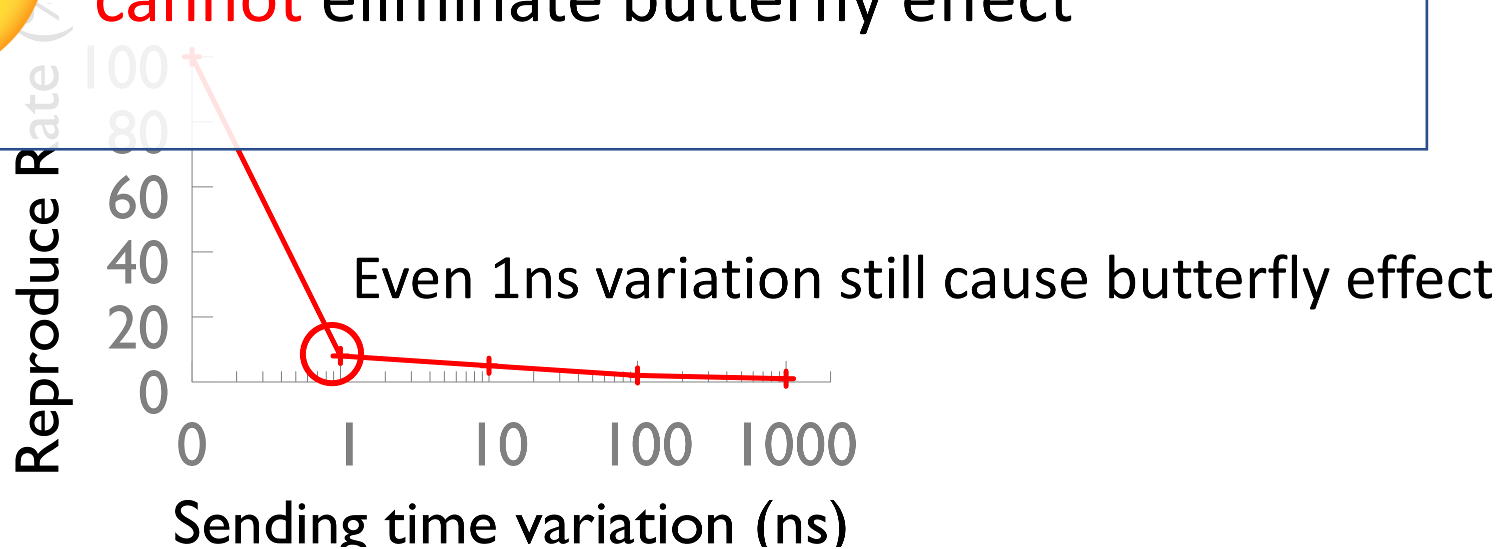
Butterfly effect

TCP behavior variation

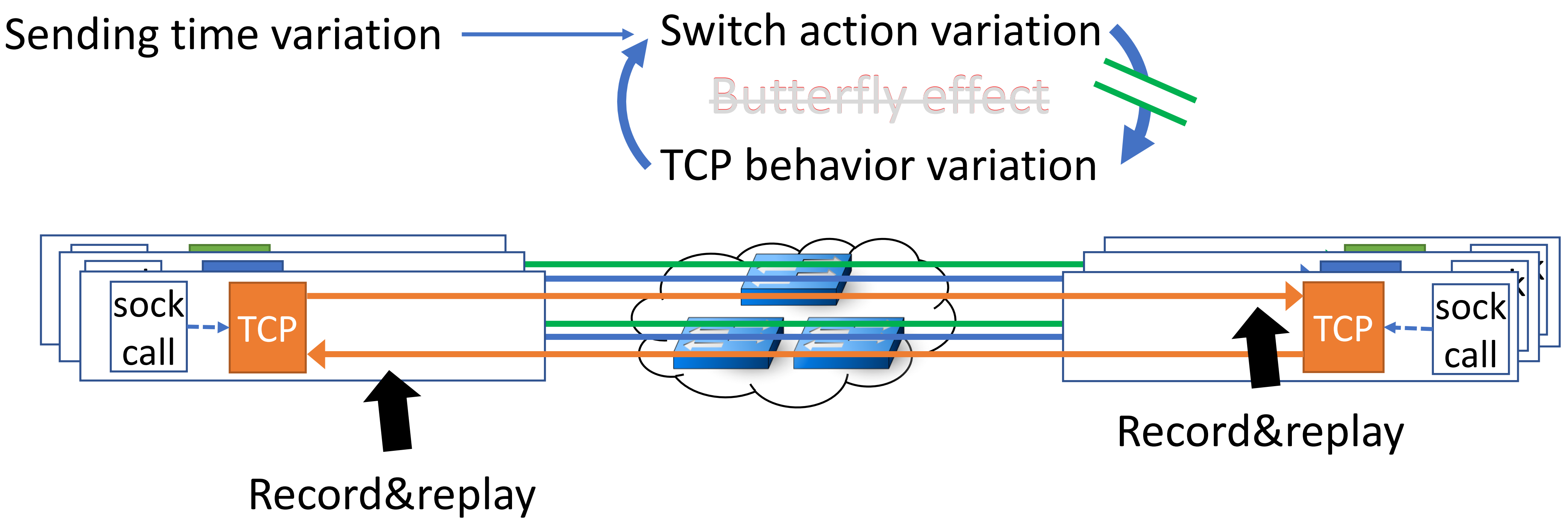
What if we reduce it?

- Run the sam eriment in simulation, while controlling the sending time variat n 0 to 1000ns

Reducing sending time variation
cannot eliminate butterfly effect



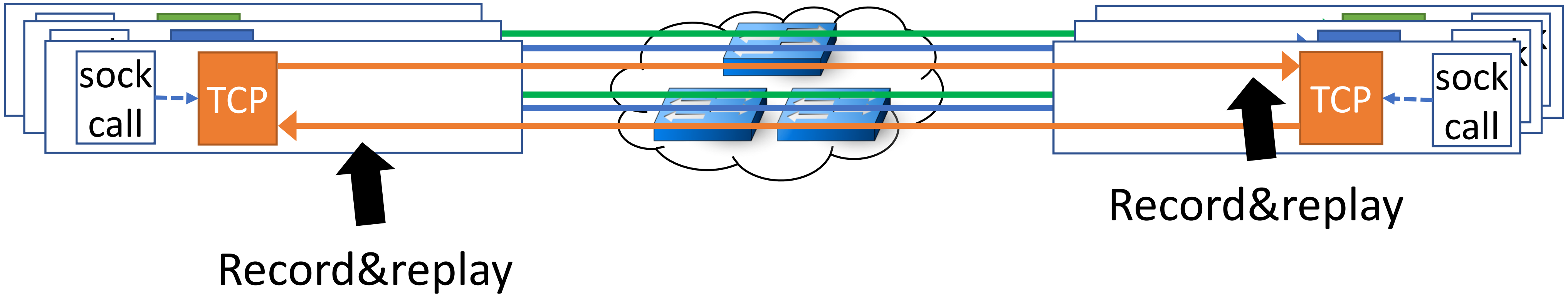
Challenge 1: butterfly effect



Challenge 1: butterfly effect

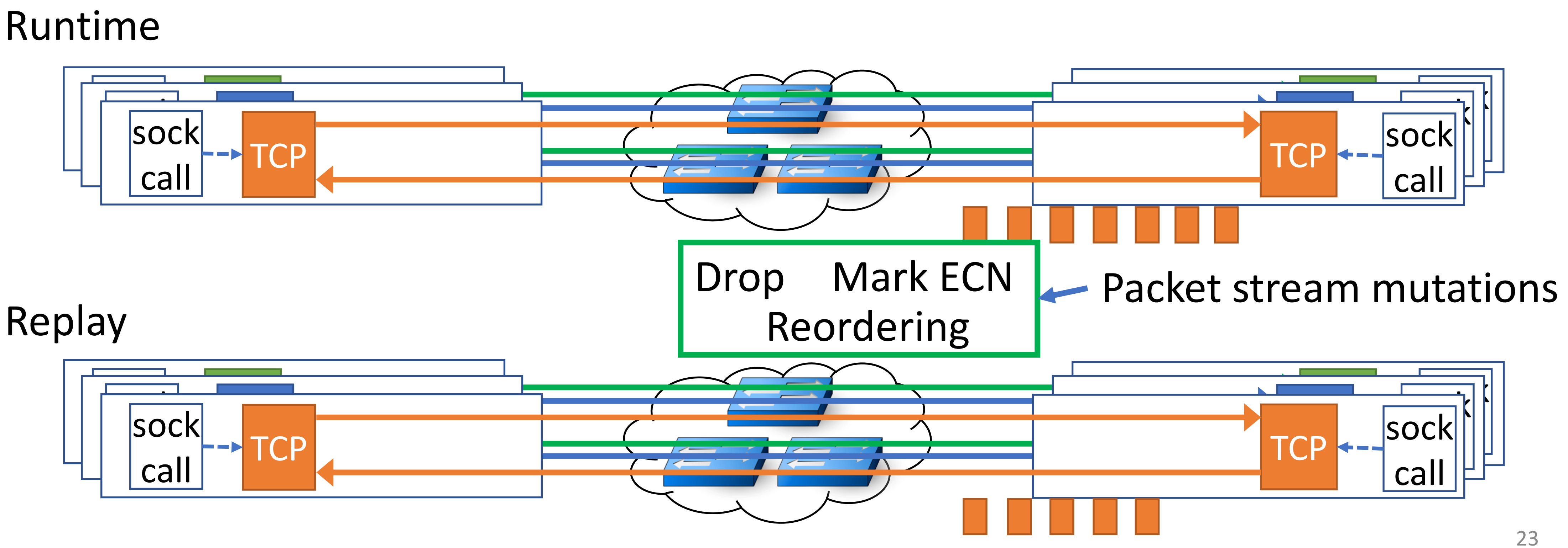
- Directly borrow classic kernel replay techniques?

High overhead



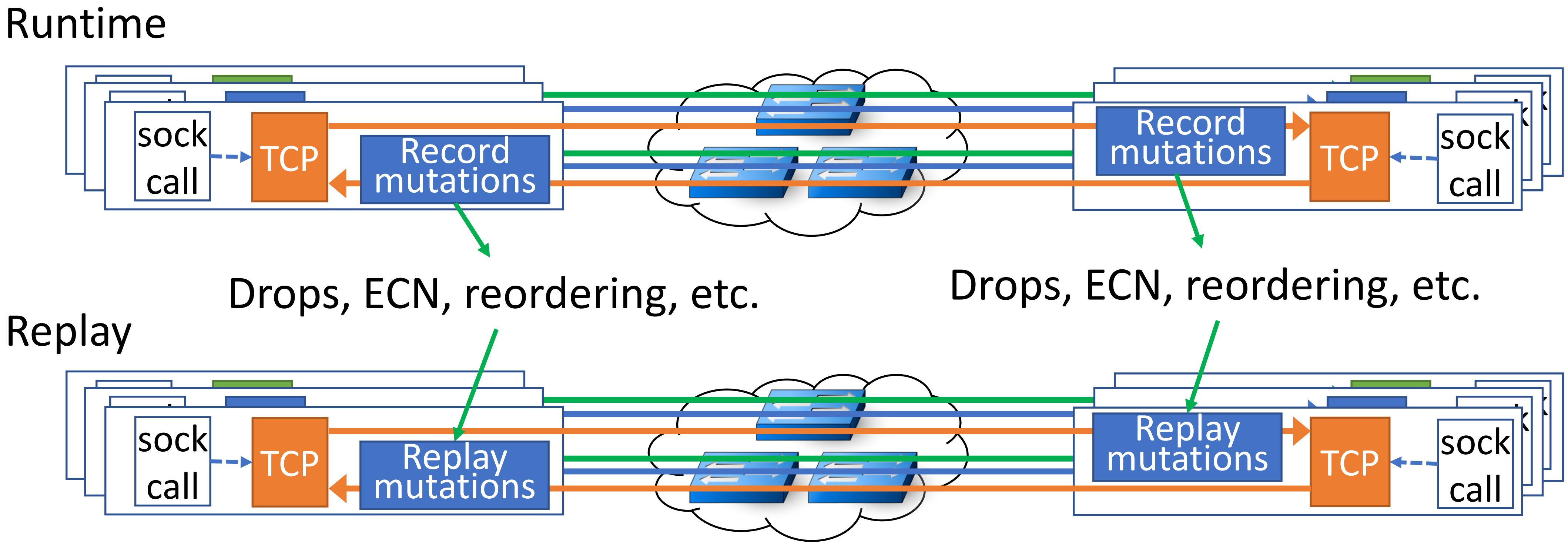
Challenge 1: butterfly effect

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



Challenge 1: butterfly effect

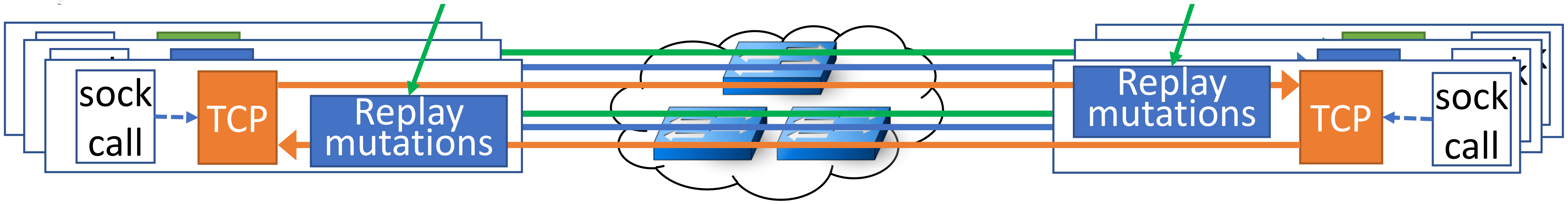
- ~~Directly borrow classic kernel replay techniques?~~
- Solution: record&replay **packet stream mutations**



Challenge 1: butterfly effect

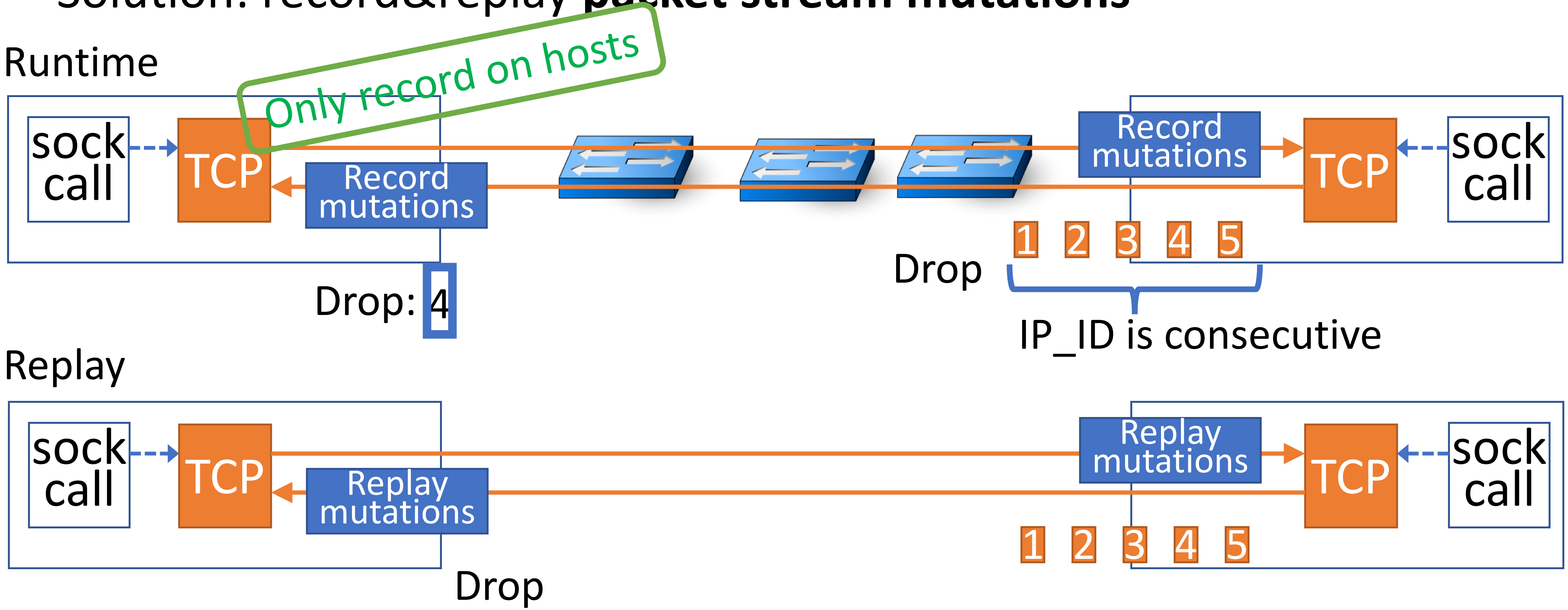
- Solution: record&replay **packet stream mutations**
  + **Low overhead**:
    Drop rate < 10^{-4};
    ECN: 1 bit/packet;
    Reordering is rare
  + **Replaying each TCP connection is independent**
    Connections interact via drops and ECN, which we replay.
  + **Need no switches for replay**

Resource-efficient replay:
- Just need two hosts



Challenge 1: butterfly effect

- Solution: record&replay **packet stream mutations**



Challenge 2: non-determinisms within the kernel

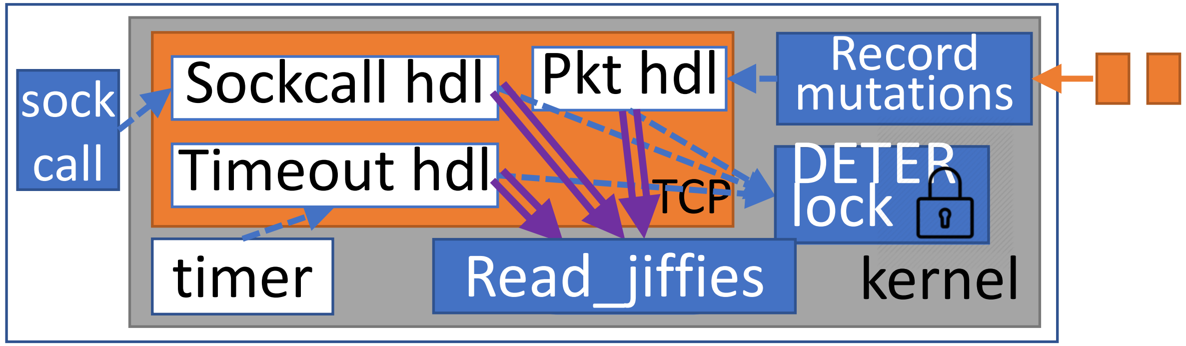
Handling non-determinisms within the kernel

- Other handler function calls (e.g., OS timer calls timeout handler) Very few
- Thread scheduling 10s of consecutive locks by the same thread, compress a lot
  - Order of lock acquisitions of diff threads
  - Normally 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 diff threads
- Reading kernel variables (e.g., jiffies)

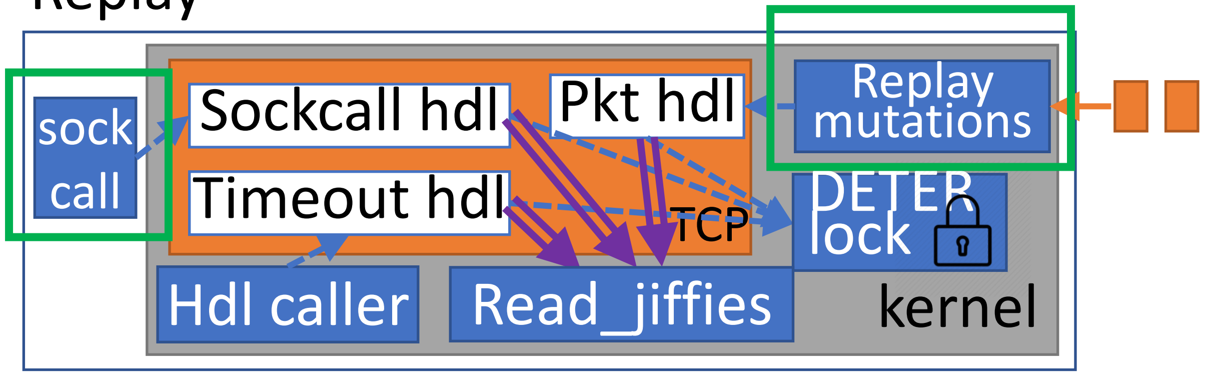
Value changes infrequently, only record new values

Correct input to TCP

Runtime



Replay



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

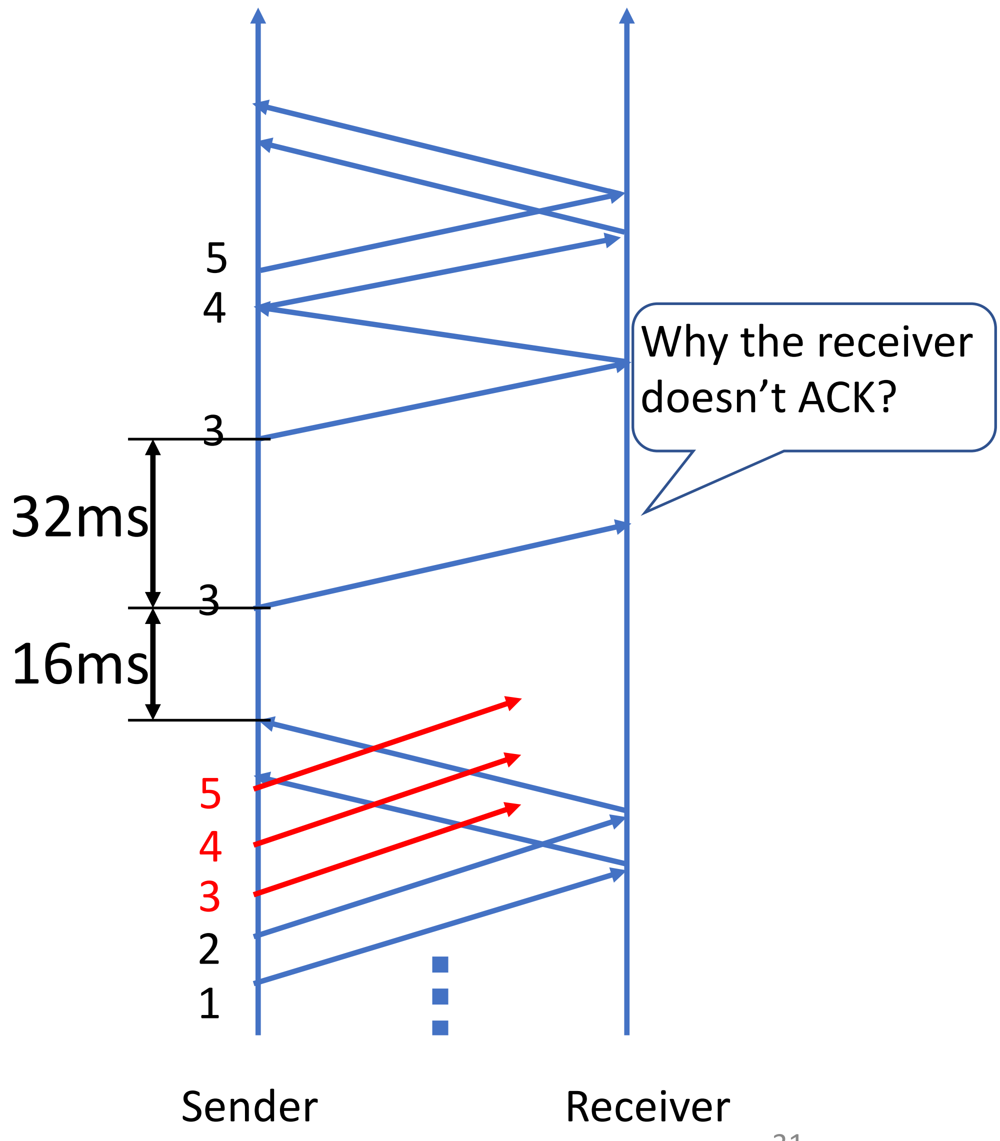
An RTO problem in testbed

- 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 shows 2 RTO, but 2*16 < 49.

An RTO problem in testbed

Diagnosis Info:

- 2 RTO
- Exponential backoff



An RTO problem in testbed

Counter → • 2 RTO

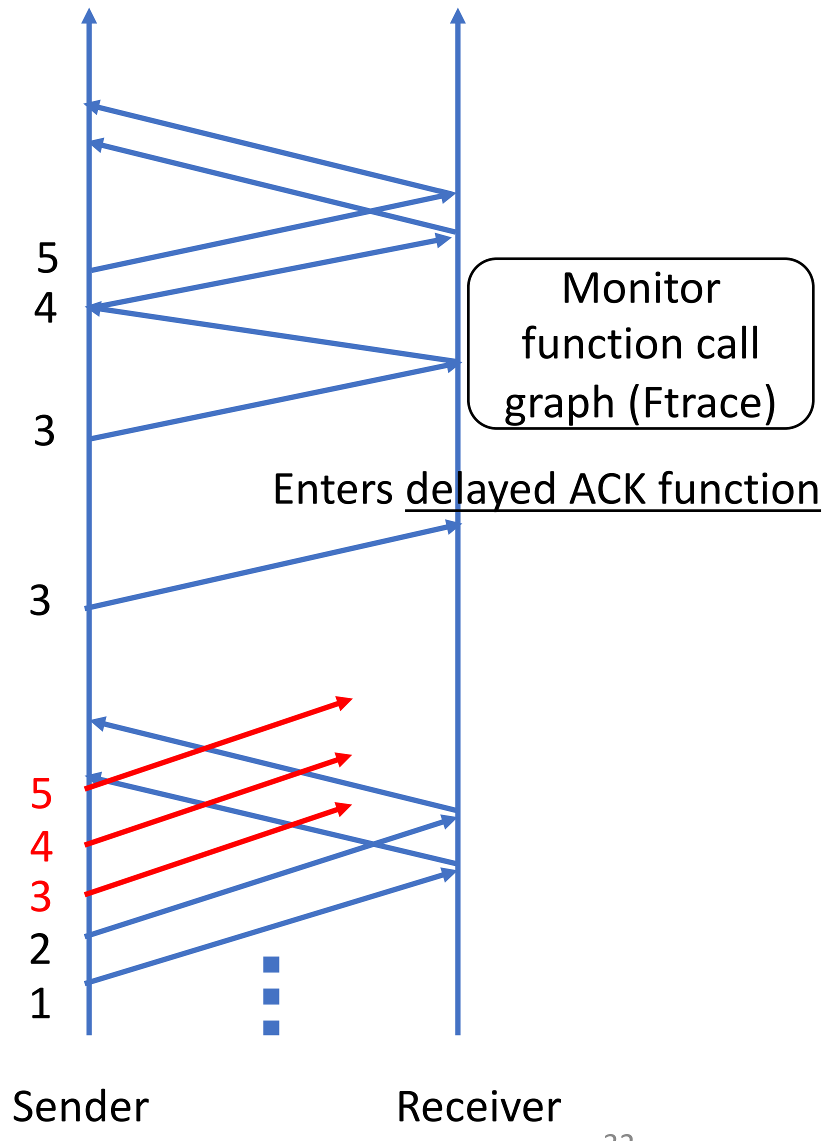
DETER+Tcpdump → • Exponential backoff

DETER+Ftrace → • Delayed ACK

Diagnosis Info:

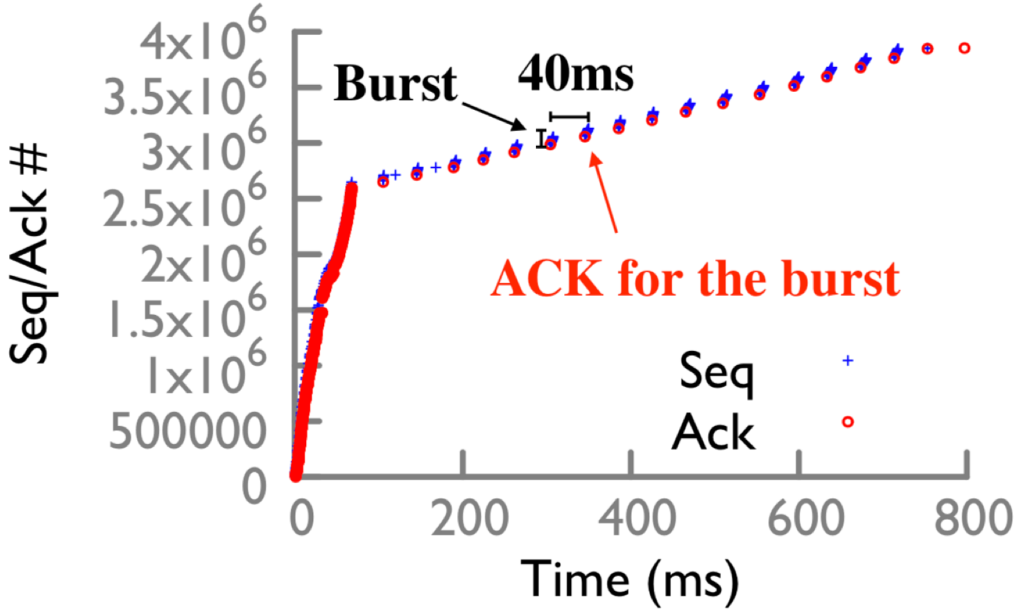
- 2 RTO
- Exponential backoff
- Delayed ACK

TCP expert may guess



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



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

- 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

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

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.