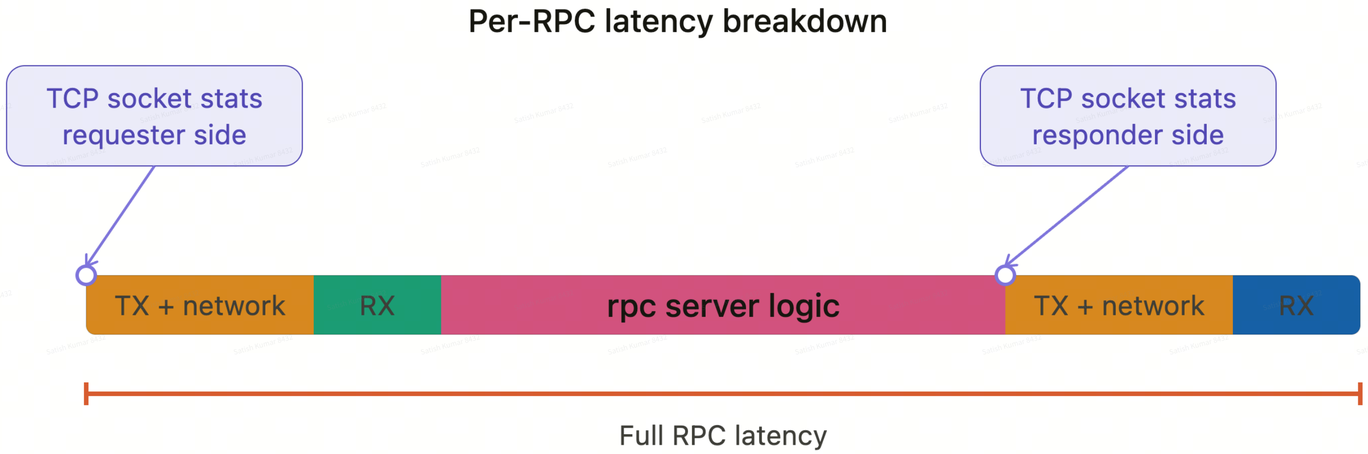
ByteDance

RPC Latency Breaker

Satish Kumar
System Techonology Engineering (STE)
ByteDance
July 13th-16th, 2026
NetDev 0x1A
satish.kumar@bytedance.com

RPC Latency Breaker



- For every RPC:
  - Full RPC latency at the system level
  - RX responder latency
  - RX requester latency
  - RPC logic latency
  - Network RTT which includes TX time
  - TCP socket stats at requester and responder

why do we care?

- RPC (remote procedure call) are fundamental communication in datacenter
  - Involve multiple servers: requester and responder
  - and multiple subsystems: network stack, scheduler, nic, network etc

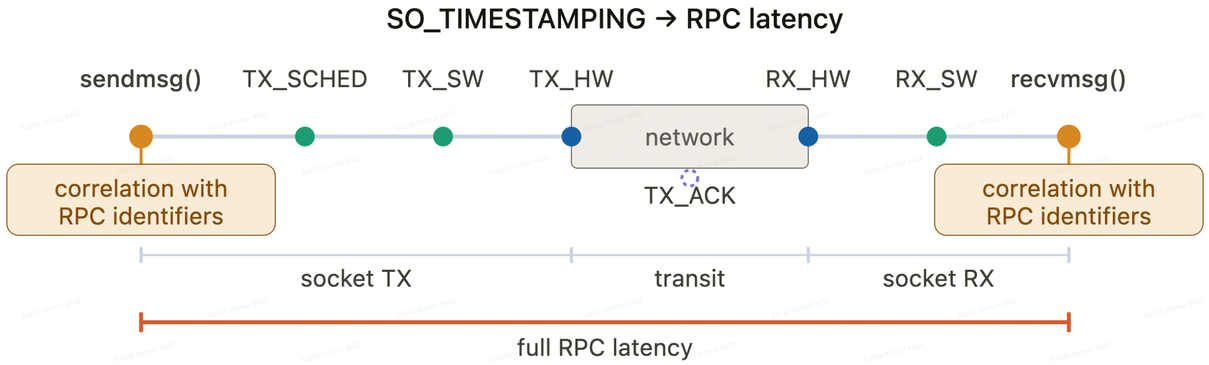
- diagnosing the issue is hard:
  - multiple components involved
  - sheer amount of rpc traffic

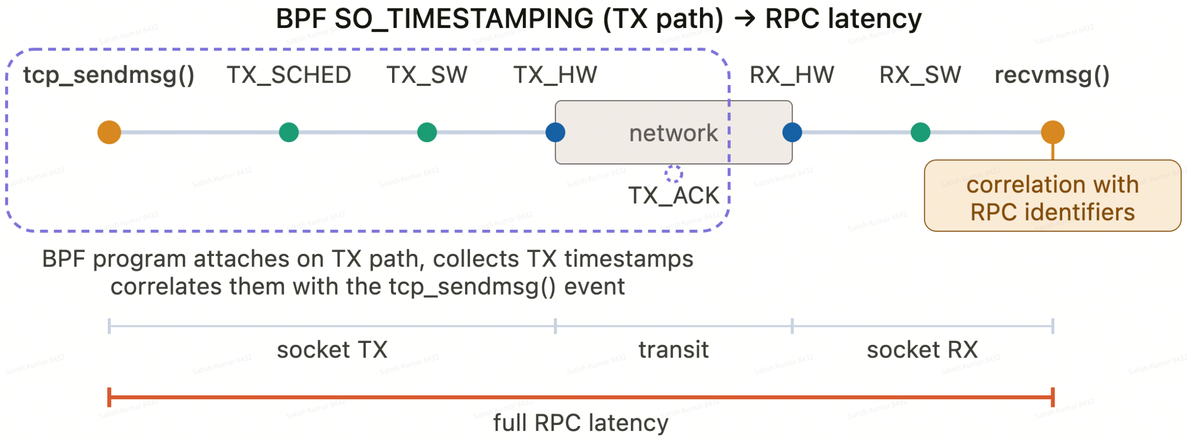
latency is the key metric

- idealy in the production
  - best average perforamnce
  - no long tail latencies
- isolate the source of the issue.
  - latency breakdown
  - tcp stats

socket timestamping

- so_timestamping:
  - application can get various RX and TX timestamps
  - correlate with sendmsg and recvmsg
  - further correlate sendmsg and recvmsg with rpc event
  - require application changes
- bpf so_timestamping:
  - more efficient tx timestamp retrieval
  - can correlate timestamps with sendmsg event
  - cannot create rpc picture
- Reference:
  - SO_TIMESTAMPING [Willem de Bruijn]
  - Future of SO_TIMESTAMPING [Jason Xing]

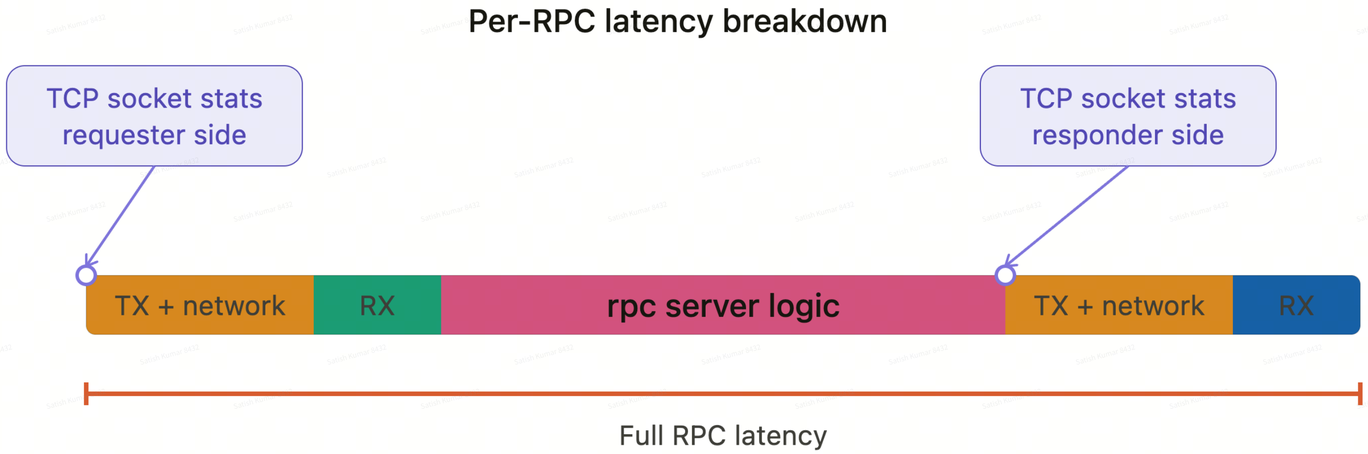




socket timestamping

- so_timestamping:
  - slow (tx timestamp collection is overhead)
  - application changes
- bpf so_timestamping:
  - fast but can not correlate with rpc

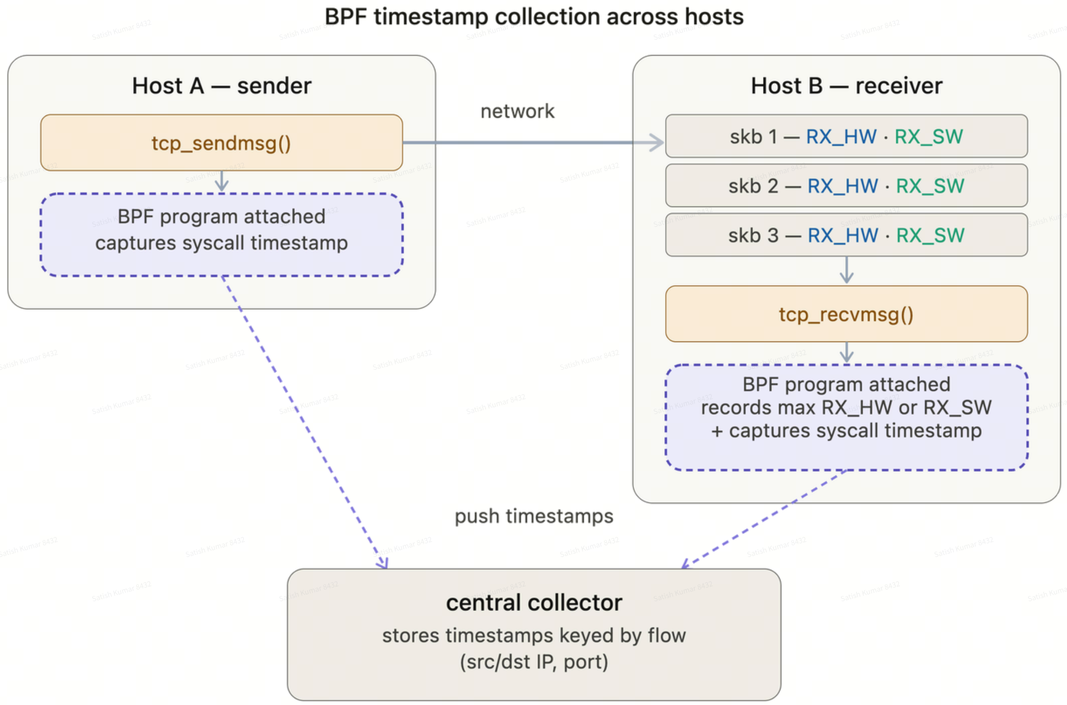
Non-intrusive, Efficient and Transparent



- rpc level correlation:
  - without application modifications -------> non-intrusive
  - efficient
    - easy to apply in production
    - cover all rpcs, no samping
  - works for most rpc types ----------------------> tranparent

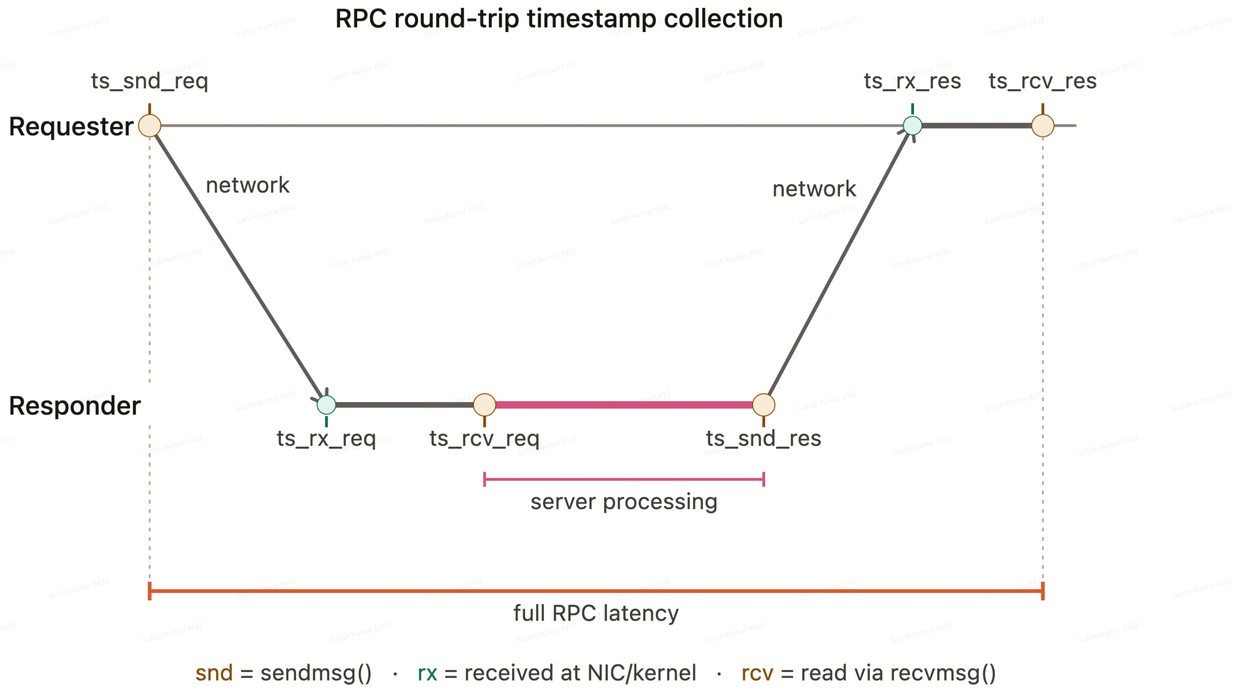
bpf program to collect timestamps

- bpf program
  - tcp_sendmsg level:
    - system call timestamp
  - tcp_recvmsg level:
    - system call timestamp
    - max of RX_HW/RX_SW so_timestamp from processed skbs

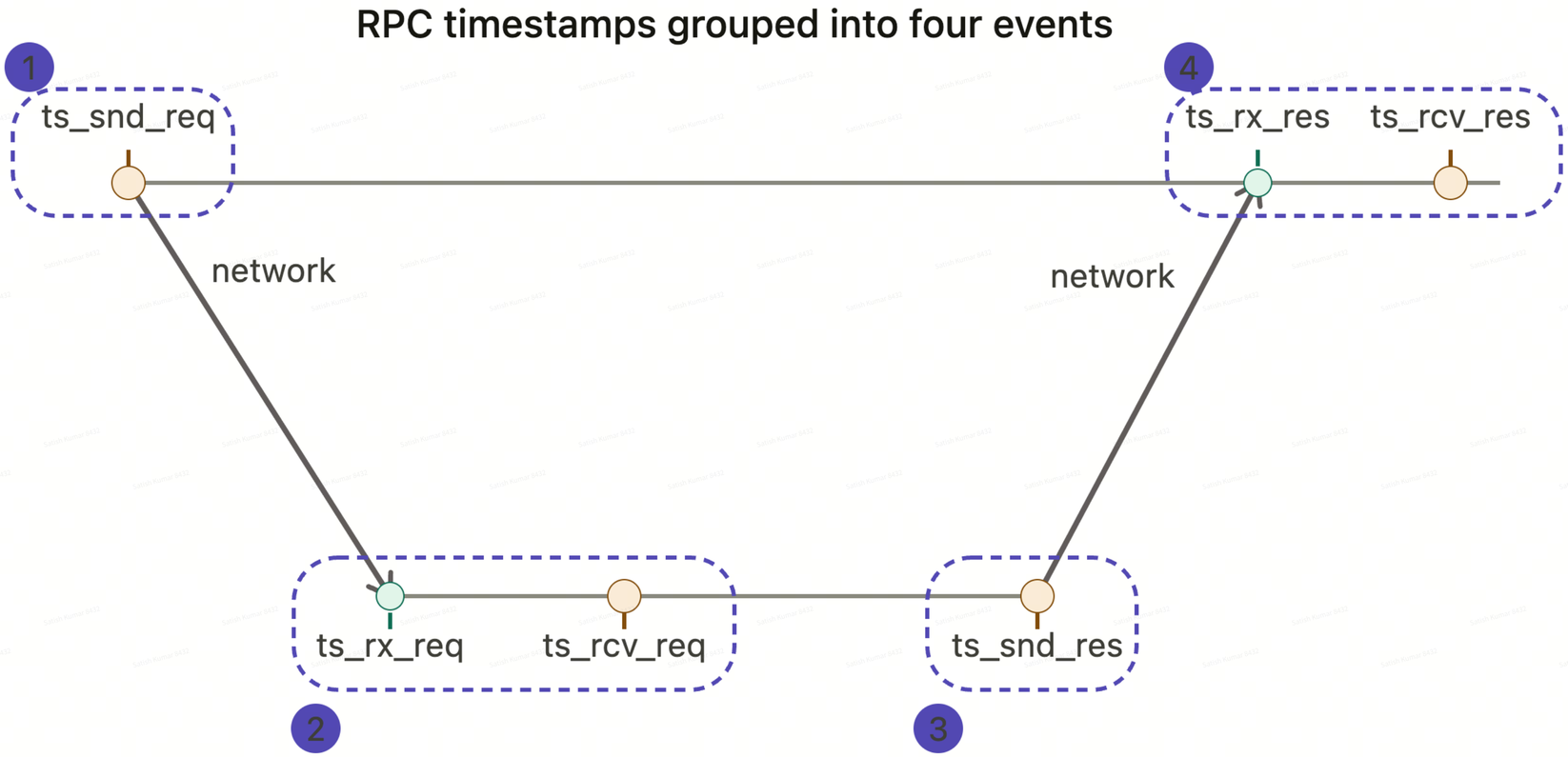


rpc timestamps

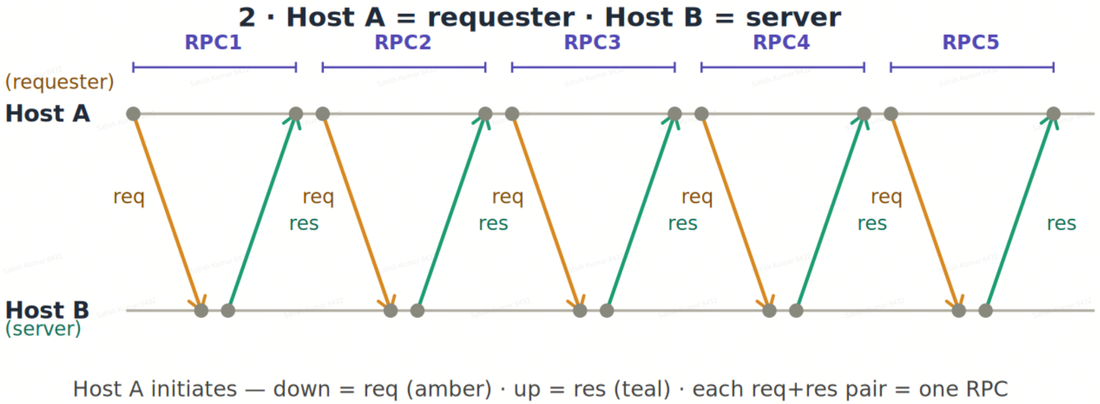
- total six timestamps per rpc
- rpc request:
  - ts_snd_req --> ts_rx_req --> ts_rcv_req
- rpc processing
- rpc response:
  - ts_snd_res --> ts_rx_res --> ts_rcv_res



independent events, how to correlate?



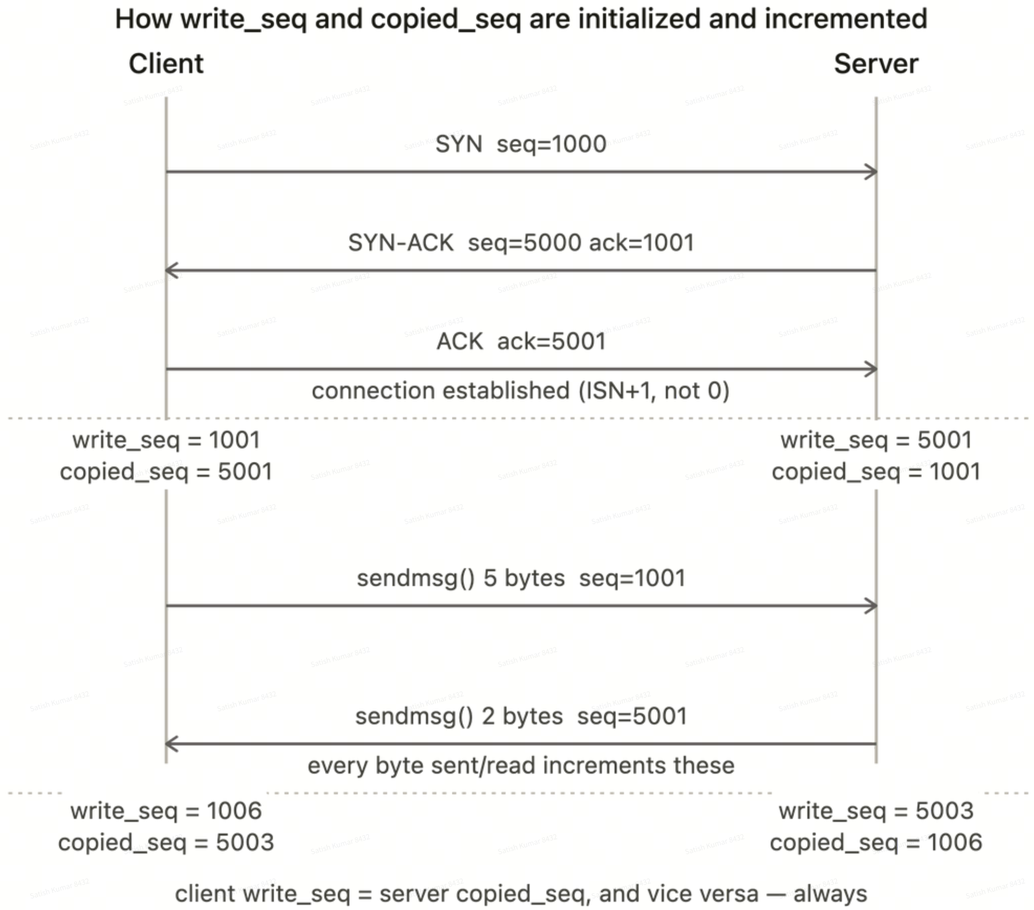
rpc's ping pong pattern for correlation



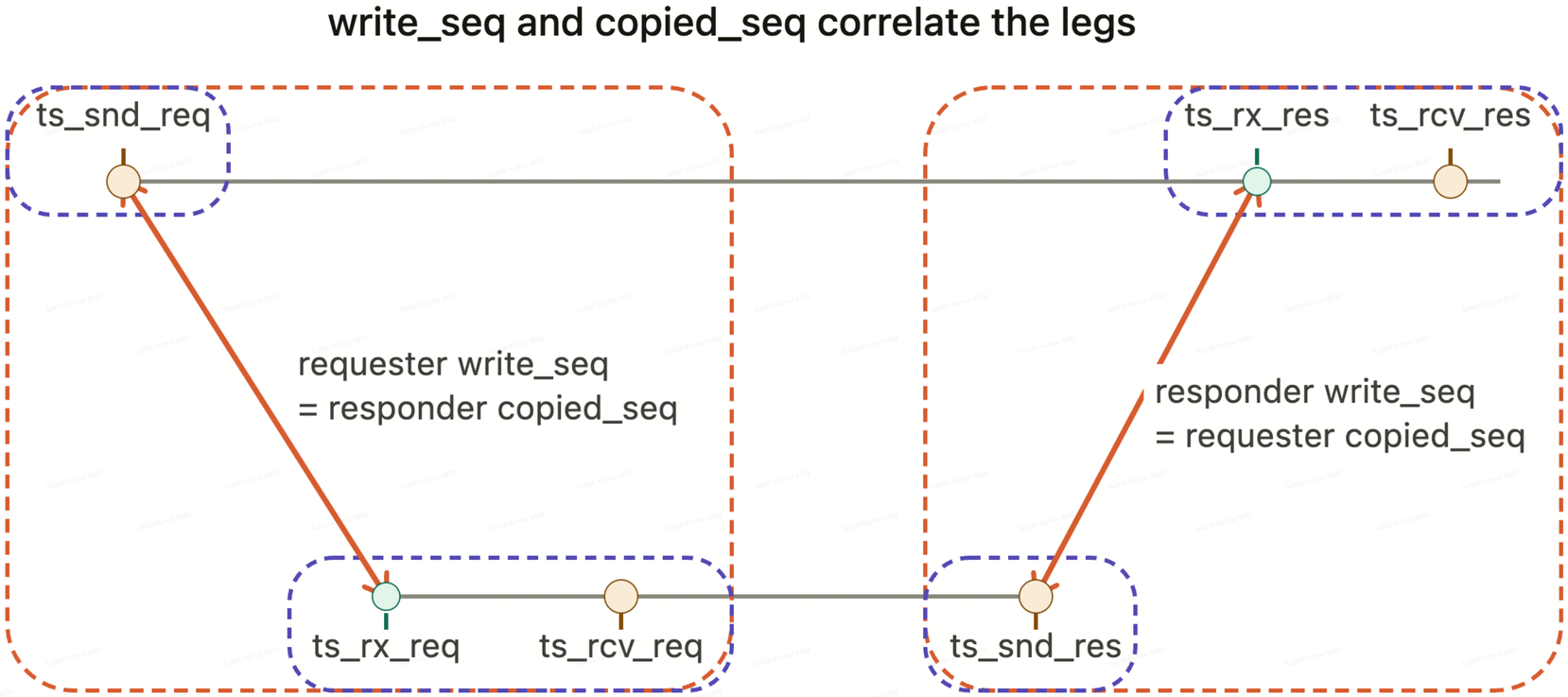
- ping: rpc request
- pong: rpc response
- its not a stream of messages towards one direction
- both side alternate between request and reponse

correlation by socket sequence number

- socket sequence numbers
  - write_seq: incremented for every tcp_sendmsg byte sent by the application
  - copied_seq: incremented for every tcp_recvmsg byte receiven by the application
- increased when app finishes the system call
- client and server both maintain them
- and initialized during connect
  - client write_seq = server copied_seq
  - server write_seq = client copied_seq



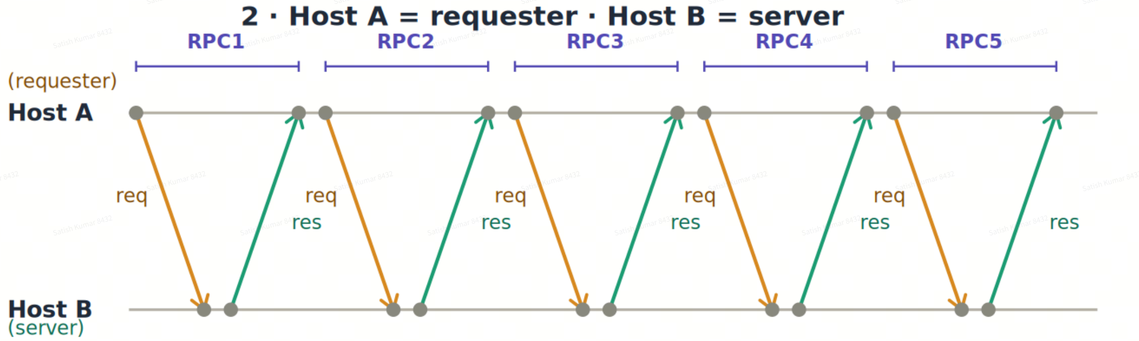
correlate sendmsg to recvmsg events

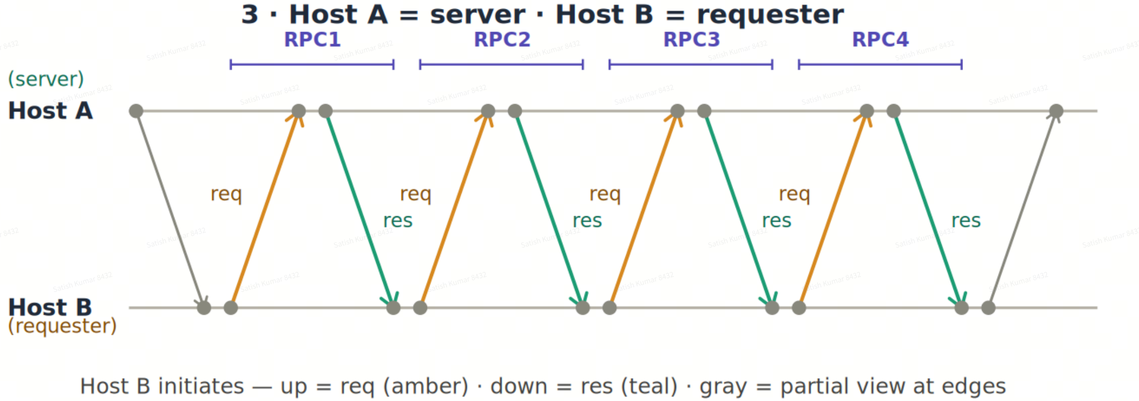


- rpc communication is sequential not stream
  - rpc request -----> rpc response
- record these sequence numbers along with timestamps

correlate request and response messages

- its just about identifying who is the rpc requester
- given ping pong behaviour, req and res can be combined to form the rpc event



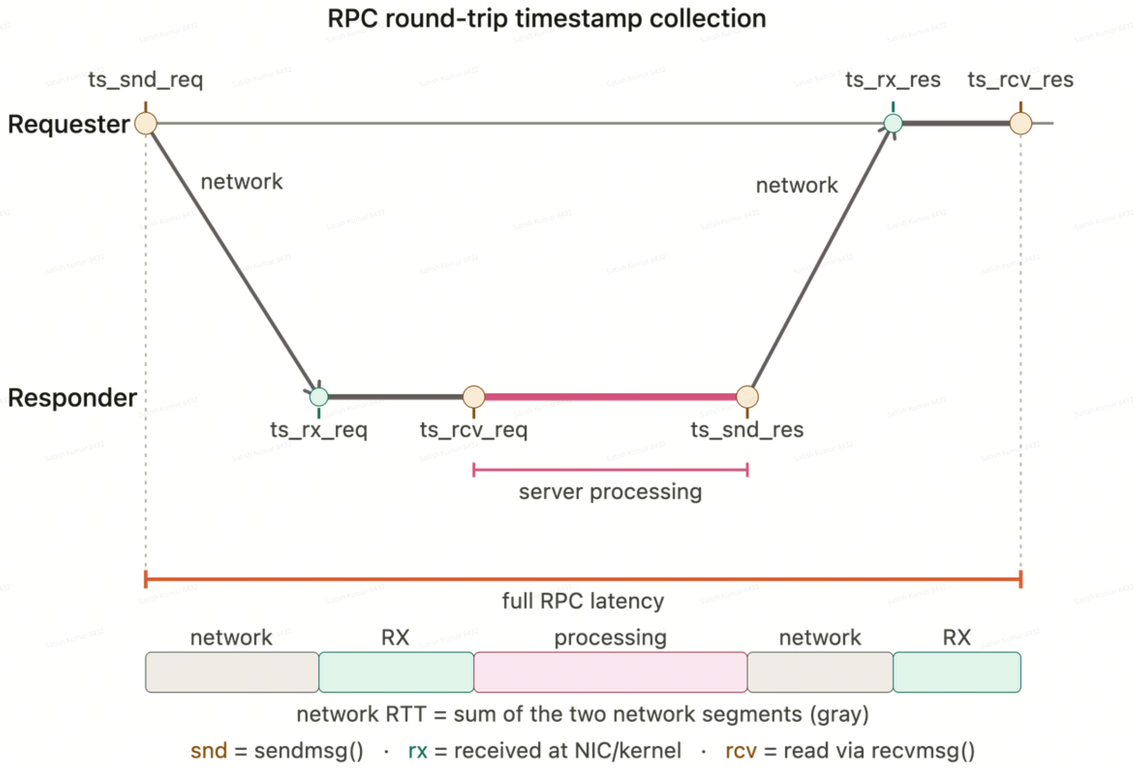


classifying requester and responder

- we know upfront, which app is rpc requester and communicating downstream to which app

complete breakdown

- rpc latency at system level
  - ts_rcv_res - ts_snd_req
- rpc server processing
  - ts_snd_res - ts_rcv_req
- rx latencies
  - time taken since complete data arrives and application reads
  - ts_rcv_req - ts_rx_req
  - ts_rcv_res - ts_rx_res
- network rtt including tx time
  - subtract all latencies above from the total latency
  - includes the physical network delays and tx stacks



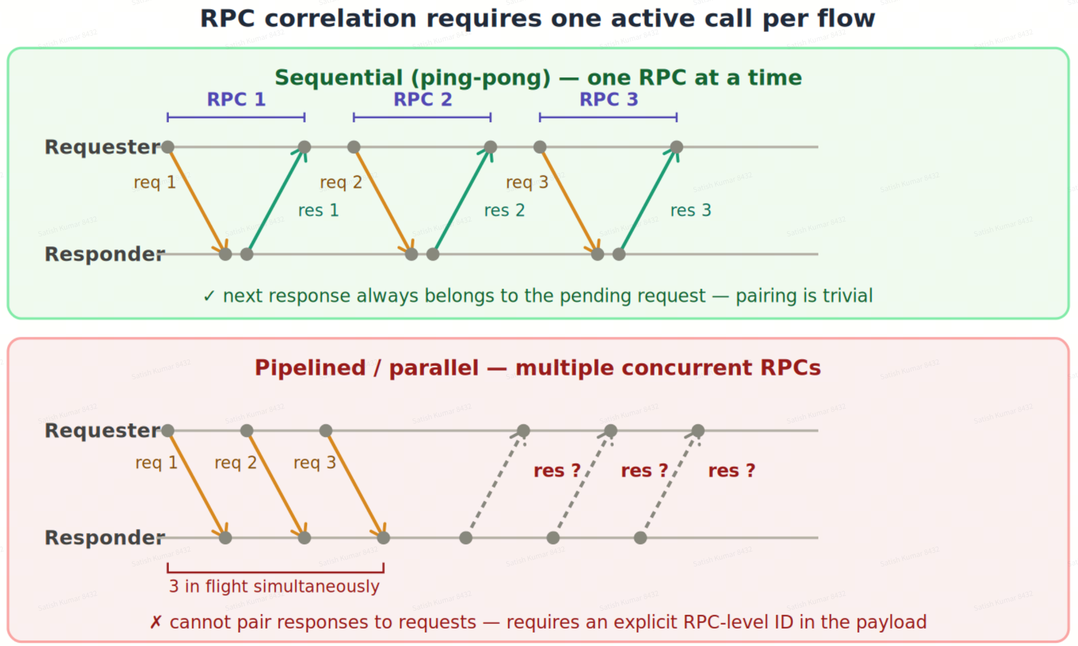
points to mention

- network rtt include tx stack
  - we could also do tx so_timestamps using bpf interface
  - we decide not to do in general due the need of packet level tracing
  - usually tx is straight path to the nic in the app context

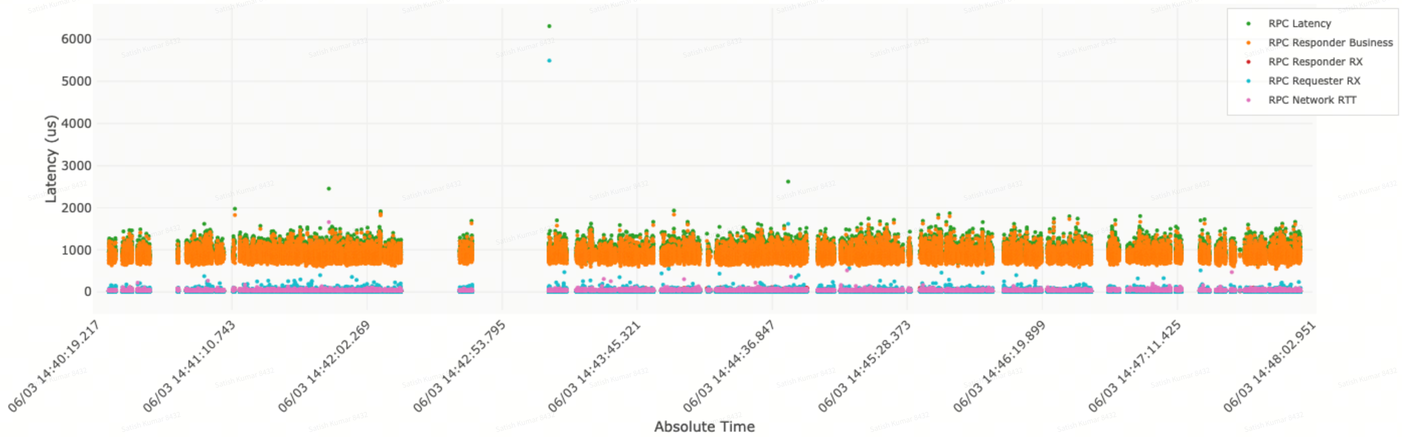
- rpc messages are just not single sendmsg or recvmsg
  - require intelligent aggregation

ping-pong condition

- rpc correlation requires one active call per flow
  - wont work for multiple concurrent rpc
- how common is ping pong pattern?
  - thrift protocol over tcp socket is strictly sequential by default
    - maintain connection pool
  - http/1.1 REST is also sequential and maintain connection pool
    - heavily used for internal datacenter traffic
  - grpc, http/2 REST send multiple rpc over single connection
    - given scale out container services, ping pong pattern is still common

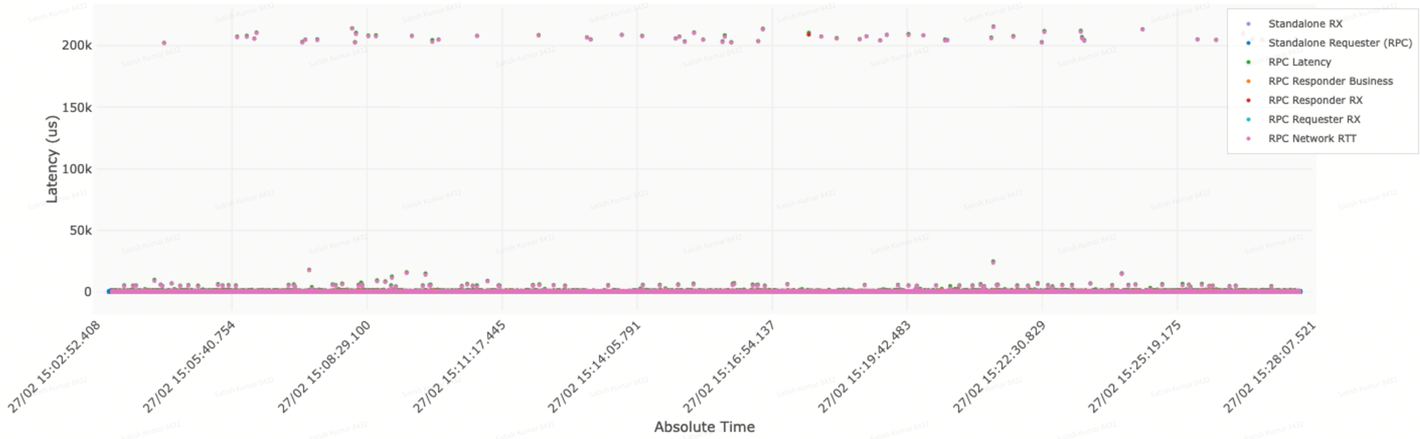


case study 1



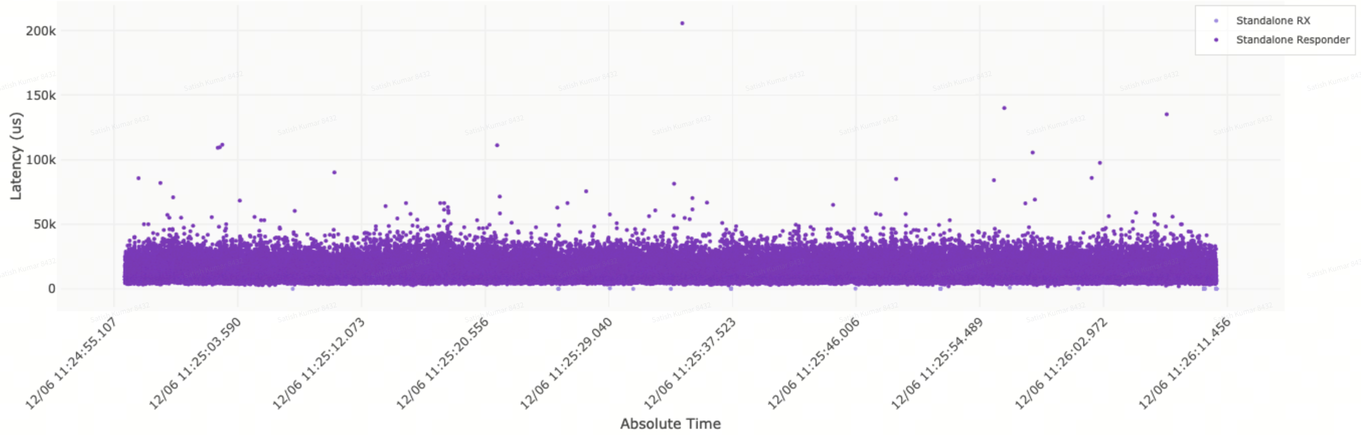
- increased requester-rx latency
  - either issue with softirq processing (rx stack) or application scheduling delay
- bcc runqlat tool and atop confirmed the scheduling delays

case study 2



- frequent jump of 200ms in network rtt
  - issue with physical network, tx stack or tcp flow
- check tcp flow stats exact at these points
  - increase in tail loss probes

case study 3



- tracing one end provide details on its communication to all other ends
- checking tcp socket stats, revelaed two problem:
  - packet losses, increase in retrans
  - high tcp rtt to some machines, hardware clock issues

Thank you
Q&A