Socket Launcher

Rapid, Lossless, and Transparent TCP Migration

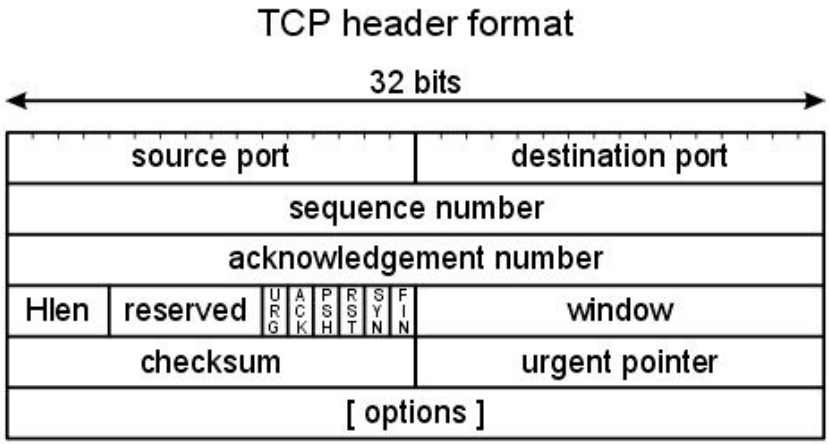
Master's Thesis Defense || Isaac Pedisich || July 17, 2019

Background and Motivation

- A (very brief) introduction to TCP
- Existing TCP migration frameworks
- Project Goals

Background and Motivation: TCP

- A stream of messages between two endpoints
- Provides higher-layer guarantees
  - Correctness
  - Reliability
  - Ordering
- Peers maintain consistent connection state
  - Transmitted via the TCP header
    - Sequence numbers
    - Acknowledgement numbers (ACK)
- Maintenance of matching states makes multi-server contributions tricky



Background and Motivation: Existing Work

- Migratory TCP (M-TCP)
  - New protocol built on top of TCP
  - Client and server must both agree to use of the protocol, or else it defaults to normal TCP
- Reliable sockets (ROCKS)
  - Must be present at both ends of the connection
- MSOCKS
  - Focuses on movement of a single host across diverse address ranges
- SockMi
  - More portable, but lacks any evaluation of transfer speed or reliability

Project Goals

Socket Launcher aims to achieve the following goals:

1. **Client-side transparency**
   - No indication that server-side migration has occurred
2. **Zero packet loss**
   - TCP works, but is hindered, when packets are dropped
   - Migration should not induce the dropping of any packets
3. **Rapid migration**
   - Minimal overhead to throughput and latency
4. **Linux kernel utilization**
   - Minimal effort to use the framework
   - Should not require a custom TCP stack or modifications to the linux kernel

Tools and Methods

- What's involved in transferring
  - Diversion of packets (proxy)
  - Server-side framework enabling migration
- What needs to be transferred?
  - Port numbers
  - Seq/Ack values
  - Buffers
  - Application state? (tabled for now)
- Building blocks:
  - Packet rewriting and forwarding (ebpf)
    - TC and XDP
    - BCC
    - [[I thought this would be the main tool]]
  - TCP_REPAIR

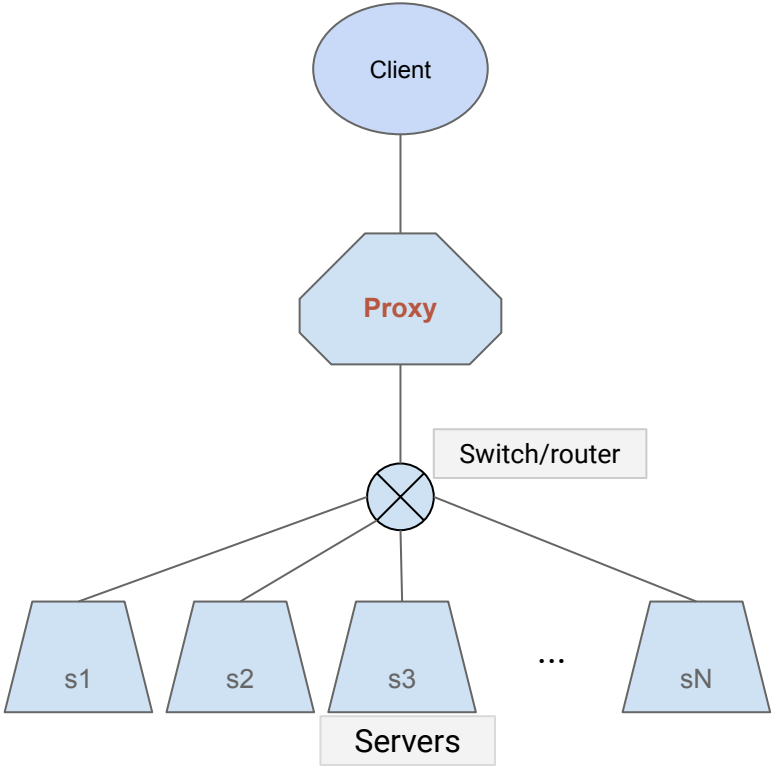
Migration Architecture

- What are the distinct components of Socket Launcher?
- What basic steps does TCP migration consist of?
- What specific needs does TCP migration need to fit?

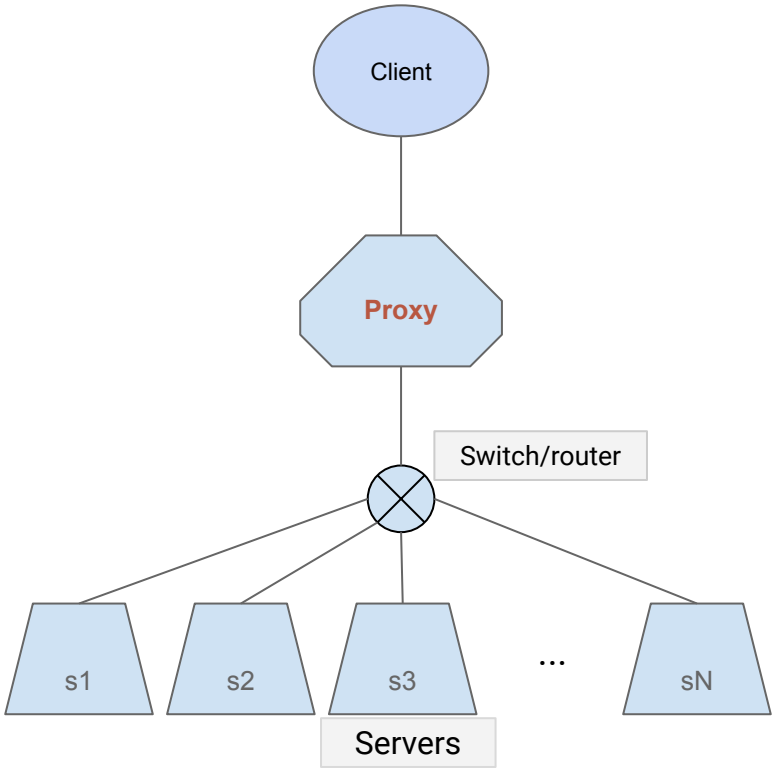
Migration Architecture: Physical Components

- Proxy / Load Balancer
  - Packets appear to have single source IP
    - Network Address Translation (NAT)
  - By default: flows are balanced across nodes
  - Flows are reroutable on-the-fly

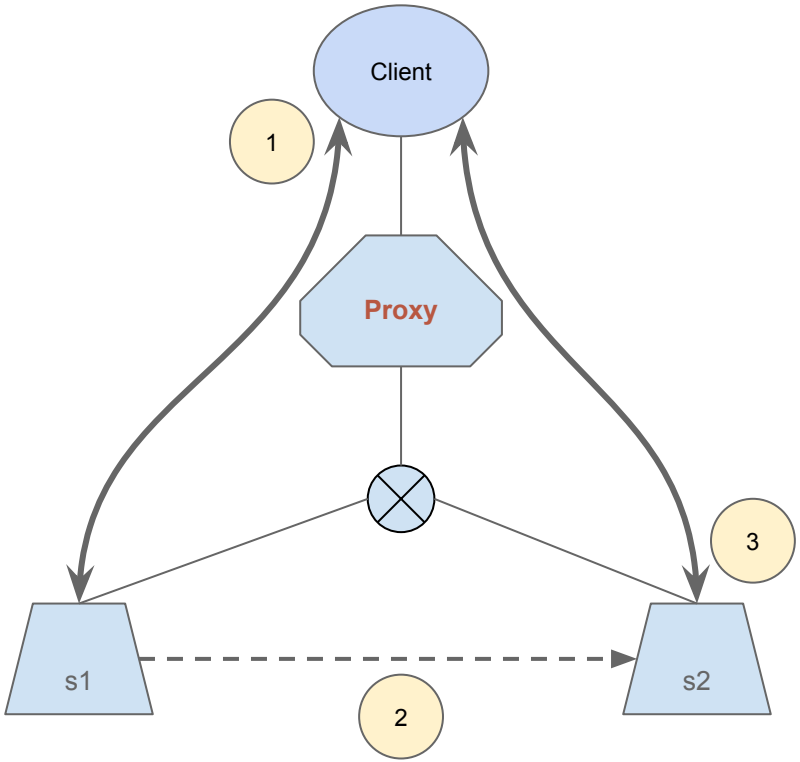
- Servers
  - Accepted and transferred sockets behave identically
  - Arbitrary number of possible endpoints
  - Utilizes direct communication with peer servers



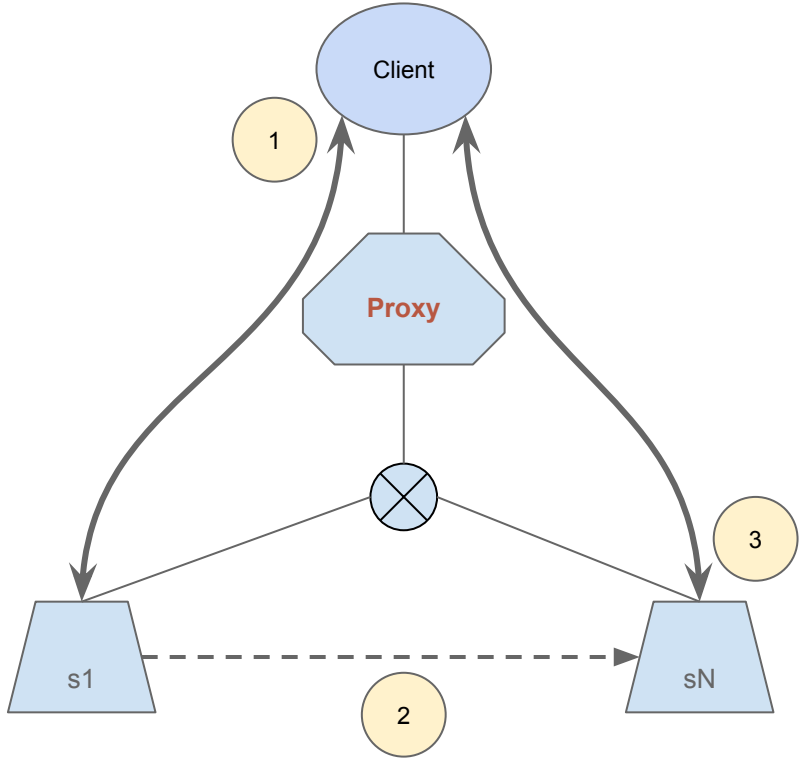
Migration Architecture: Transfer Process



Migration Architecture: Transfer Process



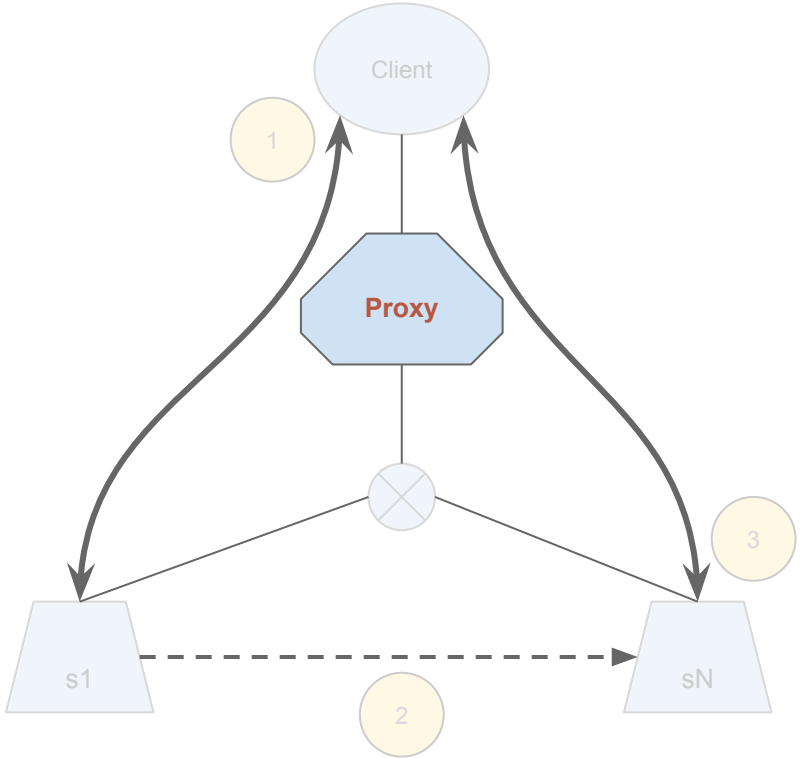
Migration Architecture: Transfer Process



Migration Architecture: Requirements

Proxy requires fast packet rewriting and forwarding

- Source/destination IP address must be changed
- IP/TCP Checksums must be updated
- Must be capable of learning and forgetting flows
- Must have interface reachable from user program



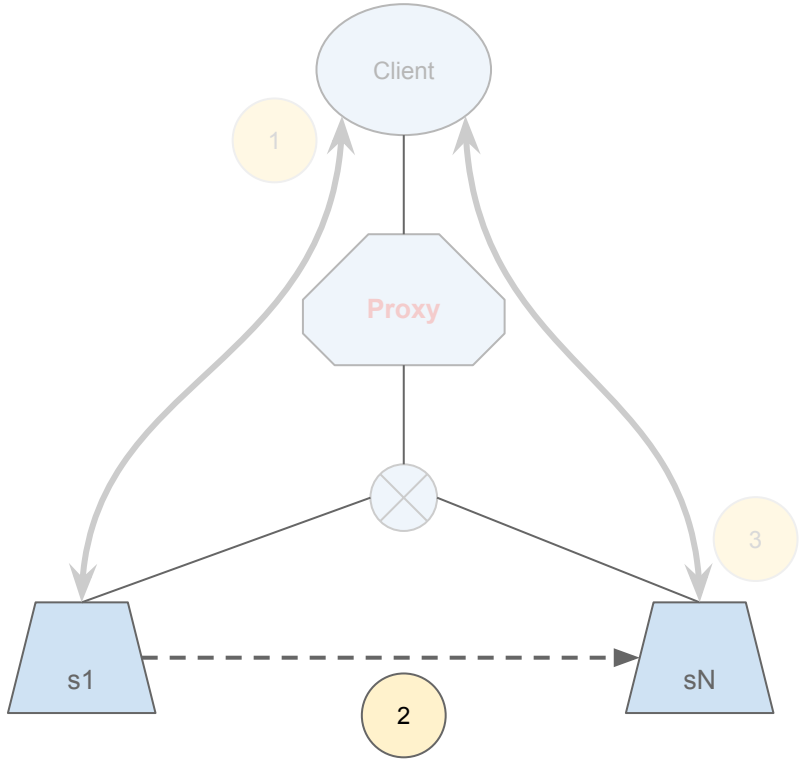
Migration Architecture: Requirements

Peer servers require state transfer:

- Ports and IP addresses
- Sequence and acknowledgement numbers
- Contents of existing read/write buffers

Also useful:

- Corresponding application state
  - Not fully realized in current implementation



Tools

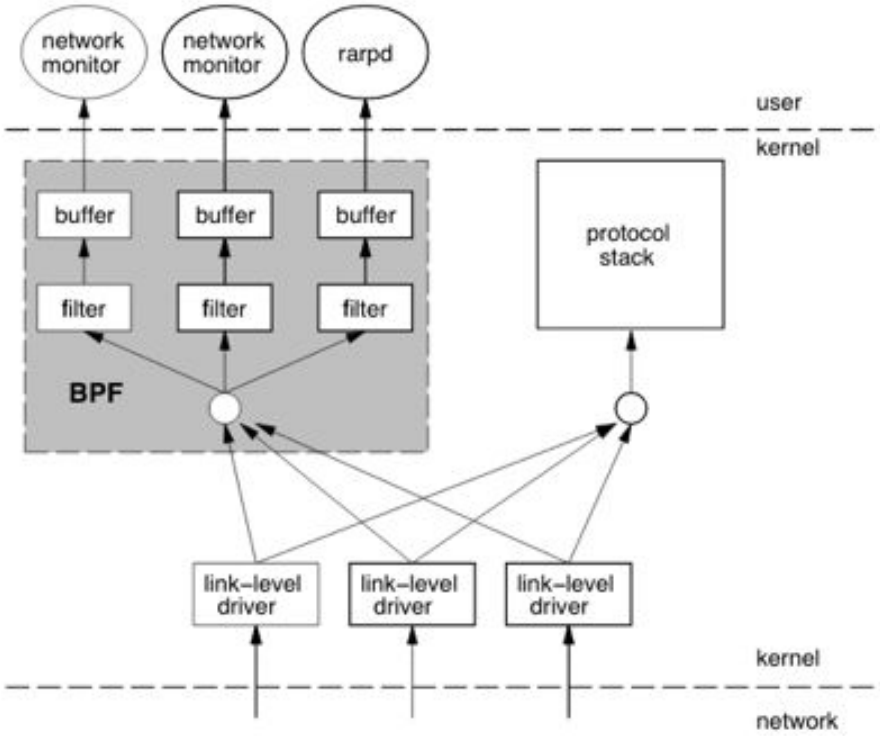
- What tools and libraries are neccessary to satisfy the requirements?
  - **Packet manipulation:**
    - Extended Berkeley Packet Filter (eBPF)
    - BPF Compiler Collection (BCC)
  - **TCP state transfer:**
    - TCP_REPAIR

Tools: eBPF

Brief History:

- 1992: BSD Packet filter (BPF) is proposed
  - Packet capture performed in kernel-space
  - Configurable from user-space
  - Designed for fast packet filtering

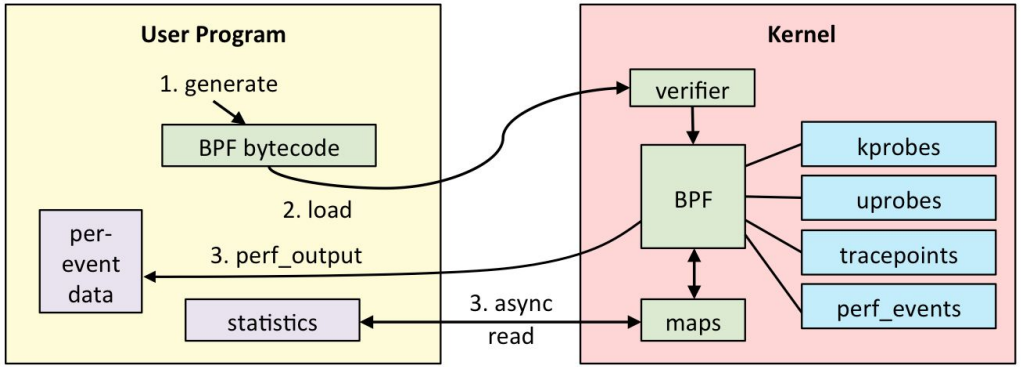
- 1997: Integrated into Linux kernel
  - PCAP filter syntax
  - Tcpdump
  - Sidesteps the kernel protocol stack for efficient packet filtering
    - Runs in a special-purpose virtual machine
    - Specific memory set aside for the packet
    - Provides isolation and flexibility



Tools: eBPF

Brief History:

- 2013: Extended BPF is proposed
  - Makes use of a similar virtual-machine architecture
  - Expansion of the available instruction set for “arbitrary” program execution
    - (subject to many constraints)
  - Many more hooks for execution besides packet receipt
  - Shared memory between user-space and eBPF code

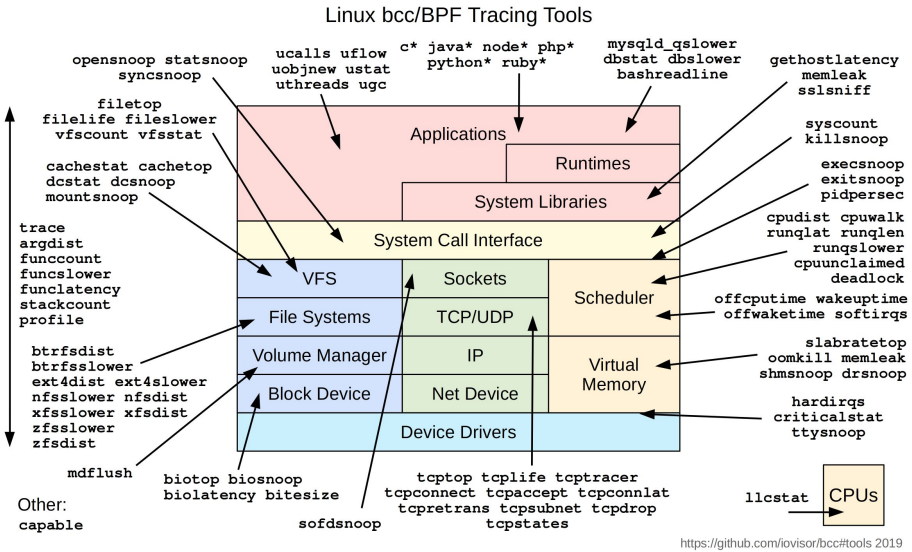


http://www.brendangregg.com/ebpf.html

Tools: eBPF

- eBPF programs can be a pain to write
  - BPF Compiler Collection (BCC) to the rescue!
    - Easier to write
    - Easy to load, interface with shared memory
    - Runnable via a python frontend
      - (more on that later)

- Multiple types of eBPF programs can be loaded
  - Two types allow for packet modification:
    - eXpress Data Path (XDP)
      - Multiple possible points of execution
        - In kernel
        - In device driver
        - On NIC
    - Traffic Control (TC)
    - Run in kernel, prior to TCP stack
  - Can manipulate, drop, or forward packets



Tools: eBPF (an aside)

- Initial version of migration framework was done entirely in eBPF
- Packets were rewritten to meet SEQ/ACK criteria
- I was then informed of...

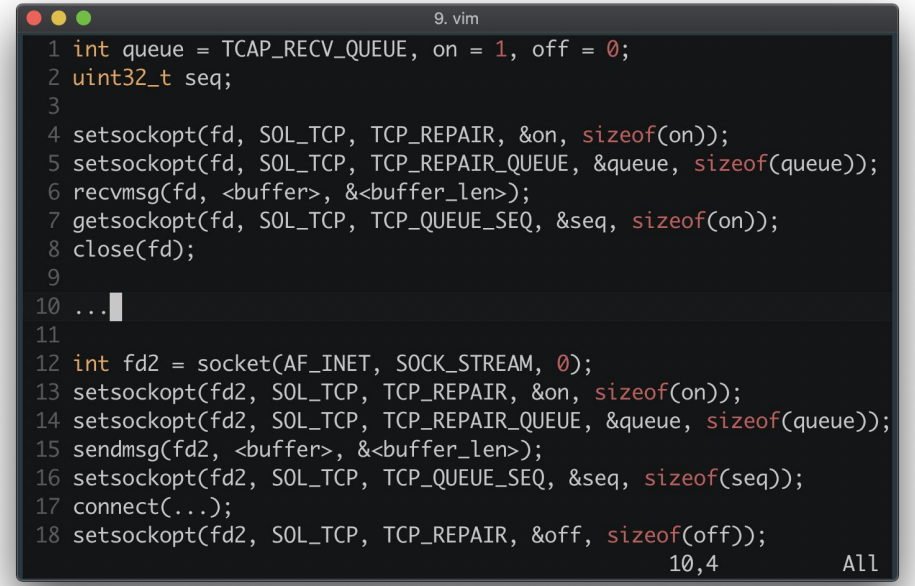
Tools: TCP REPAIR

- Direct setting and retrieval of TCP socket parameters
  - Get/set SEQ, ACK, buffer contents
  - Limited documentation
    - (some of which is incorrect)

Tools: TCP REPAIR

- Direct setting and retrieval of TCP socket parameters
  - Get/set SEQ, ACK, buffer contents
  - Limited documentation
    - (some of which is incorrect)

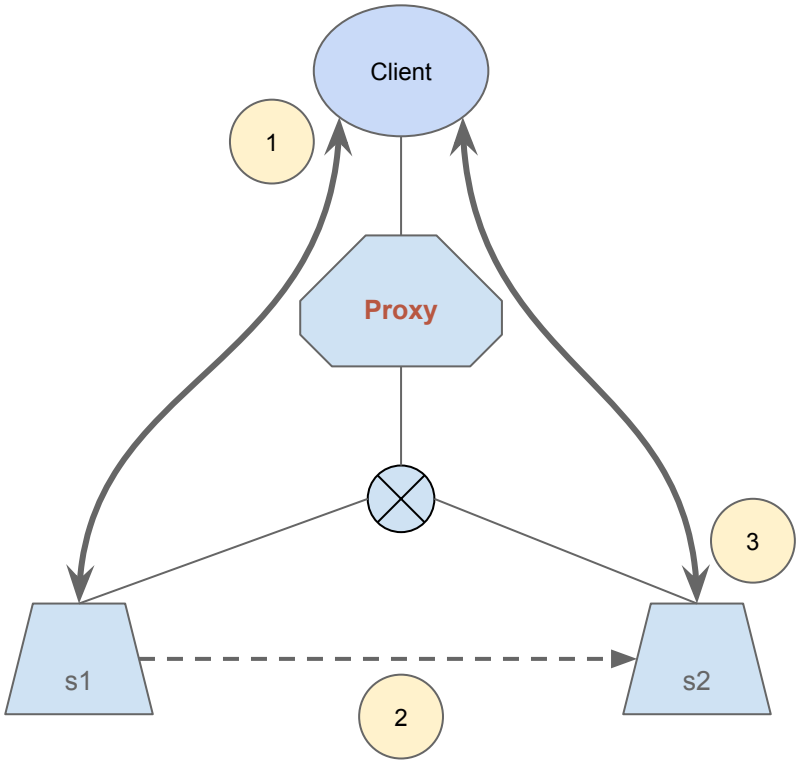
- Socket is placed into TCP_REPAIR mode
  - *recvmsg()* peeks into TCP buffer
  - *close()* removes the socket without shutting down connection
  - *sendmsg()* sets TCP buffer
  - connect() opens socket without performing handshake
  - Turning off TCP_REPAIR causes a window-probe to be sent (will come back to this)



Migration Protocol

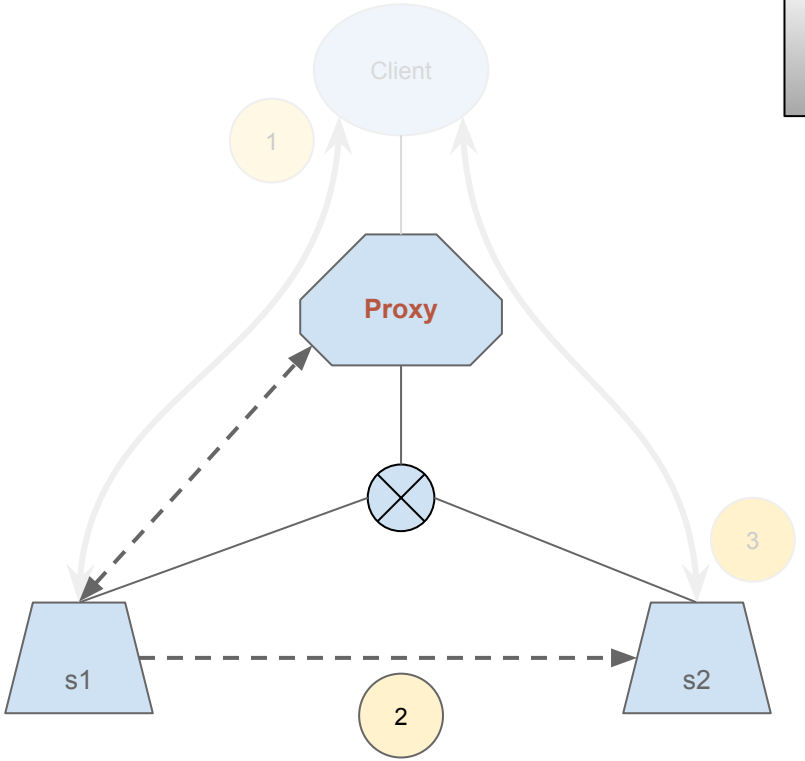
- What messages are passed to perform TCP migration?
- What steps are taken to ensure lossless transfer?
- What eBPF programs were necessary to perform those steps?

Migration Protocol: Messages



Migration Protocol: Messages

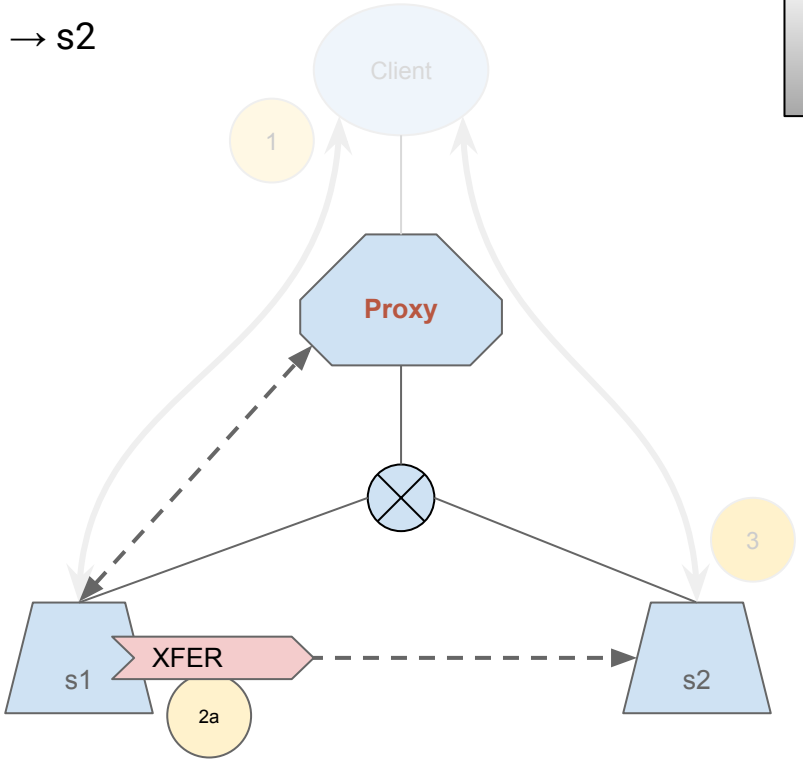
Migration is performed
In four steps



Migration Protocol: Messages

1. State is transferred: s1 → s2

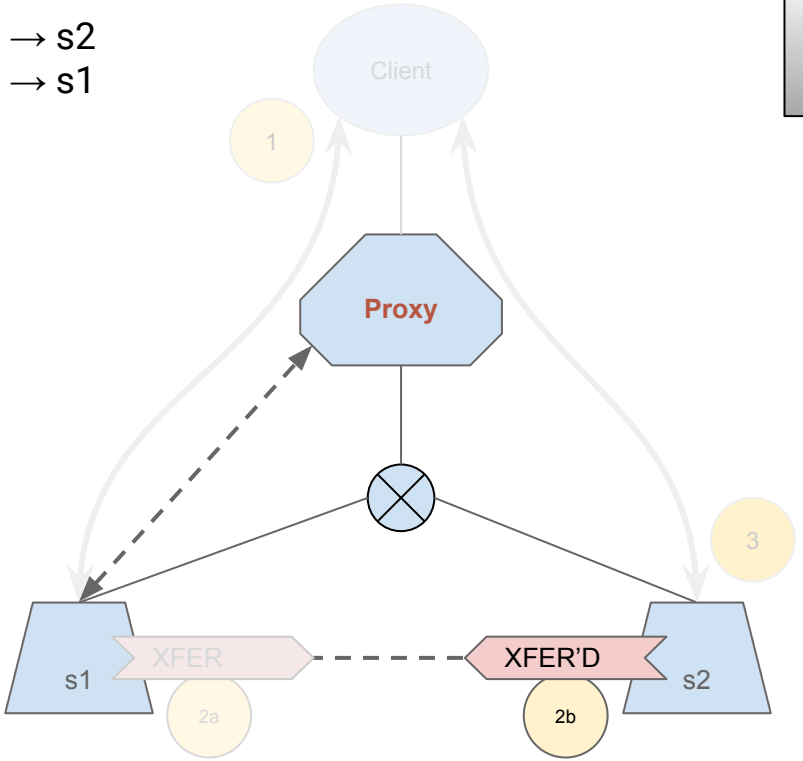
Migration is performed
In four steps



Migration Protocol: Messages

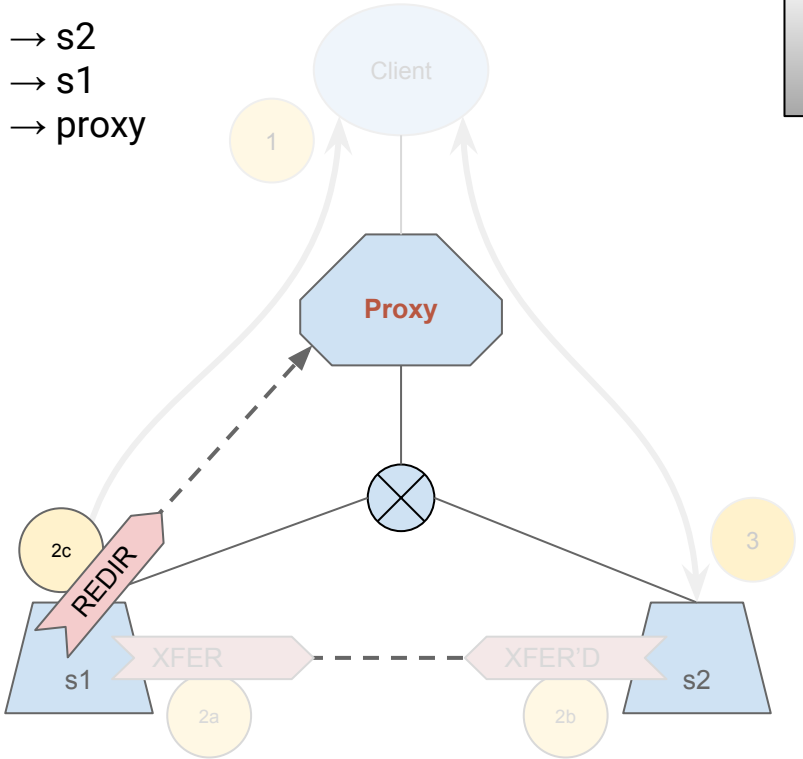
1. State is transferred: s1 → s2
2. Transfer is confirmed: s2 → s1

Migration is performed
In four steps



Migration Protocol: Messages

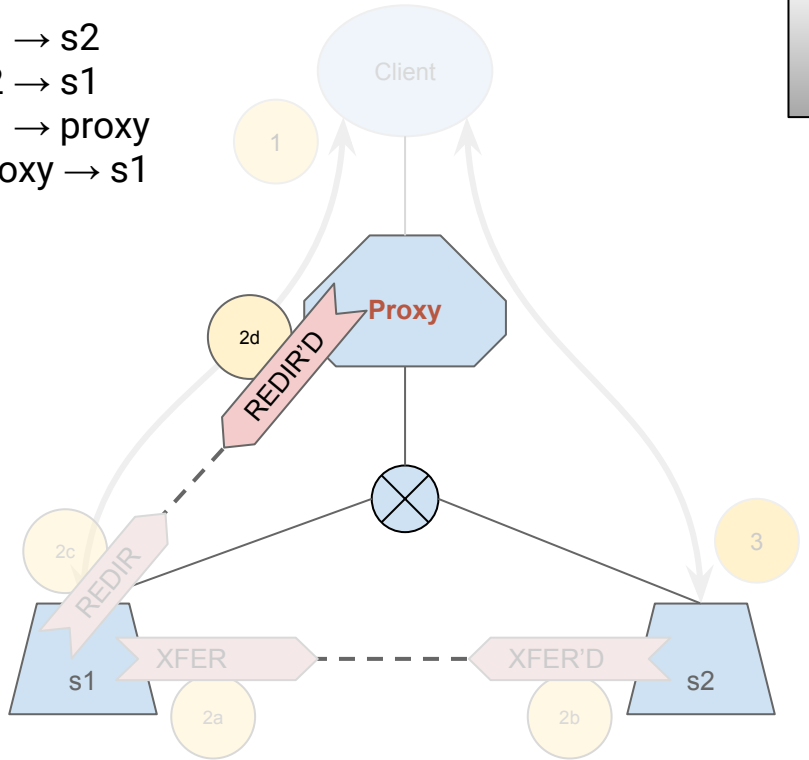
1. State is transferred: s1 → s2
2. Transfer is confirmed: s2 → s1
3. Redirect is requested: s1 → proxy



Migration is performed
In four steps

Migration Protocol: Messages

1. State is transferred: s1 → s2
2. Transfer is confirmed: s2 → s1
3. Redirect is requested: s1 → proxy
4. Redirect is confirmed: proxy → s1

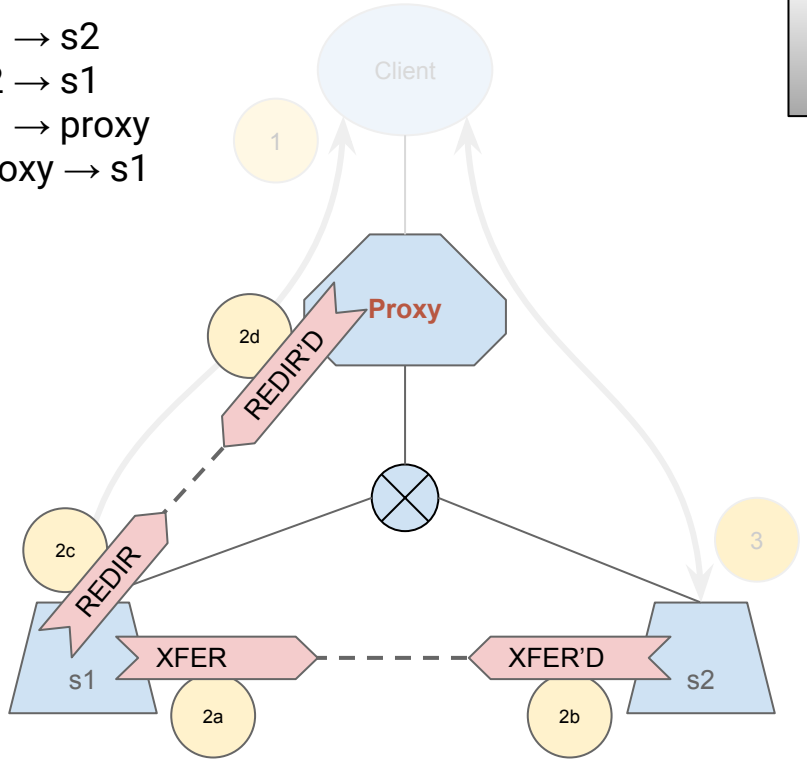


Migration is performed
In four steps

Migration Protocol: Messages

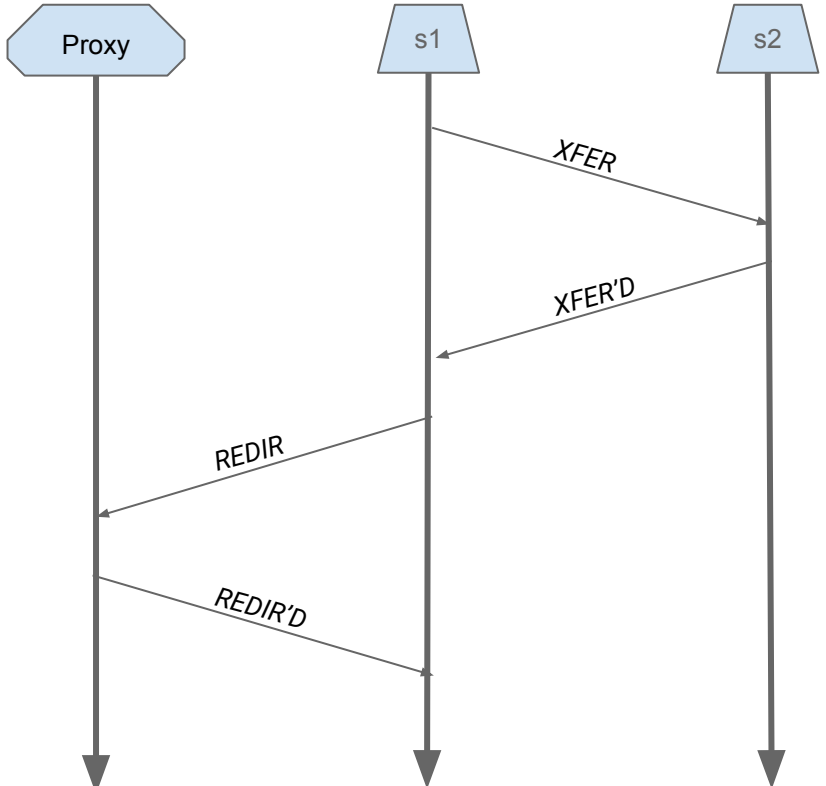
1. State is transferred: s1 → s2
2. Transfer is confirmed: s2 → s1
3. Redirect is requested: s1 → proxy
4. Redirect is confirmed: proxy → s1

Migration is performed
In four steps



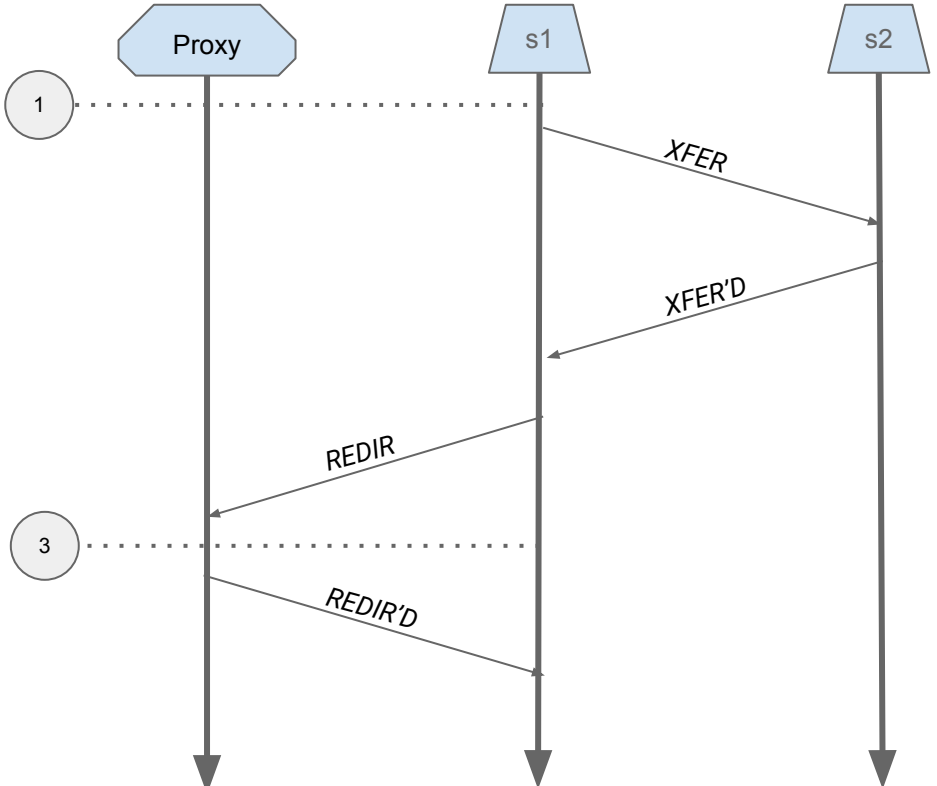
Migration Protocol: Messages

1. State is transferred: s1 → s2
2. Transfer is confirmed: s2 → s1
3. Redirect is requested: s1 → proxy
4. Redirect is confirmed: proxy → s1



Migration Protocol: Avoiding Loss

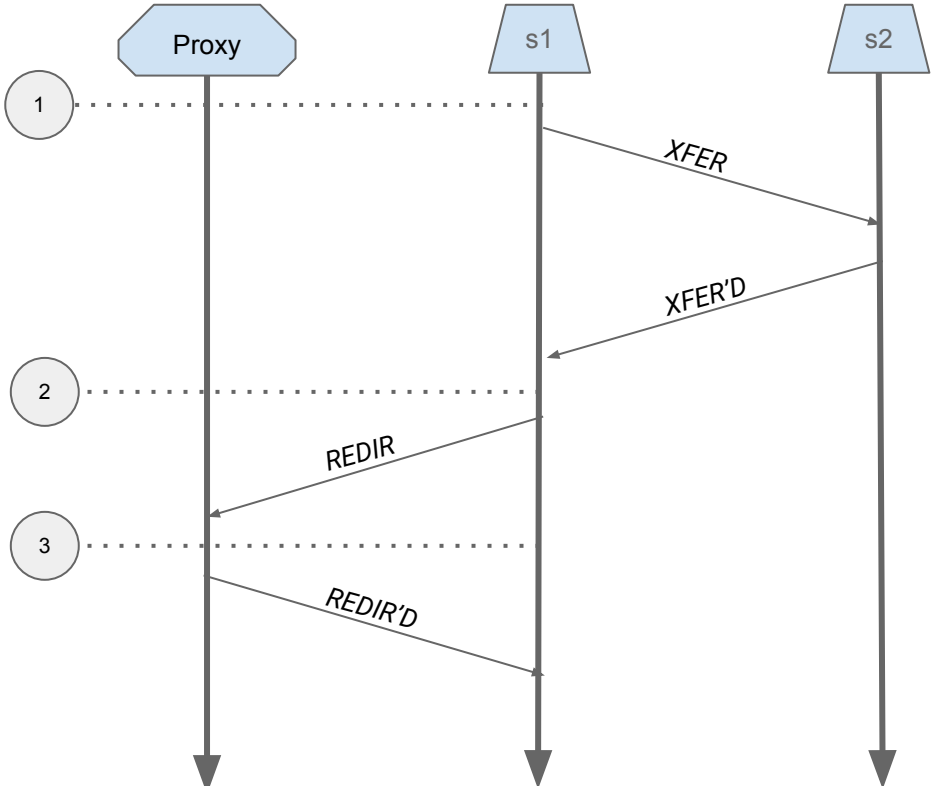
- Without extra consideration, packets arriving between (1) and (3) will be
  - Dropped
    - Best case, TCP will just retry
  - Terminated
    - Non-open ports will respond with RST



Migration Protocol: Avoiding Loss

The problematic window consists of two distinct epochs:

- (1) → (2)
  - s2 is not yet ready to receive
- (2) → (3)
  - s2 is ready to receive



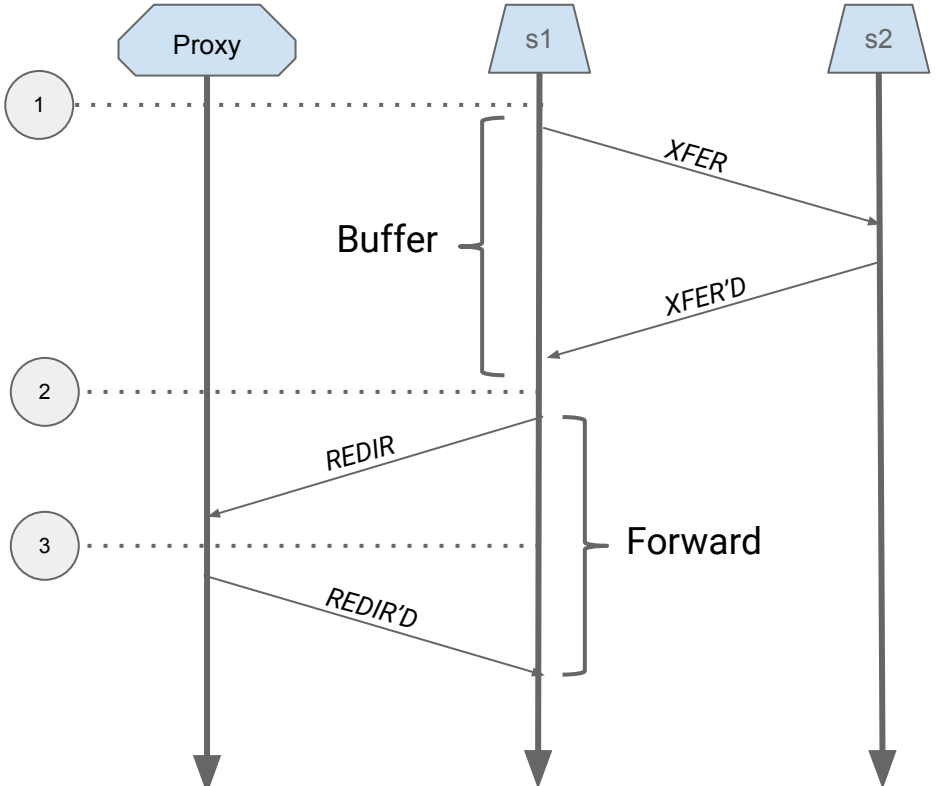
Migration Protocol: Avoiding Loss

The problematic window consists of two distinct epochs:

- (1) → (2)
  - s2 is not yet ready to receive
- (2) → (3)
  - s2 is ready to receive

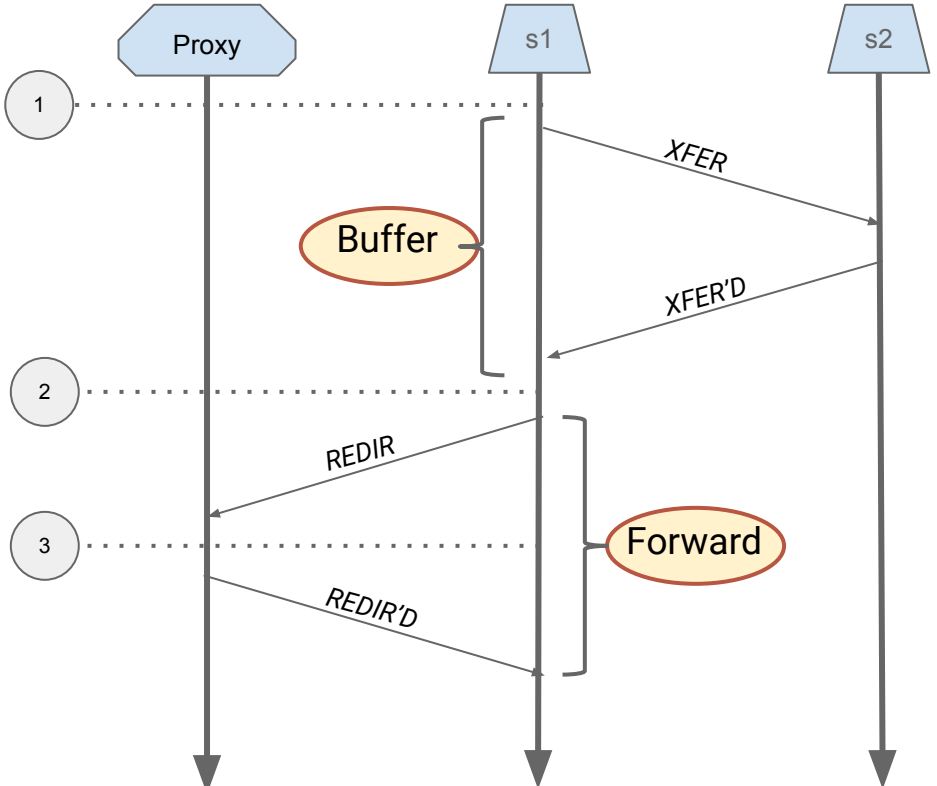
Solution:

- (1) → (2)
  - Buffer incoming packets
- (2)
  - Empty buffer towards s2
- (2) → (3)
  - Forward incoming packets to s2



Migration Protocol: eBPF Tools

Buffering and forwarding can be accomplished with eBPF, but not without a struggle

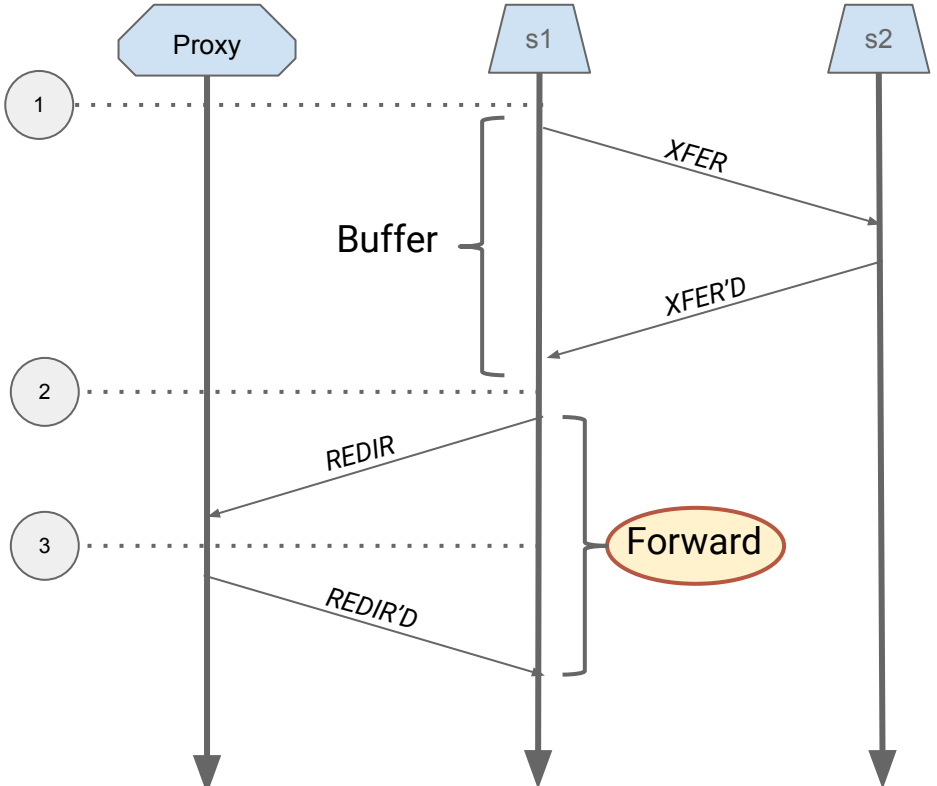


Migration Protocol: eBPF Tools

Buffering and forwarding can be accomplished with eBPF, but not without a struggle

Forwarding is the simpler of the two

- Out-of-the-box, and fast, with XDP

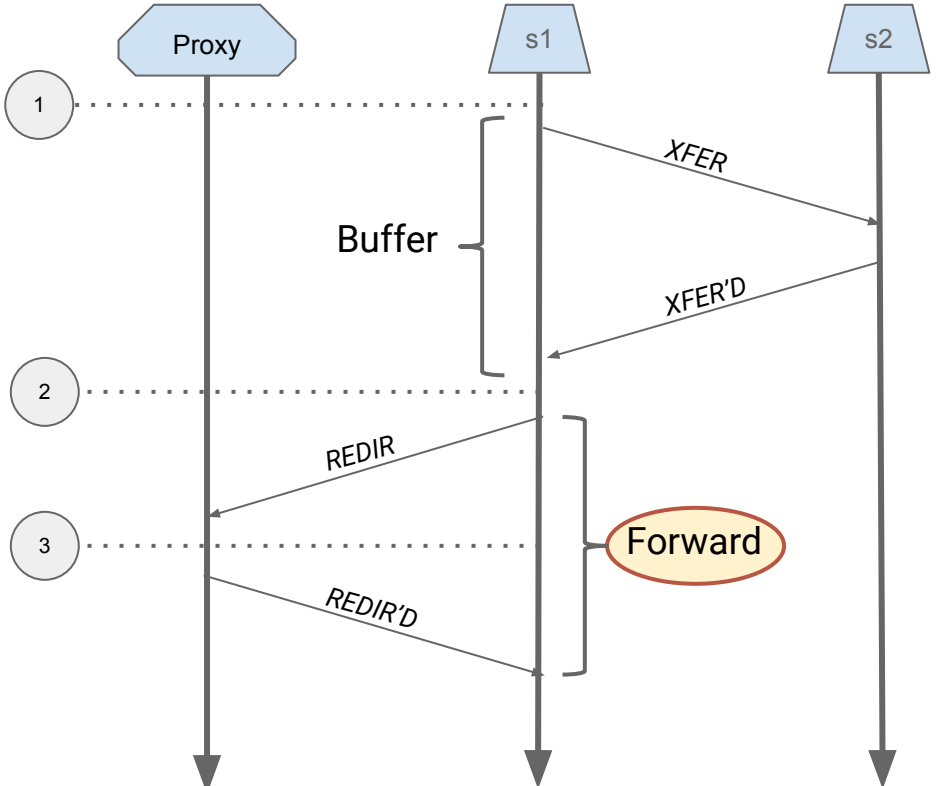


Migration Protocol: eBPF Tools :(

Buffering and forwarding can be accomplished with eBPF, but not without a struggle

Forwarding is the simpler of the two

- Out-of-the-box, and fast, with XDP
- However!

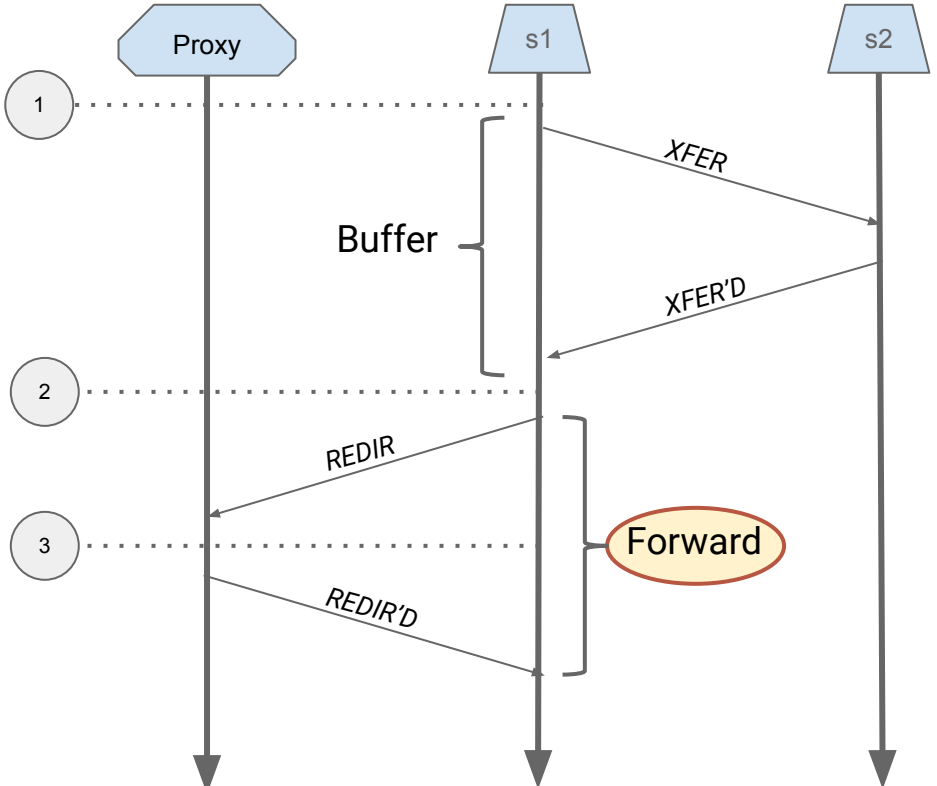


Migration Protocol: eBPF Tools

Buffering and forwarding can be accomplished with eBPF, but not without a struggle

Forwarding is the simpler of the two

- Out-of-the-box, and fast, with XDP
- However!
  - Simulation router required that source IP matches router port subnet
  - Thus, encapsulate to enable recovery of source

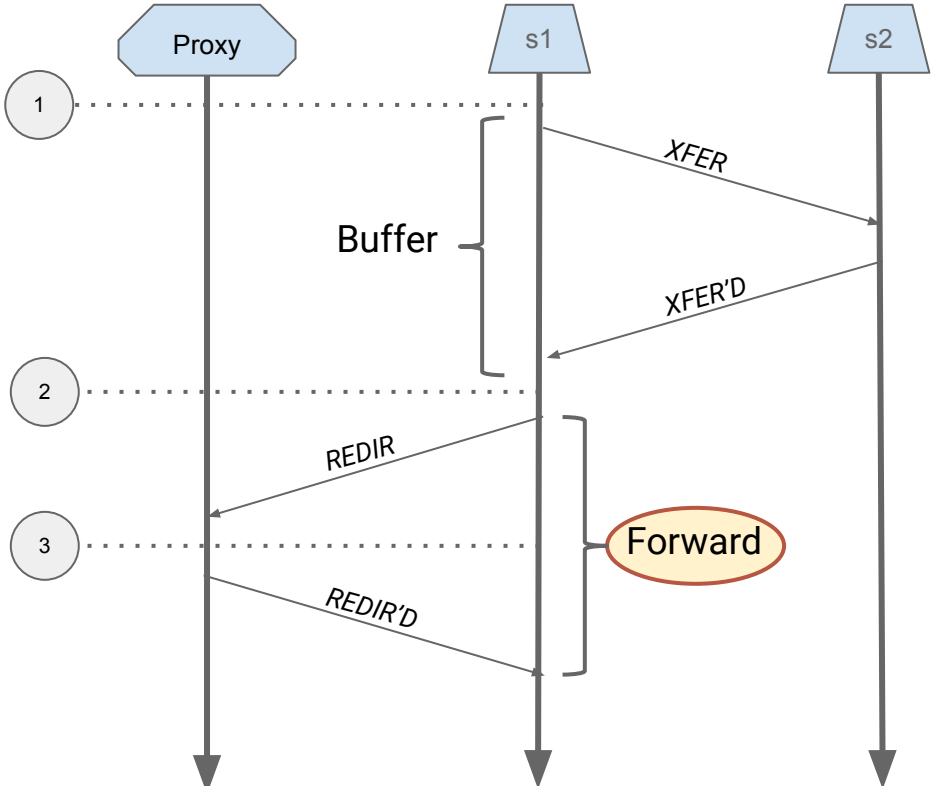


Migration Protocol: eBPF Tools :(

Buffering and forwarding can be accomplished with eBPF, but not without a struggle

Forwarding is the simpler of the two

- Out-of-the-box, and fast, with XDP
- However!
  - Simulation router required that source IP matches router port subnet
  - Thus, encapsulate to enable recovery of source
  - However!

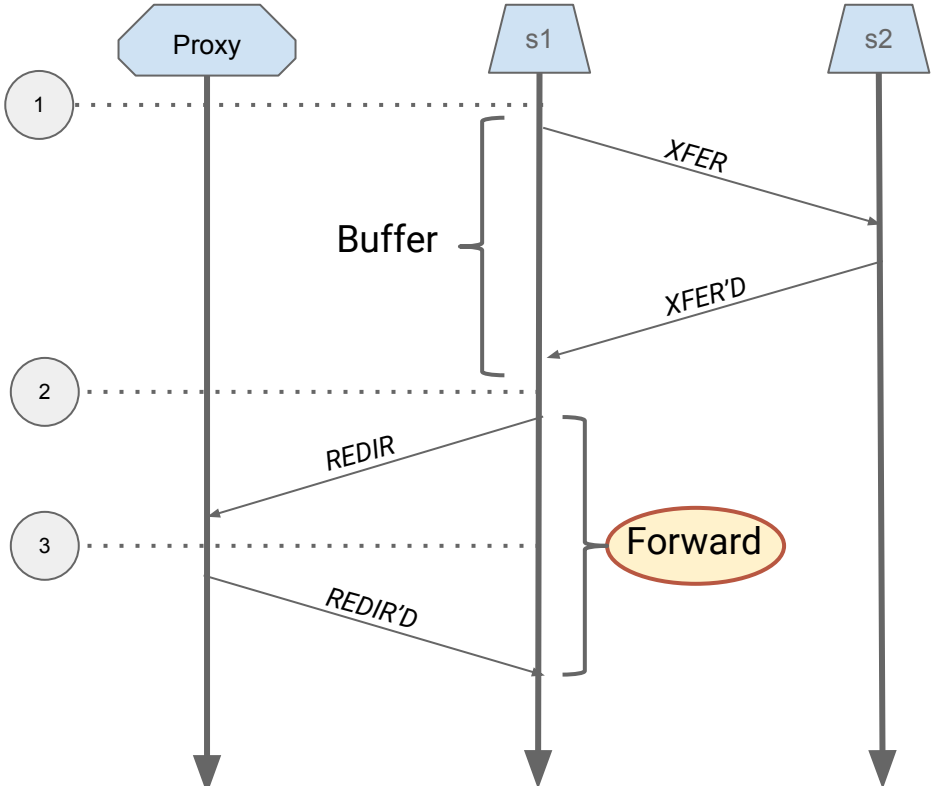


Migration Protocol: eBPF Tools

Buffering and forwarding can be accomplished with eBPF, but not without a struggle

Forwarding is the simpler of the two

- Out-of-the-box, and fast, with XDP
- However!
  - Simulation router required that source IP matches router port subnet
  - Thus, encapsulate to enable recovery of source
  - However!
    - TC does not support enlarging packets
    - XDP does not provide utilities for calculating TCP checksum, and cannot loop over packet buffer
    - Encapsulate in UDP (checksum optional)

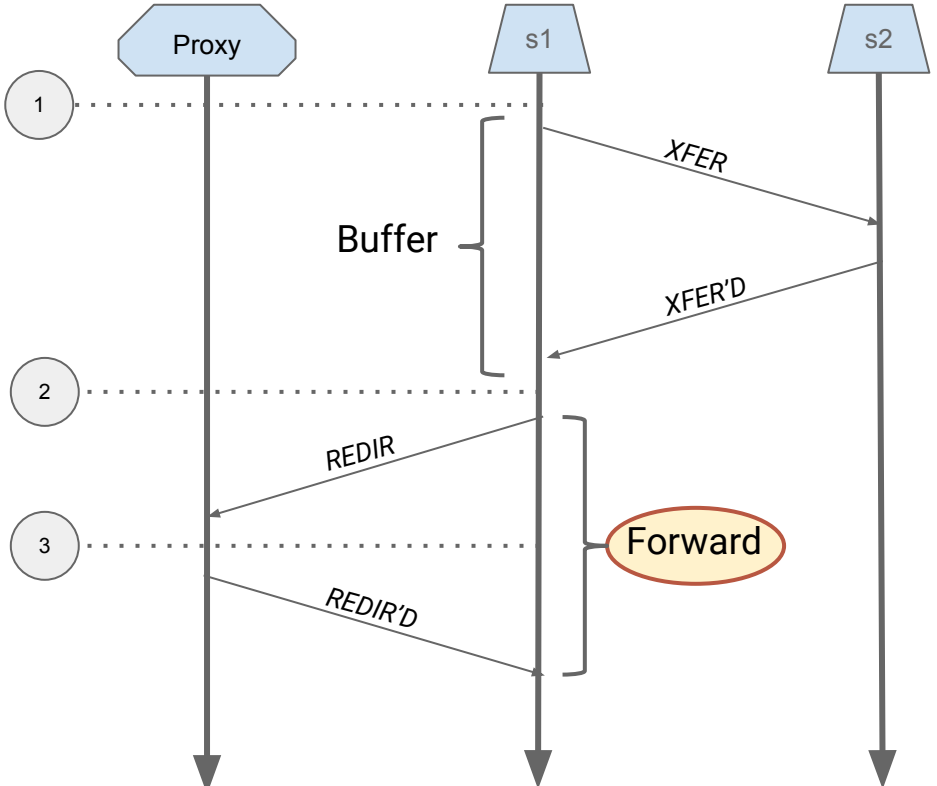


Migration Protocol: eBPF Tools :(

Buffering and forwarding can be accomplished with eBPF, but not without a struggle

Forwarding is the simpler of the two

- Out-of-the-box, and fast, with XDP
- However!
  - Simulation router required that source IP matches router port subnet
  - Thus, encapsulate to enable recovery of source
  - However!
    - TC does not support enlarging packets
    - XDP does not provide utilities for calculating TCP checksum, and cannot loop over packet buffer
    - Encapsulate in UDP (checksum optional)
    - However!

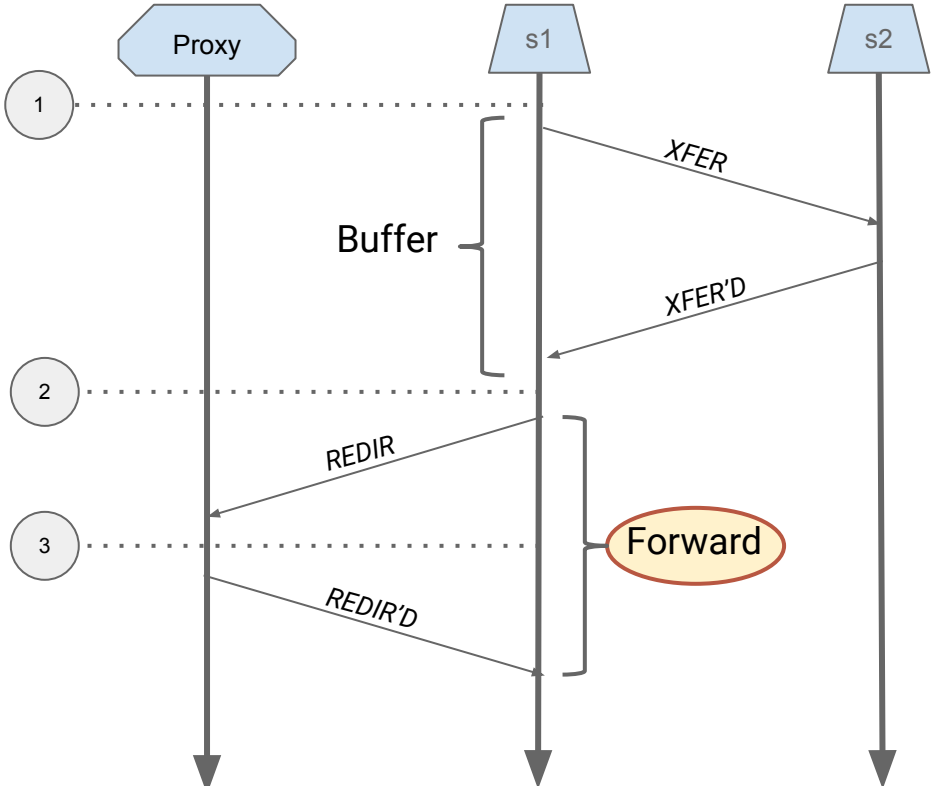


Migration Protocol: eBPF Tools

Buffering and forwarding can be accomplished with eBPF, but not without a struggle

Forwarding is the simpler of the two

- Out-of-the-box, and fast, with XDP
- However!
  - Simulation router required that source IP matches router port subnet
  - Thus, encapsulate to enable recovery of source
  - However!
    - TC does not support enlarging packets
    - XDP does not provide utilities for calculating TCP checksum, and cannot loop over packet buffer
    - Encapsulate in UDP (checksum optional)
    - However!
      - Undocumented bug in ubuntu 16.04 adds space to the wrong end of the packet

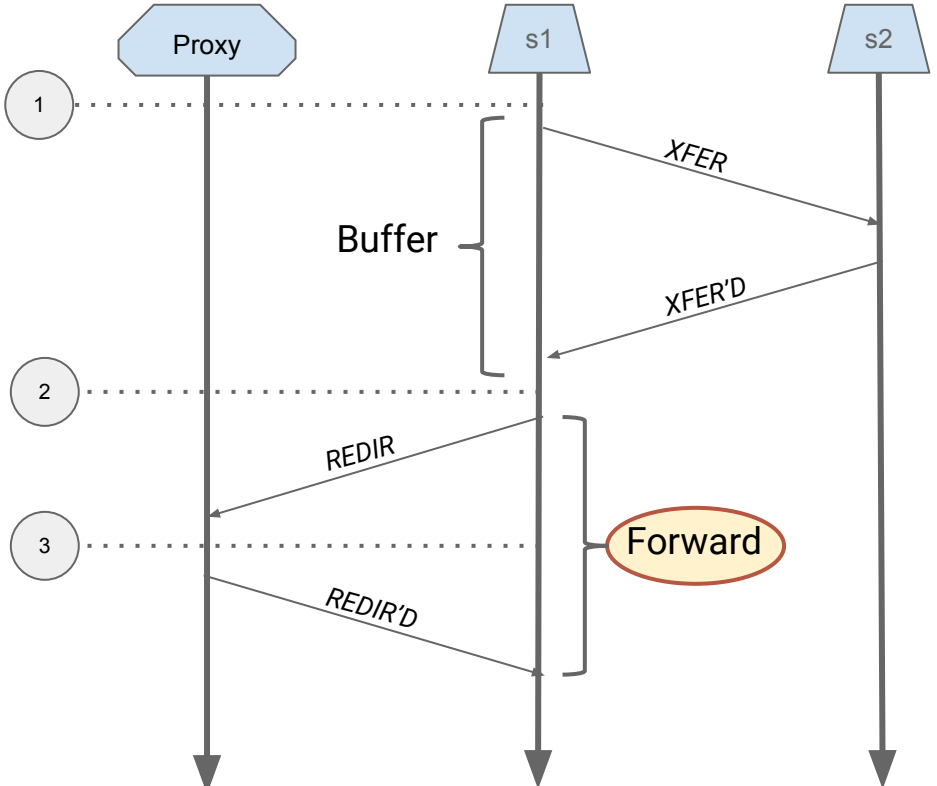


Migration Protocol: eBPF Tools :)

Buffering and forwarding can be accomplished with eBPF, but not without a struggle

Forwarding is the simpler of the two

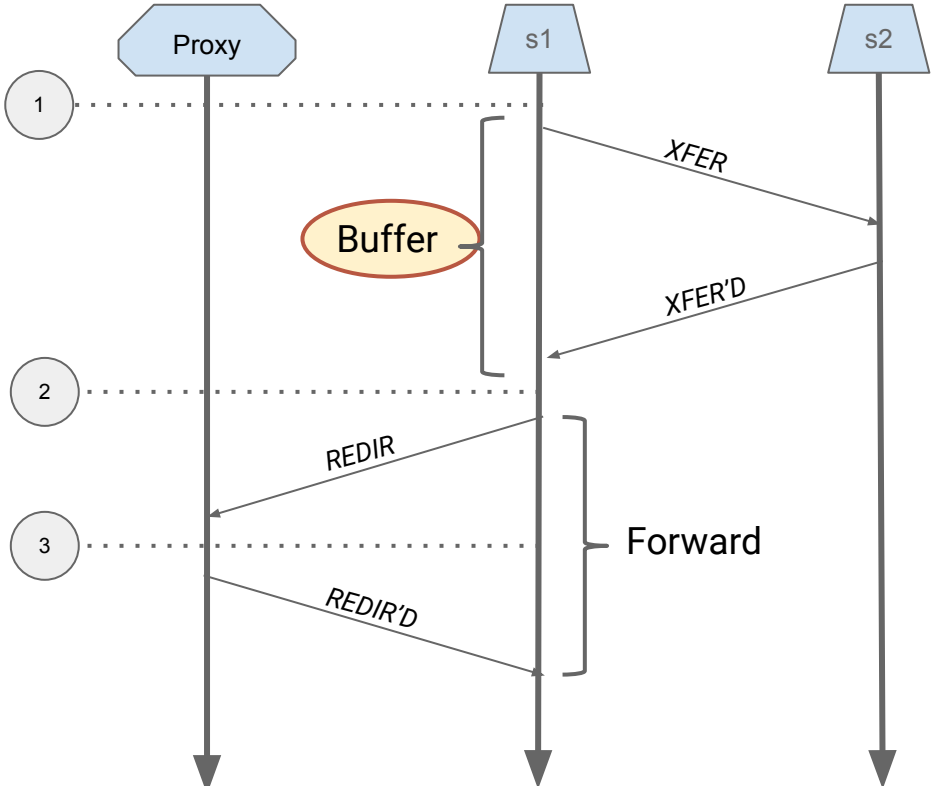
- Out-of-the-box, and fast, with XDP
- However!
  - Simulation router required that source IP matches router port subnet
  - Thus, encapsulate to enable recovery of source
  - However!
    - TC does not support enlarging packets
    - XDP does not provide utilities for calculating TCP checksum, and cannot loop over packet buffer
    - Encapsulate in UDP (checksum optional)
    - However!
      - Undocumented bug in ubuntu 16.04 adds space to the wrong end of the packet
      - Get frustrated and switch to a different cluster ✓



Migration Protocol: eBPF Tools

Buffering and forwarding can be accomplished with eBPF, but not without a struggle

Buffering is not directly supported by eBPF

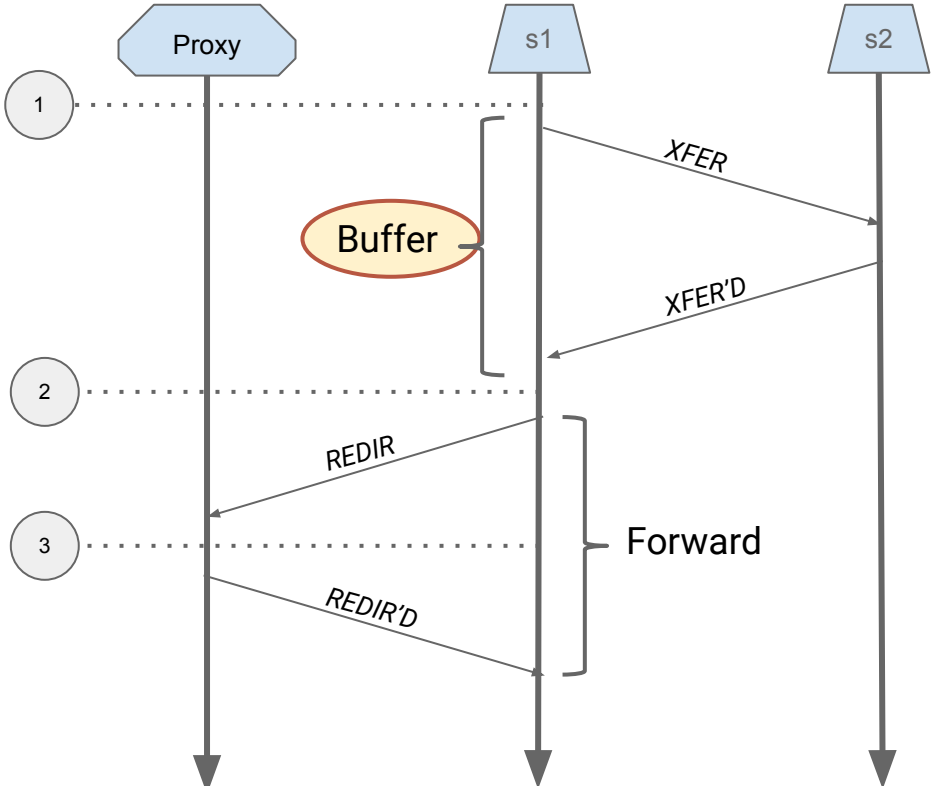


Migration Protocol: eBPF Tools

Buffering and forwarding can be accomplished with eBPF, but not without a struggle

Buffering is not directly supported by eBPF

- Flexibly sized storage or memory copy is unavailable
- No way to generate new packets at a later time
- Luckily, eBPF can attach to the loopback interface, and use that as a temporary buffer

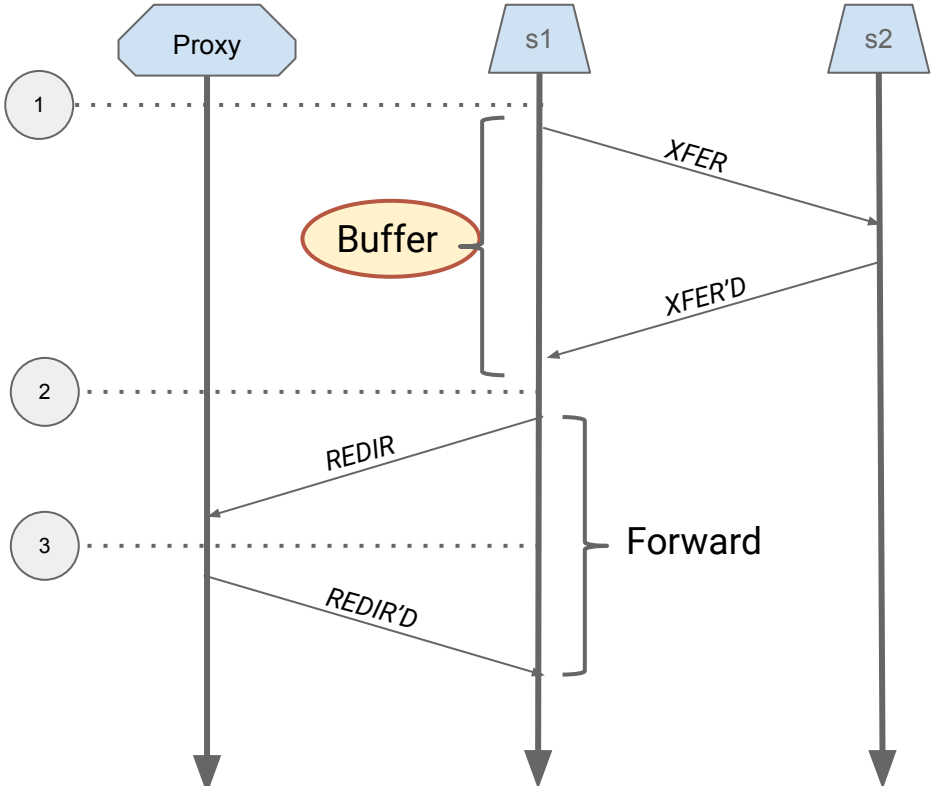


Migration Protocol: eBPF Tools :(

Buffering and forwarding can be accomplished with eBPF, but not without a struggle

Buffering is not directly supported by eBPF

- Flexibly sized storage or memory copy is unavailable
- No way to generate new packets at a later time
- Luckily, eBPF can attach to the loopback interface, and use that as a temporary buffer
- However!

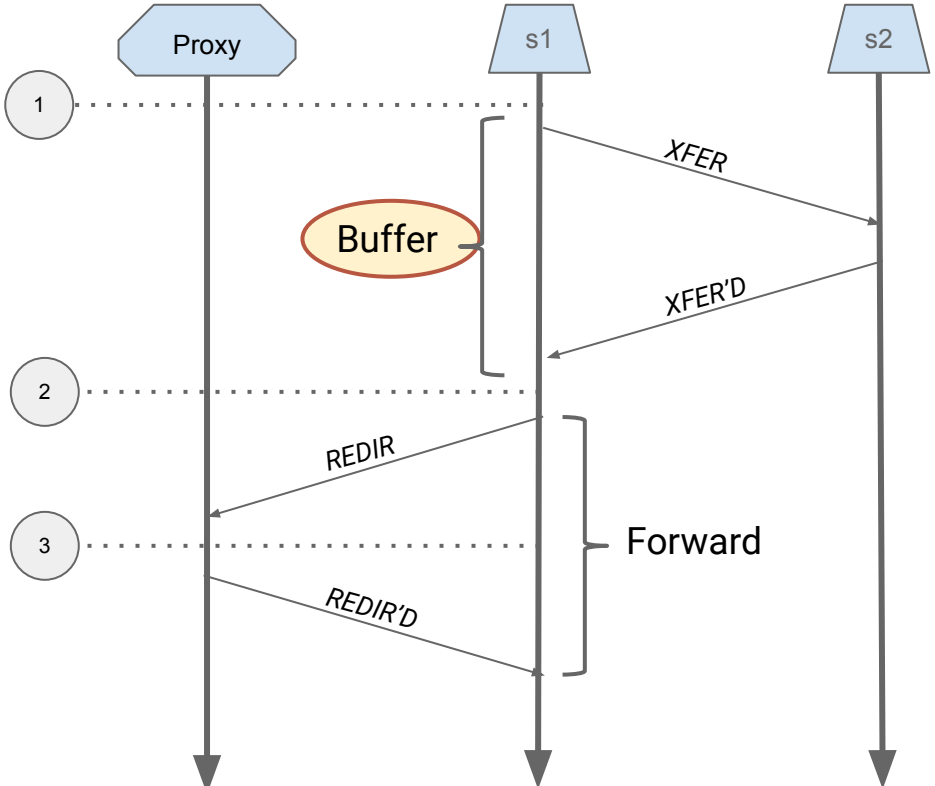


Migration Protocol: eBPF Tools

Buffering and forwarding can be accomplished with eBPF, but not without a struggle

Buffering is not directly supported by eBPF

- Flexibly sized storage or memory copy is unavailable
- No way to generate new packets at a later time
- Luckily, eBPF can attach to the loopback interface, and use that as a temporary buffer
- However!
  - In simulation, XDP can redirect to the loopback - on hardware it silently fails
  - TC can redirect to loopback, but cannot encapsulate (see previous slide)

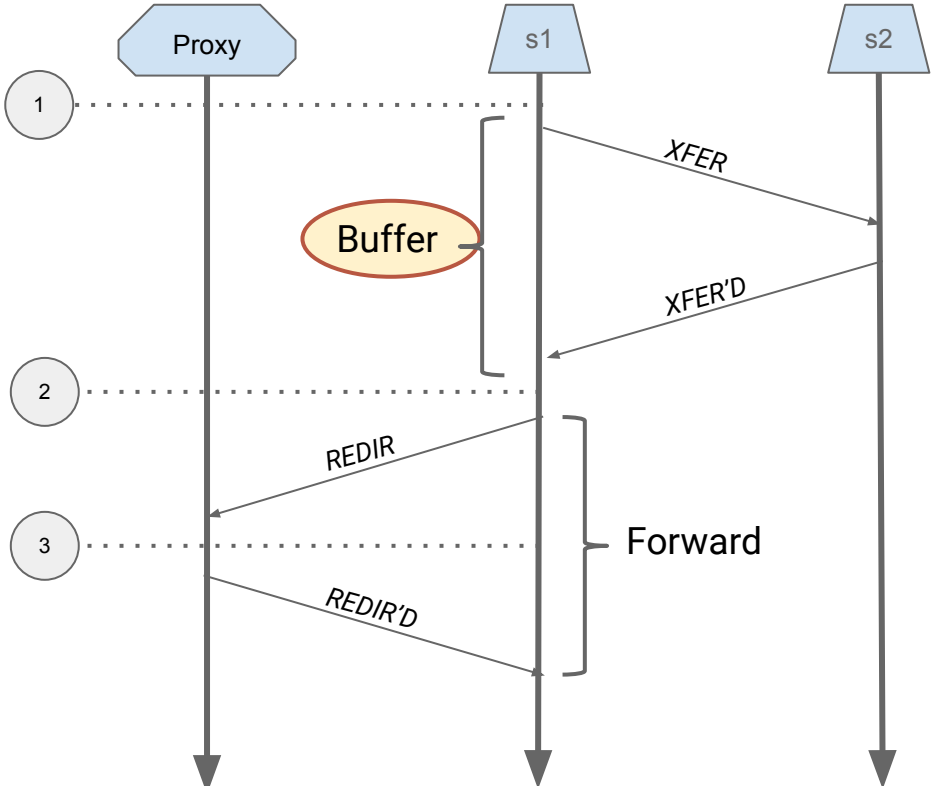


Migration Protocol: eBPF Tools

Buffering and forwarding can be accomplished with eBPF, but not without a struggle

Buffering is not directly supported by eBPF

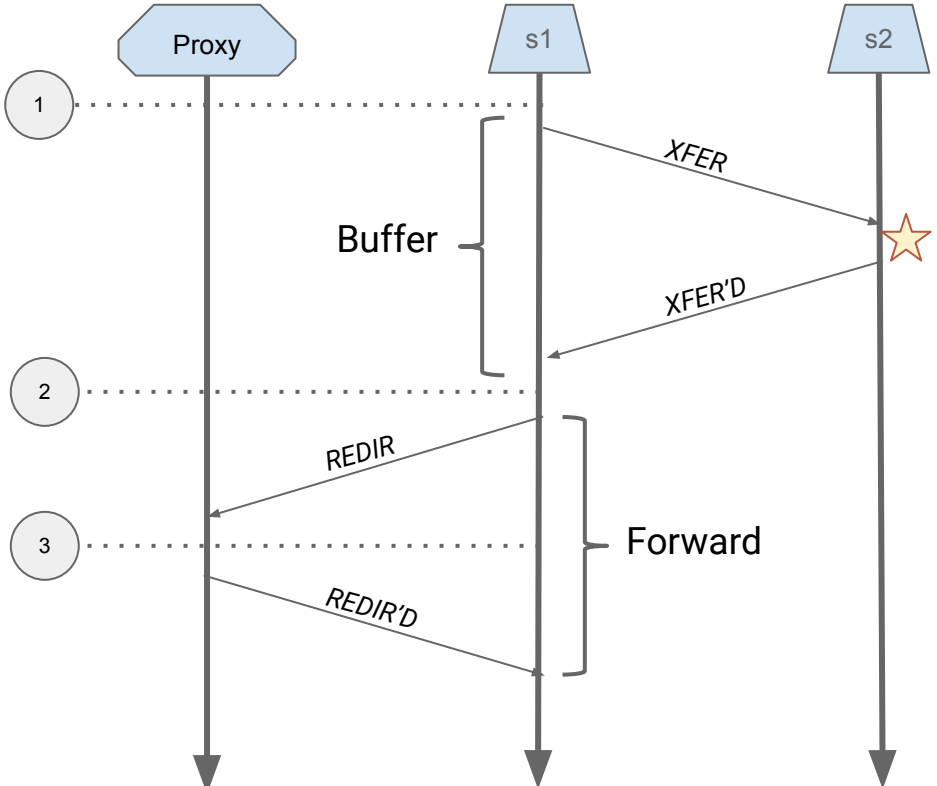
- Flexibly sized storage or memory copy is unavailable
- No way to generate new packets at a later time
- Luckily, eBPF can attach to the loopback interface, and use that as a temporary buffer
- However!
  - In simulation, XDP can redirect to the loopback - on hardware it silently fails
  - TC can redirect to loopback, but cannot encapsulate (see previous slide)
  - Thus:
    - 1) Encapsulate with XDP
    - 2) Forward to loopback with TC
    - 3) if buffering : GOTO 2
    - 4) else: forward to egress on interface
    - 5) Decapsulate with XDP on receiver ✓



Migration Protocol: eBPF Tools

One last eBPF probe was included to improve performance:

- Recall: unsetting TCP_REPAIR causes a window probe to be sent
- More recent updates make this probe optional (https://lore.kernel.org/patchwork/patch/962787/)
- Ebpf probe installed to automatically acknowledge a single message
  - Allows sending and receipt of messages to begin immediately

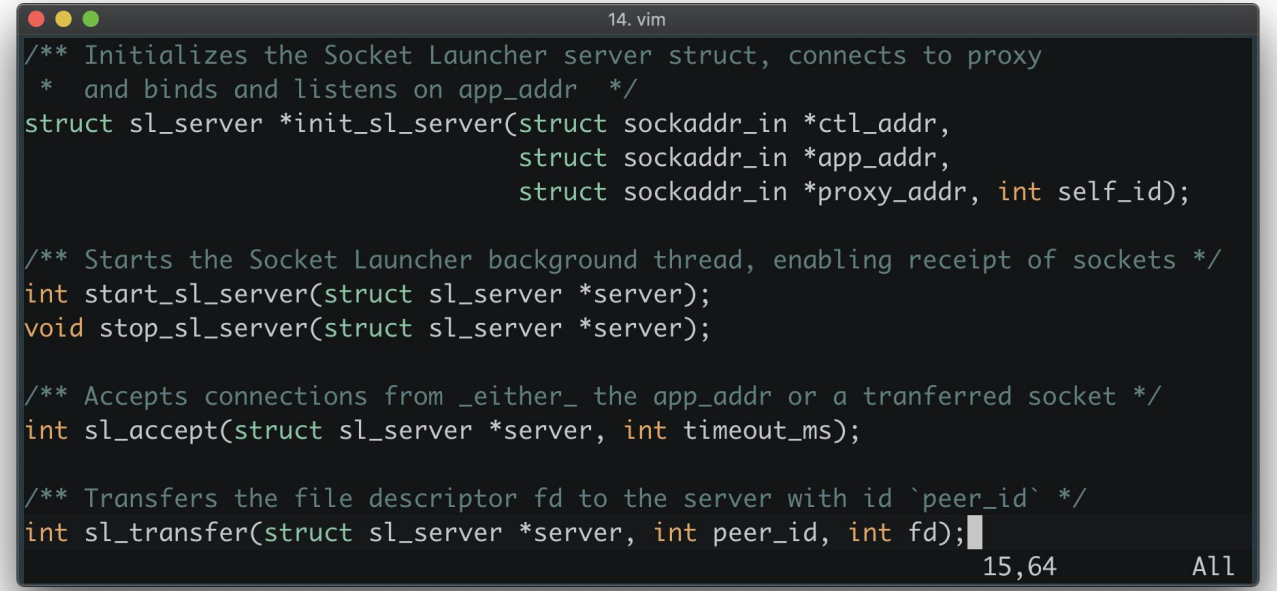


Library Design

- What is the API that SL provides?
- How does an SL enabled application function?
- What is design considerations affected SL performance?

Library Design: API

SL Server API consists of the following five functions in addition to exposing a socket that can be used for polling



Upon connection to the proxy, the server is notified of all other connected peers

Library Design: Considerations and regrets

- SL Server socket communication is entirely lockless
  - Scatter-gather sending ensures that multi-part messages are received intact without any sort of locking
  - Removal of locks improved performance almost 10x

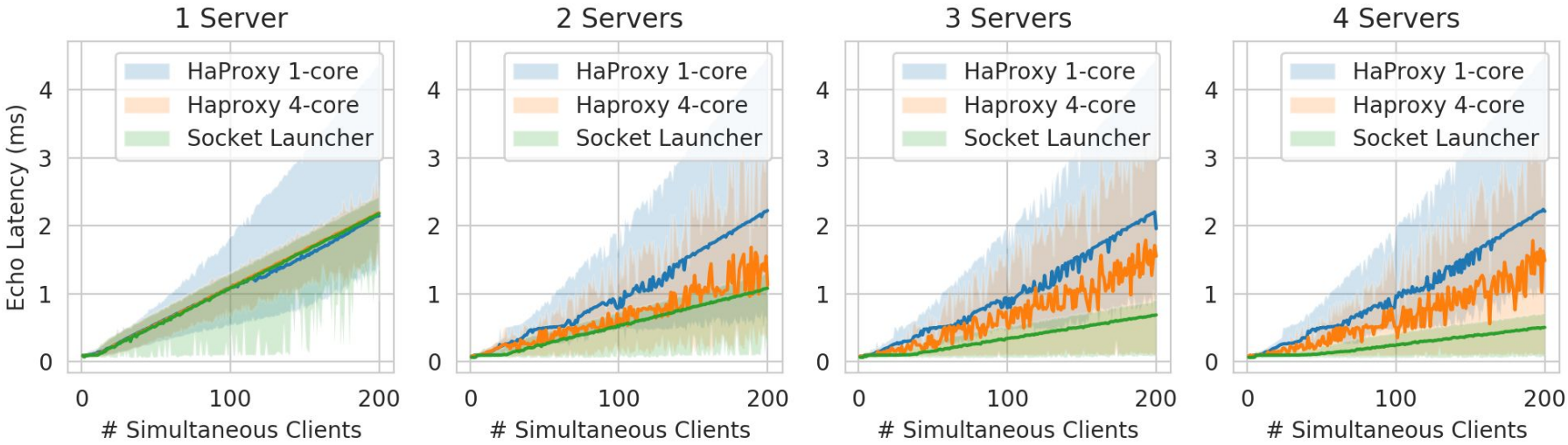
- Locks used for communication with eBPF python frontend account for at least half of transfer latency
  - eBPF management runs it its own process, using the BCC python interface
  - IPC communication between c and python using zmq was ultimately a mistake
  - Would ideally be integrated into a single process using eBPF's C interface

Evaluation: Micro-benchmarks

- What is the performance of the Socket Launcher proxy?
- What is the cost associated with transferring a socket?

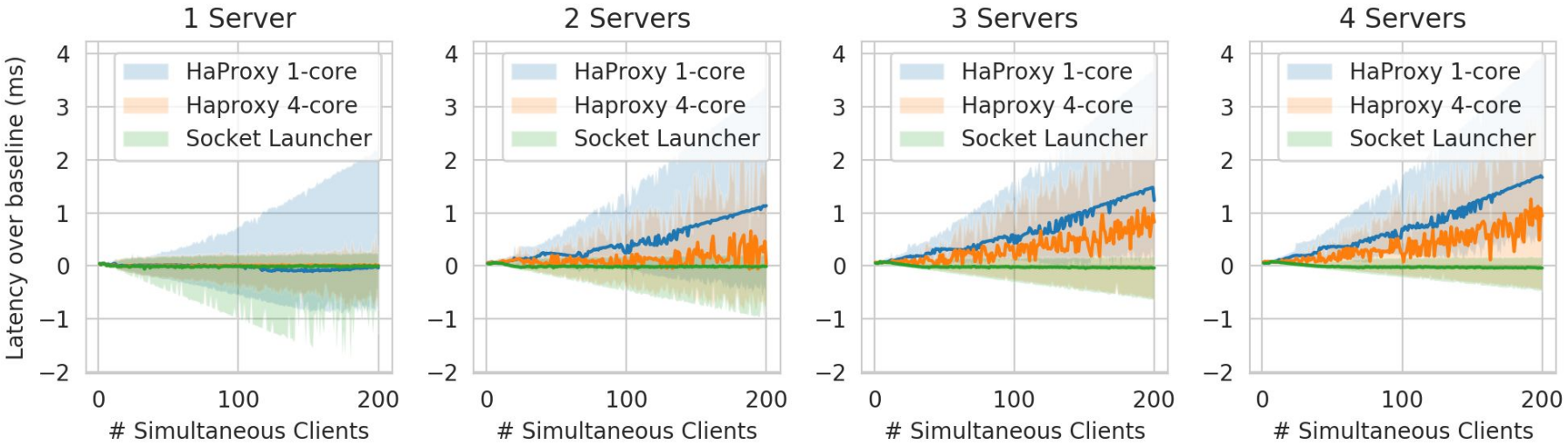
Micro-Benchmarks: Proxy Latency

- Latency measured using echo-server
- Balancing load across 1-4 servers
- Compared with HaProxy as a baseline
- Line shows median, shaded regions shows 1st - 99th percentile



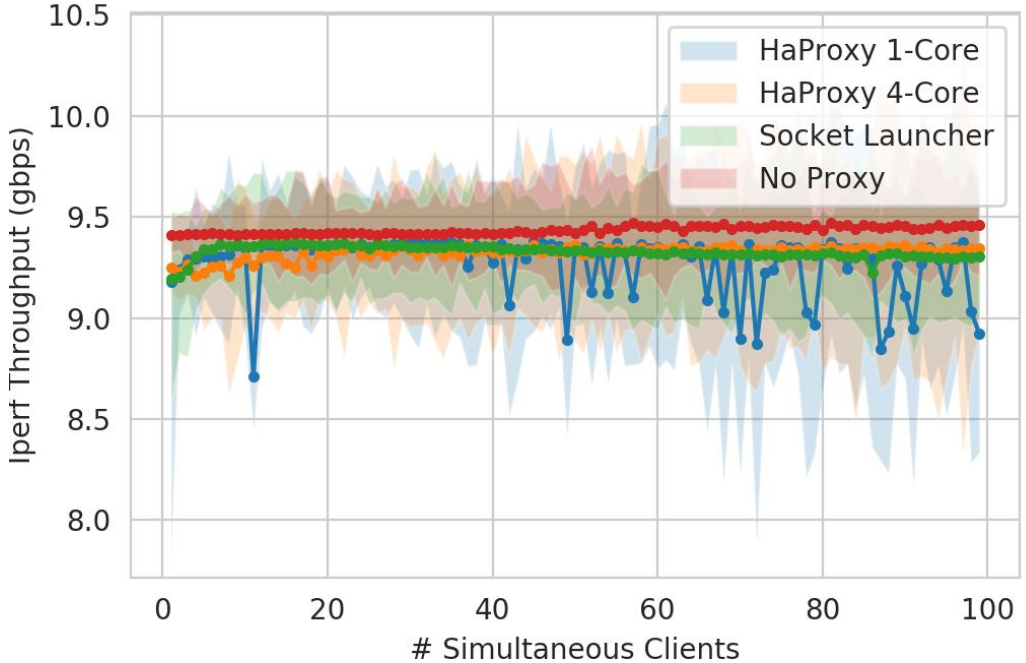
Micro-Benchmarks: Proxy Latency

- Subtracted ideal performance (no proxy / # of servers)
- HaProxy suffers with more servers and higher concurrency
- SLProxy does not add significant latency on top of baseline measurements



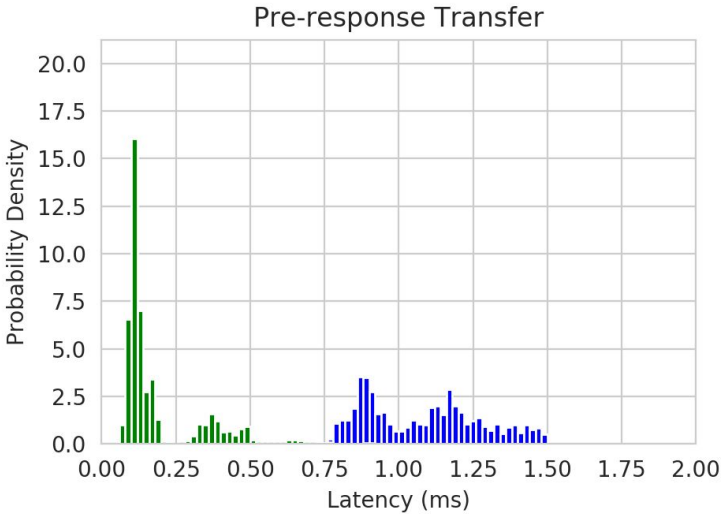
Micro-Benchmarks: Proxy Throughput

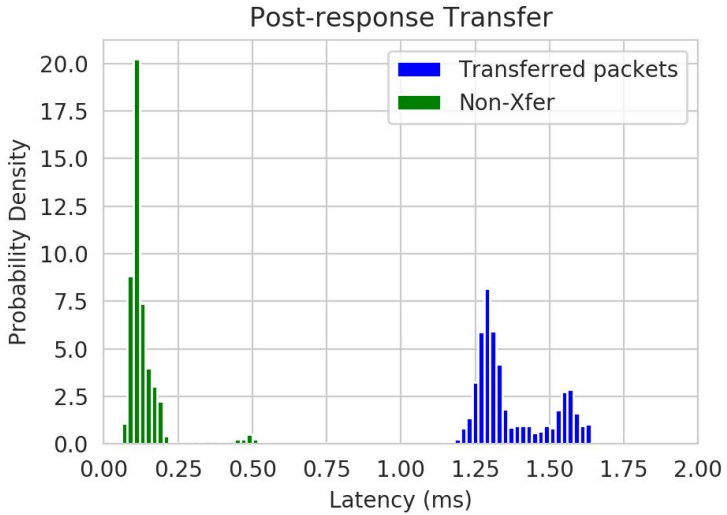
- Iperf used to test maximum proxy throughput
- Cannot be load-balanced across servers due to iperf limitations
- 30 reporting intervals used
- Points show mean
- Shading shows max/min
- SLProxy slightly underperforms, especially with low numbers of clients



Micro-Benchmarks: Transfer Latency

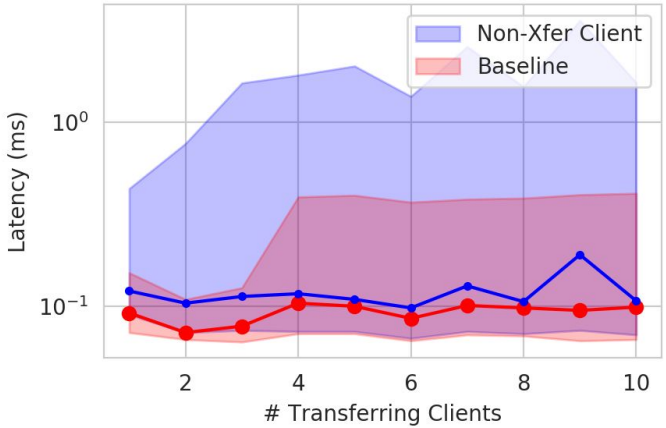
- Client initiates transfer by sending “xfer” message to server : 1 / 10 packets is XFER
- Two tested scenarios:
  1. Pre-response: s1 peeks at buffer and sends to s2 who responds
  2. Post-response: s1 reads from buffer, responds, then transfers to s2 for the next message

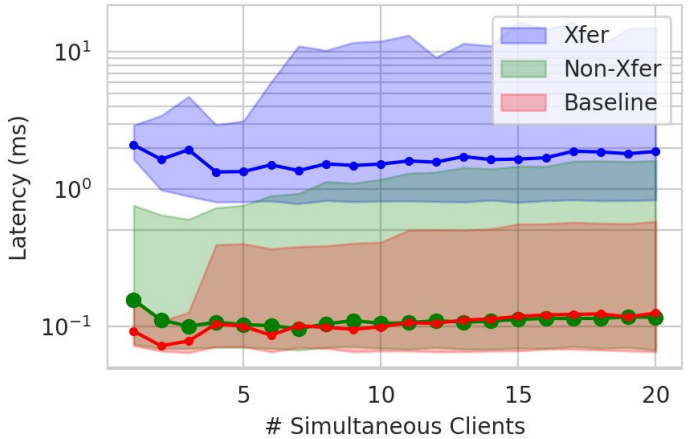




Micro-Benchmarks: Transfer Latency

- Client initiates transfer by sending “xfer” message to server : 1 / 10 packets is XFER (pre-response)
- Varying number of simultaneously connected and transferring clients
- Transfer latency ~1ms
- Minimal effect on median latency of other packets on the same connection (right), or unrelated packets (left)
- Does have a significant effect on tail latency for all packets





Evaluation: End-to-end

- How can transferrable sockets help to load-balance in the presence of uneven distribution of computational load?
- How do the parameters of the transfer process affect performance?

End-to-end Evaluation: Motivation

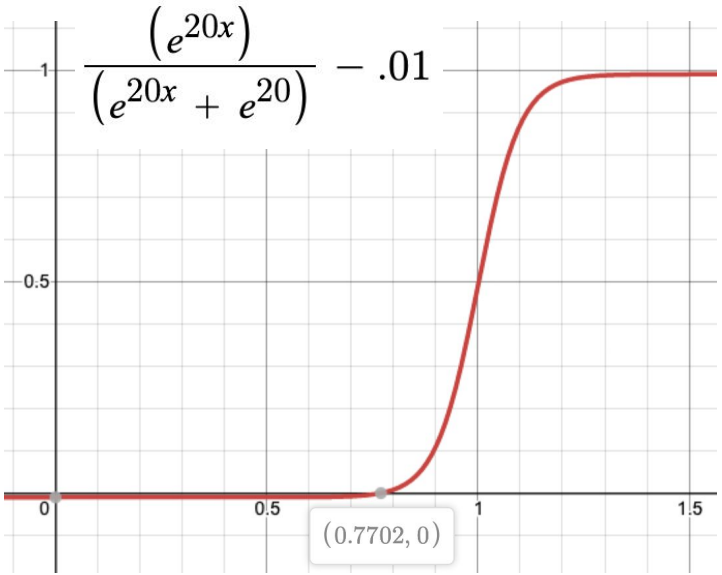
- Event-driven architectures are inherently reliant on queuing
- Typically, multiple threads can pull from the same queue
  - Reduces risk of a single anomalously expensive request blocking request processing
- Queues that directly serve TCP connections are incapable of balancing across runtimes
  - Thus, if queues on a single runtime can be blocked or delayed, normal traffic can suffer
  - Even if other peer nodes lay idle
- Socket launcher enables servers that lack resources to transfer connections to other machines for further processing

End-to-end Evaluation: Setup

- Two servers accept connections from clients
- Clients may send any number of requests to the servers
- Requests consist of a number which indicating an amount of computation that should be performed
    - Requests scale exponentially (e.g. requesting “10” requires 2^10 computations)
- The size of the client pool remains constant throughout the experiment
- If a client completes its specified number of requests, it is immediately replaced by a new client
- A small portion of clients request anomalously expensive computation, causing build-up in queues
- If Socket Launcher is enabled, queue length is polled before a new request is added
    - As the queue length approaches a threshold, requests are probabilistically transferred to another machine

End-to-end Evaluation: Setup

- If Socket Launcher is enabled, queue length is polled before a new request is added
  - As the queue length approaches a threshold, requests are probabilistically transferred to another machine
- Probability distribution follows a logistic function which is 50% likely to transfer a connection when the queue length reaches the threshold

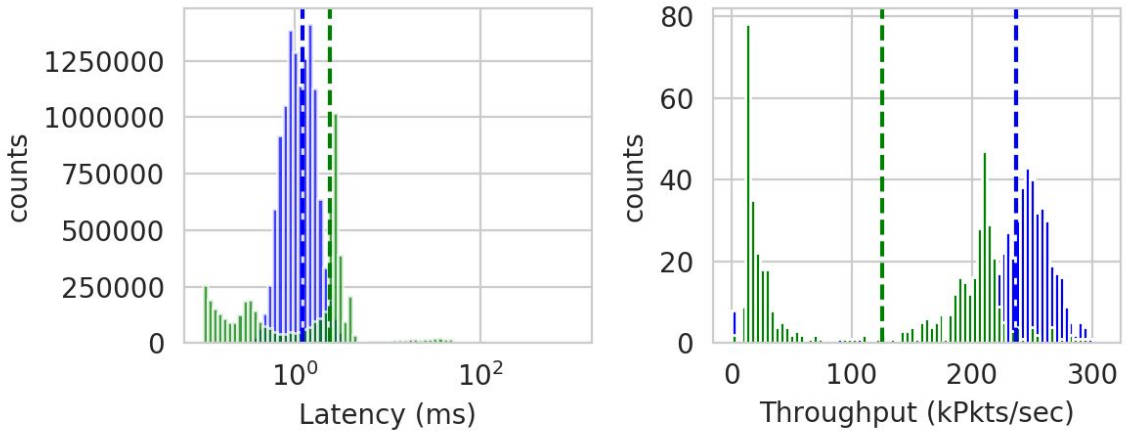


End-to-end Evaluation: Results

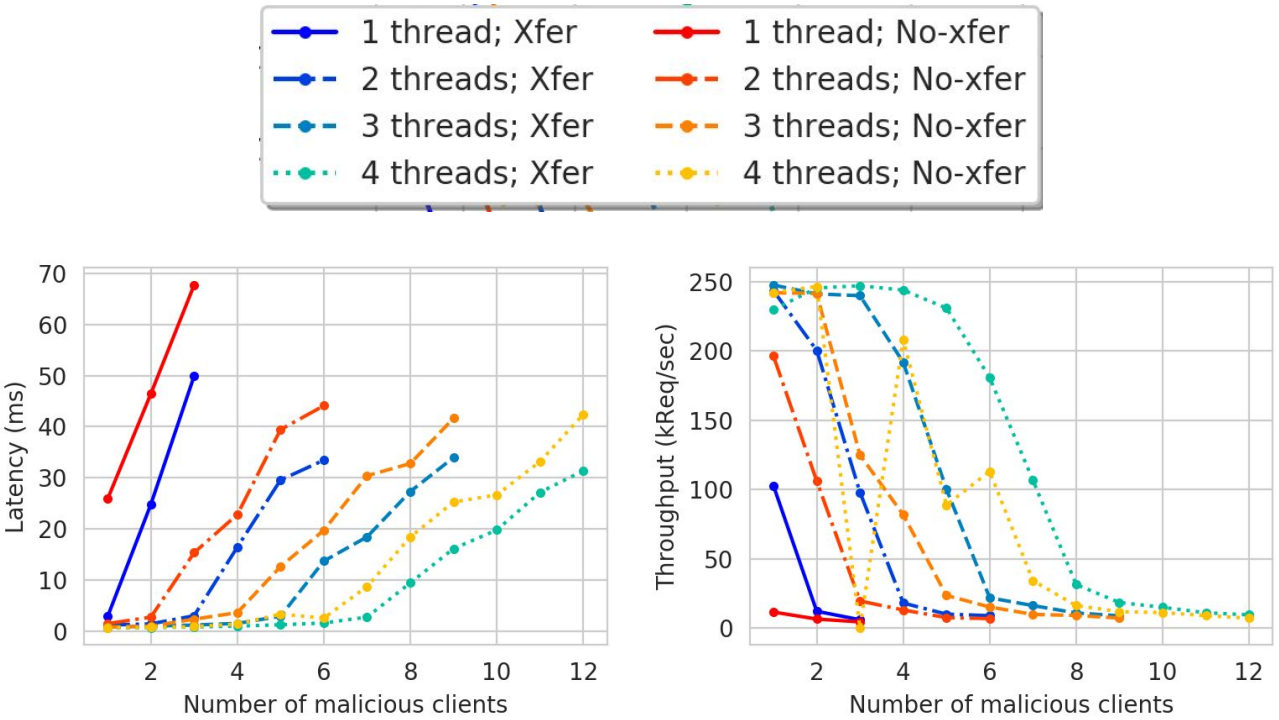
Many variables to tweak:

- Threads per runtime
- Simultaneous clients
- Requests per connection
- Malicious clients
- Strength of malicious requests
- Queue threshold

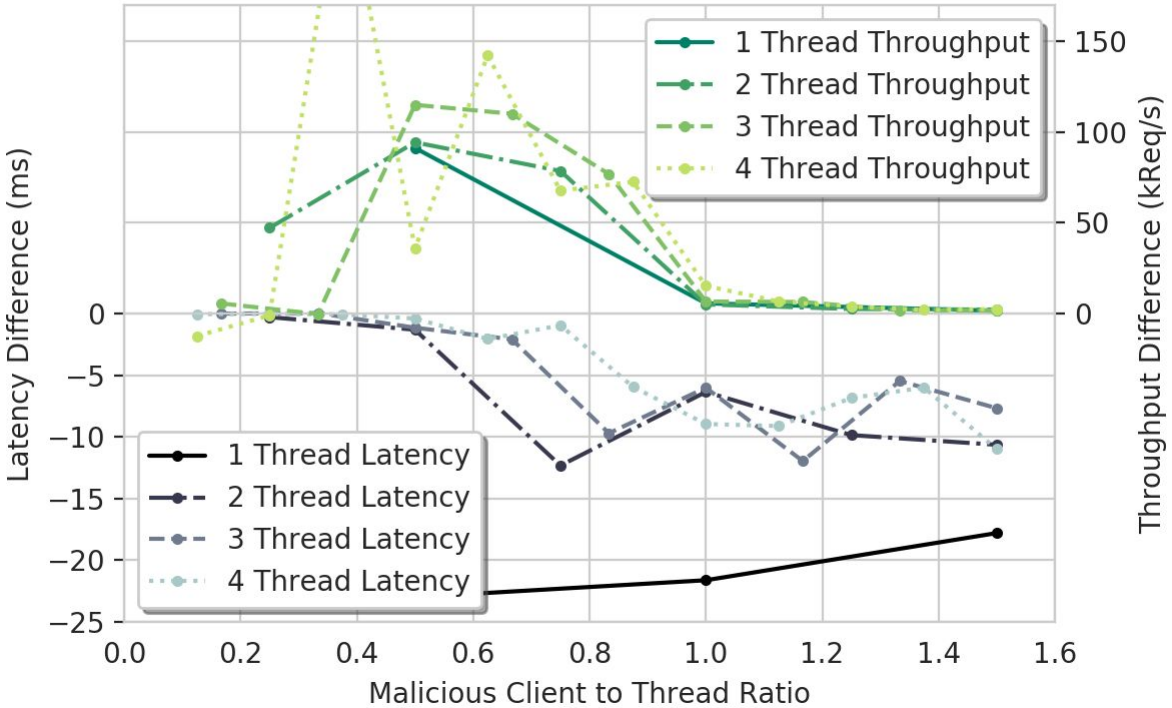
Sample distributions with 3 threads and 3 malicious clients



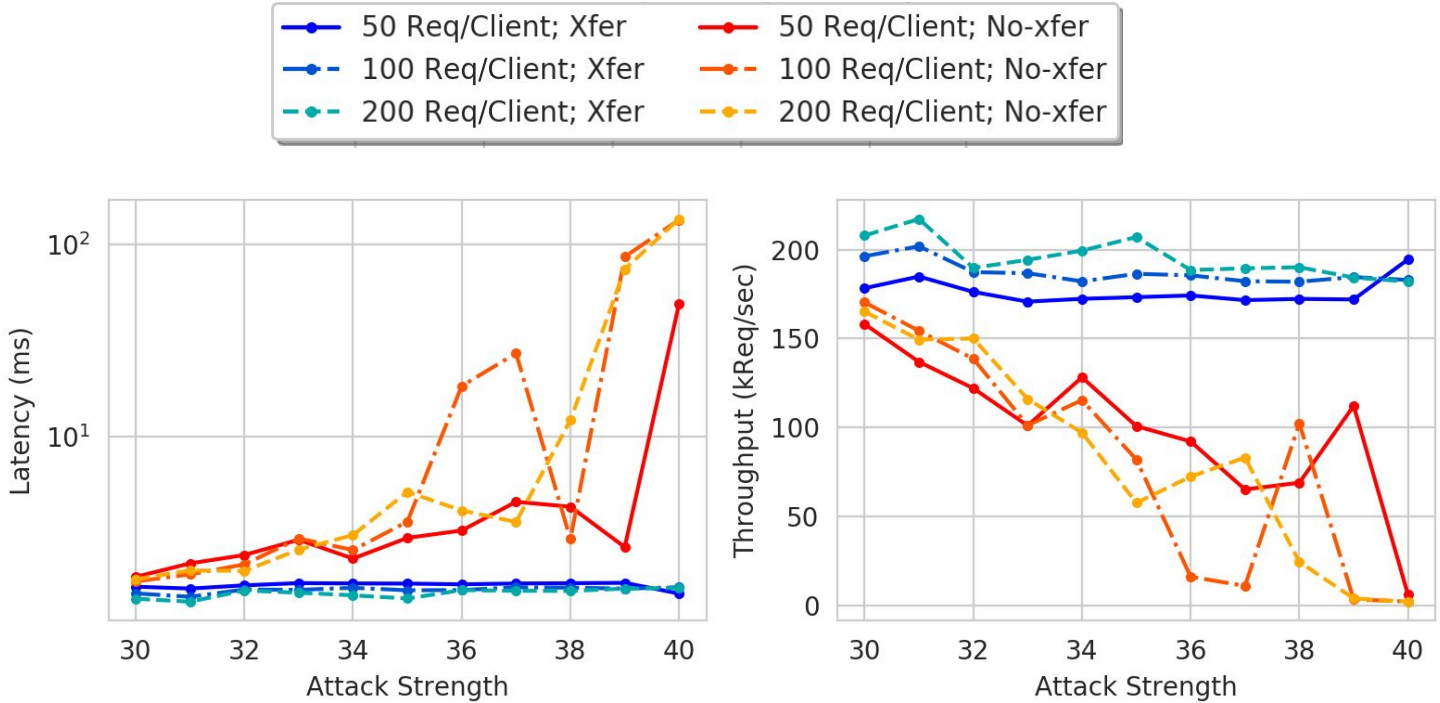
End-to-end Evaluation: Results (Threads)



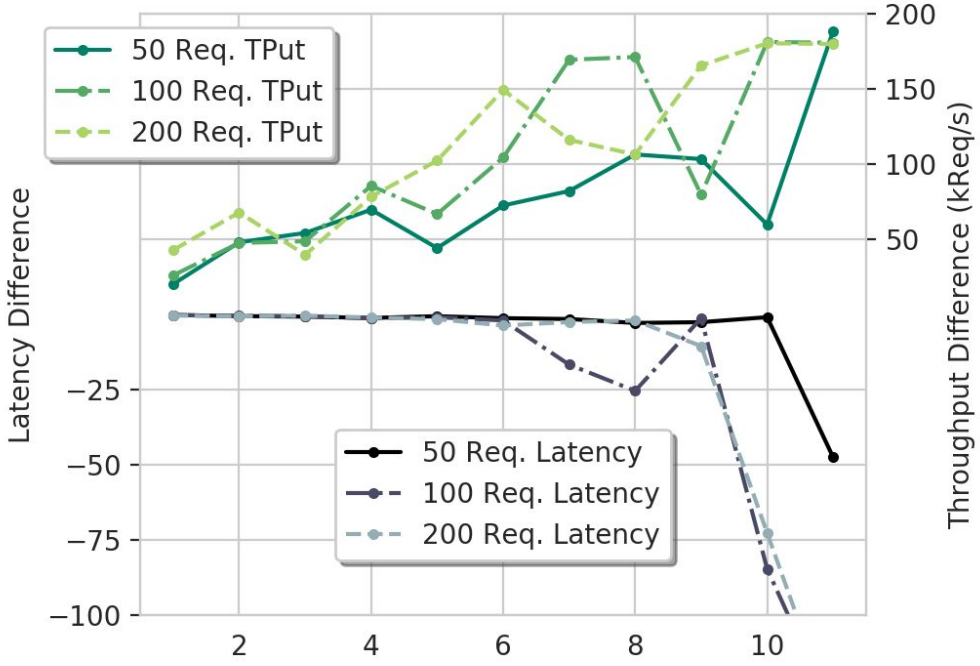
End-to-end Evaluation: Results (Threads)



End-to-end Evaluation: Results (Requests)



End-to-end Evaluation: Results (Requests)



Future Directions

- Framework improvements
  - Remove separation between eBPF and library
- Smarter load balancer
  - Load balancer is aware of the number of open connections
  - It could redirect traffic based on those numbers
- Dynamic determination of queue threshold
  - Communication between peers
  - “Stealing” work
- Transfer of application state
  - SSL migration (original intended project)
- Other use case:
  - Service-chaining model
    - (requires lower latency than is achieved with python eBPF/BCC)

Thank you!