mOS: A Reusable Networking Stack for Flow Monitoring Middleboxes

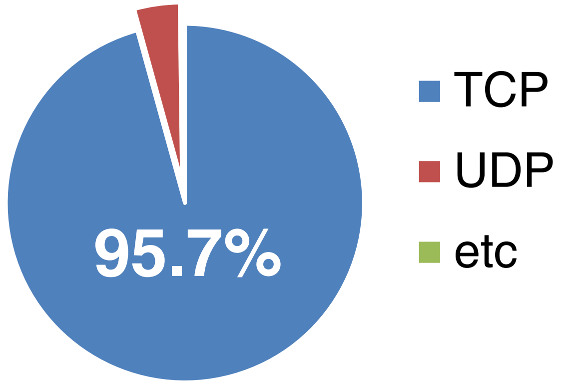
---

M. Asim Jamshed, YoungGyoun Moon, Donghwi Kim,
Dongsu Han, KyoungSoo Park

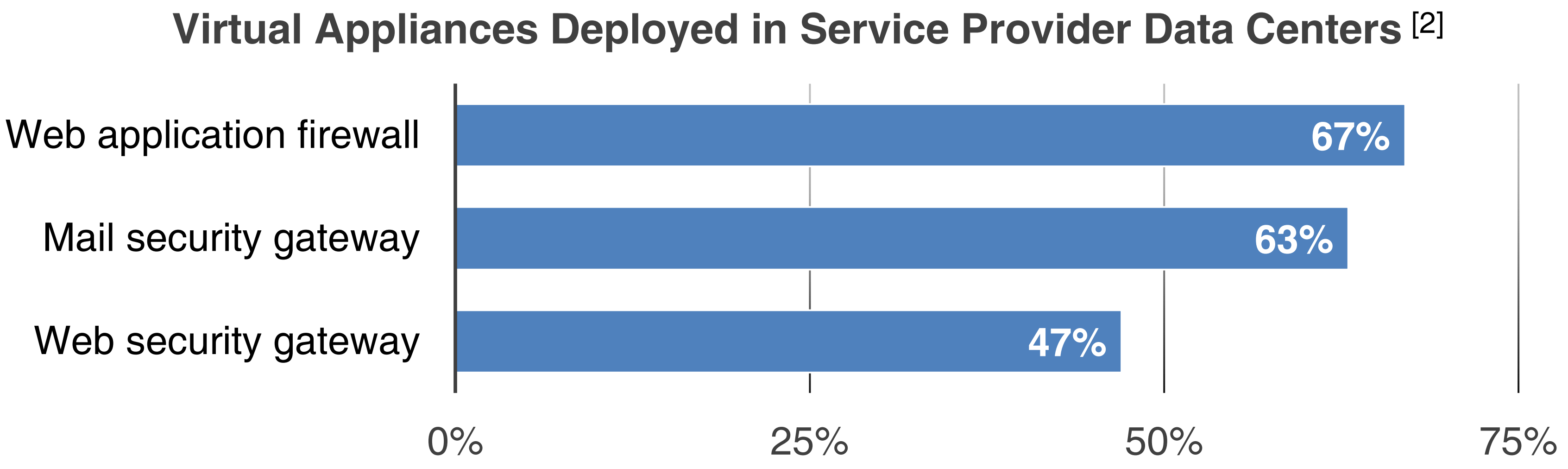
KAIST EE

Most Middleboxes Deal with TCP Traffic

- TCP dominates the Internet
  - 95+% of traffic is TCP [1]



- Top 3 middleboxes in service providers rely on L4/L7 semantics



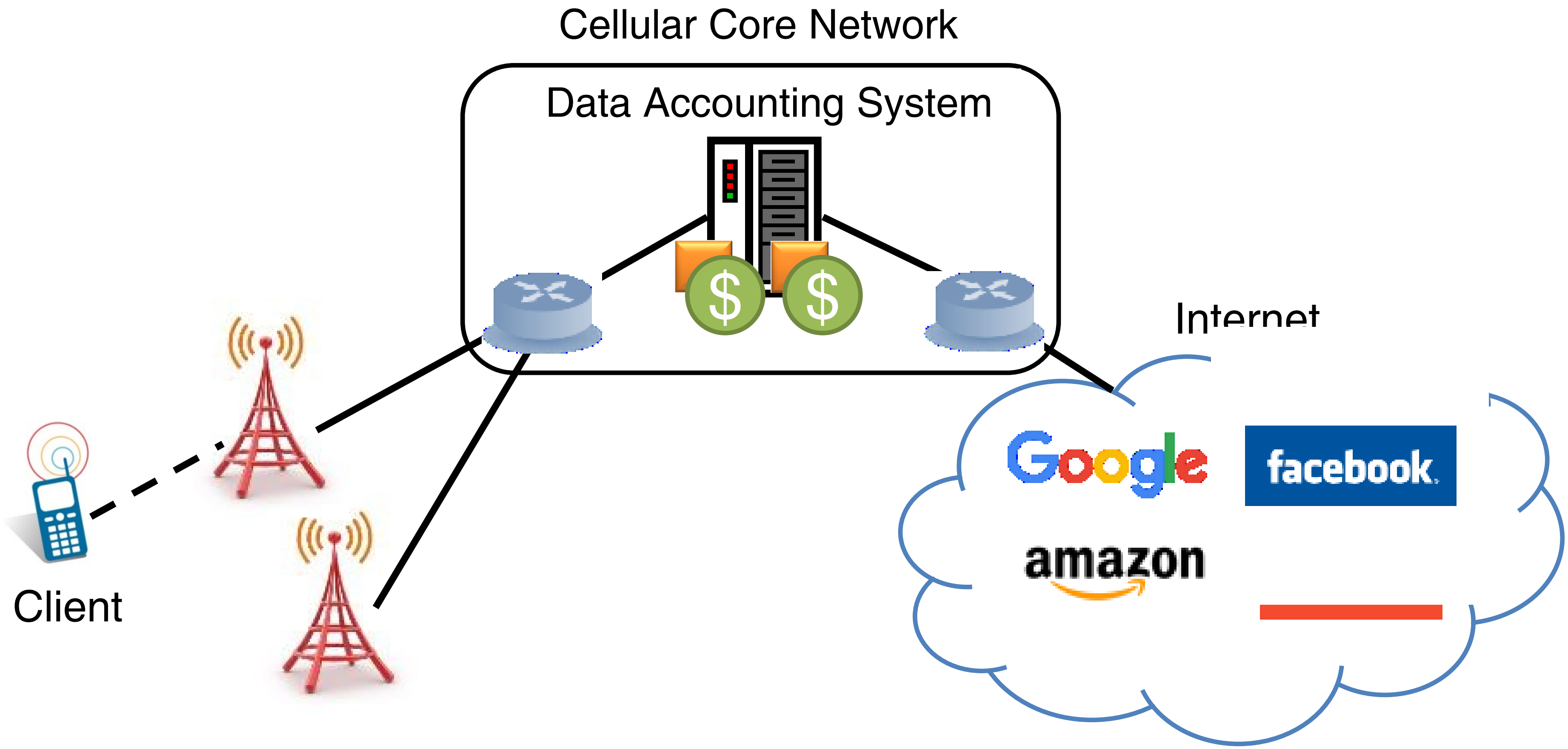
---

[1] "Comparison of Caching Strategies in Modern Cellular Backhaul Networks", ACM MobiSys 2013.

[2] IHS Infonetics Cloud & Data Center Security Strategies & Vendor Leadership: Global Service Provider Survey, Dec. 2014.

Example: Cellular Accounting System

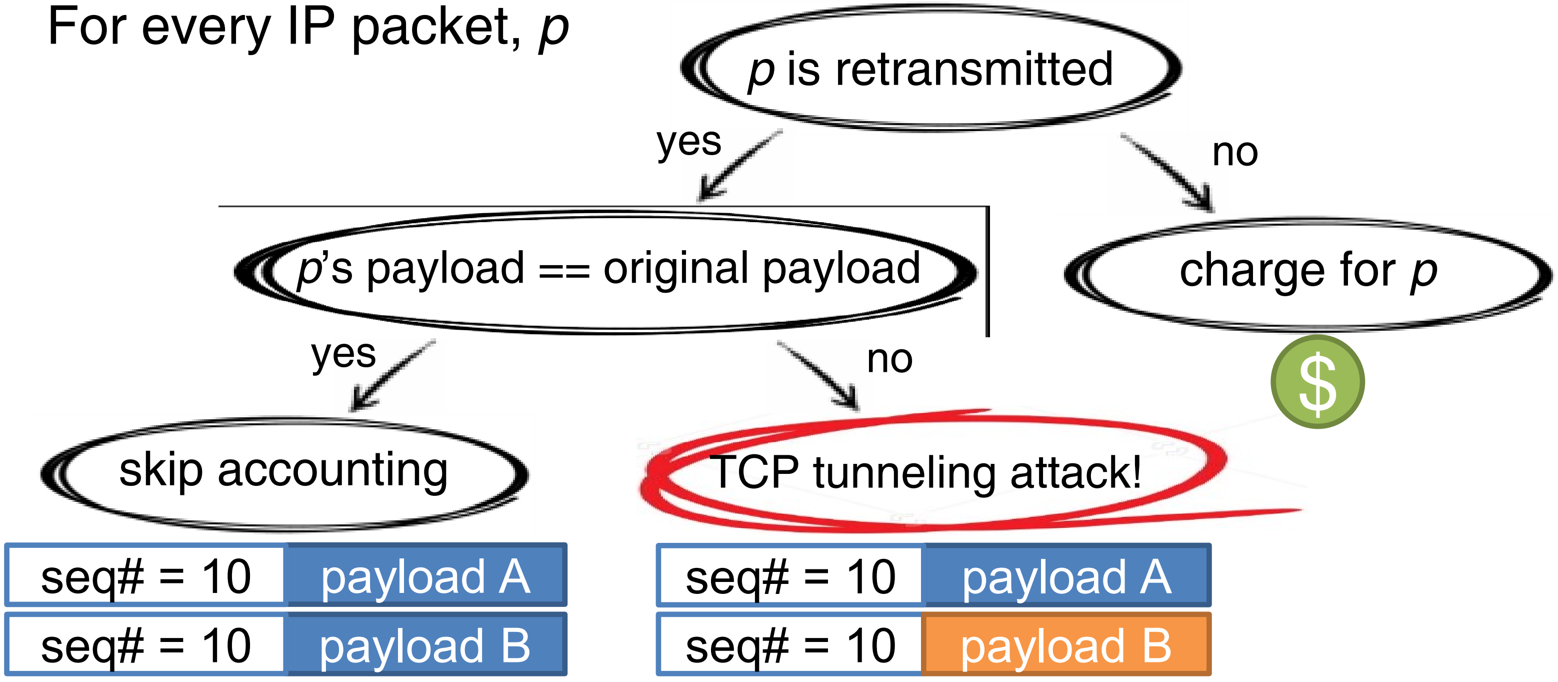
- Custom middlebox application
  - No open source solution



Challenges in Building Flow-level Middleboxes

- The main logic for a cellular accounting system
  - No charge for TCP retransmission, only if payloads match.

For every IP packet, *p*



p is retransmitted

yes

no

p's payload == original payload

charge for *p*

$

yes

no

skip accounting

TCP tunneling attack!

seq# = 10	payload A
seq# = 10	payload B

seq# = 10	payload A
seq# = 10	payload B

Core logic itself is straightforward!

Challenges in Building Flow-level Middleboxes

- Requires handling complex flow-level states and events

- The accounting system requires:
  - Reassembly buffer that holds the original payload
  - Non-contiguous fragments that holds the original payload
  - Event notification on TCP retransmission
  - Storage for per-flow accounting metadata and statistics

Challenges in Building Flow-level Middleboxes

- How to implement flow-processing features beneath its core logic?

Borrow code from open-source IDS (e.g., snort, suricata)

- 50K~100K code lines tightly coupled with their IDS logic

Borrow code from open-source kernel (e.g., Linux/FreeBSD)

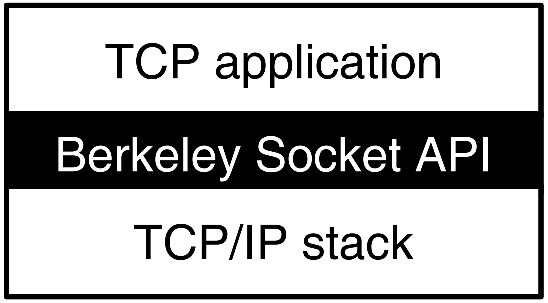
- Designed for TCP end host
- Different from middlebox semantics

Implement your own flow management code

- Complex and error-prone
- ***Repeat*** it for every custom middlebox

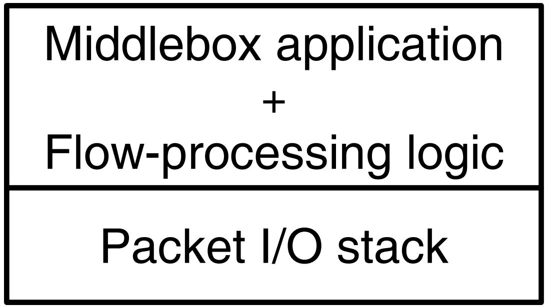
Difference from End-host TCP Applications

- Typical end-host TCP applications



→ Nice abstraction that separates TCP/IP stack from application

- Typical flow-processing middleboxes



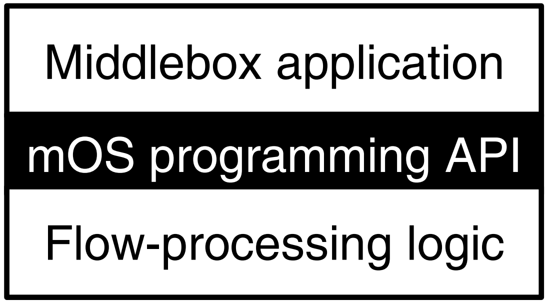
→ Developers build own flow-processing logic from scratch (e.g., on top of PCAP, DPDK, PF_RING)

Our Goal

Build a reusable flow-processing networking stack for modular development of middleboxes

mOS Networking Stack

- A reusable stack for flow-processing middleboxes
    - Abstraction for sub-TCP layer middlebox operations
- Exposes programming abstractions
    - Monitoring sockets abstracting TCP flows
    - Flexible event system
    - Fine-grained resource usage

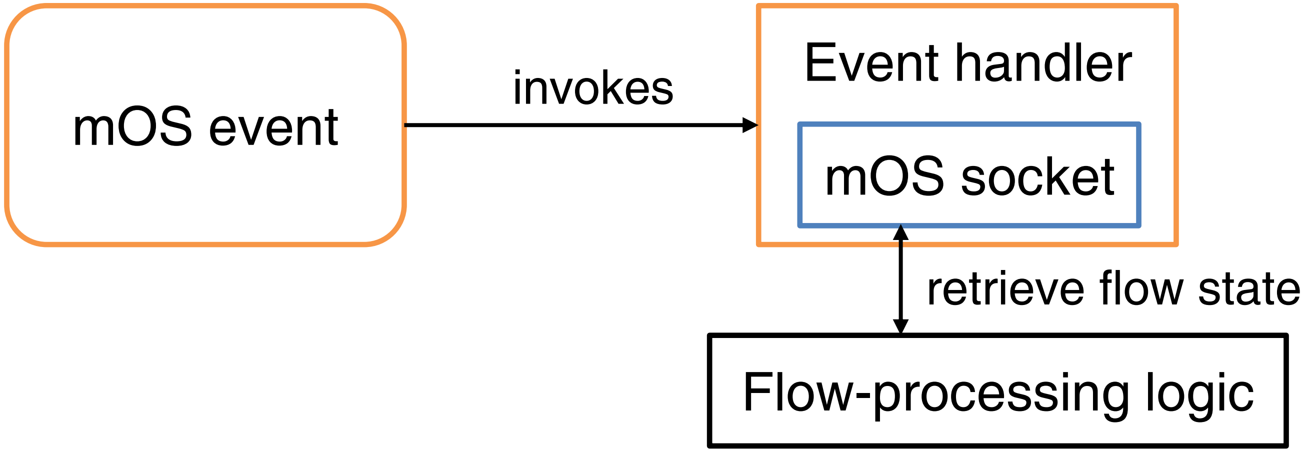


- Benefits
    - Clean, modular development of stateful middleboxes
    - Developers focus on core logic rather than flow management
    - Highly scalable on multi-10Gbps networks

Key Programming Abstractions in mOS

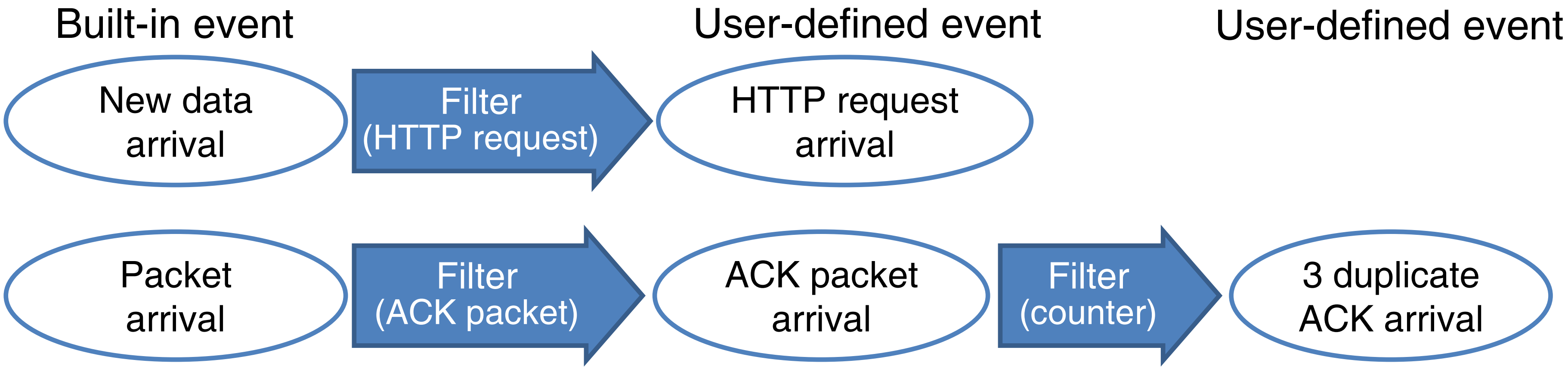
- For better reusability, mOS encourages decomposing a complex application into a set of <event, event handler> pairs
  - One can share a well-designed set of event definitions

- mOS provides two key programming abstractions:
  - mOS events for expressing custom flow-level conditions
  - mOS sockets for retrieving comprehensive flow-level features



Key Abstraction: mOS Events

- Notable condition that merits middlebox processing
- Built-in event (BE)
  - Events that happen naturally in TCP processing
    - e.g., packet arrival, TCP connection start/teardown, retransmission
- User-defined event (UDE)
  - User can define their own event (= base event + filter function)



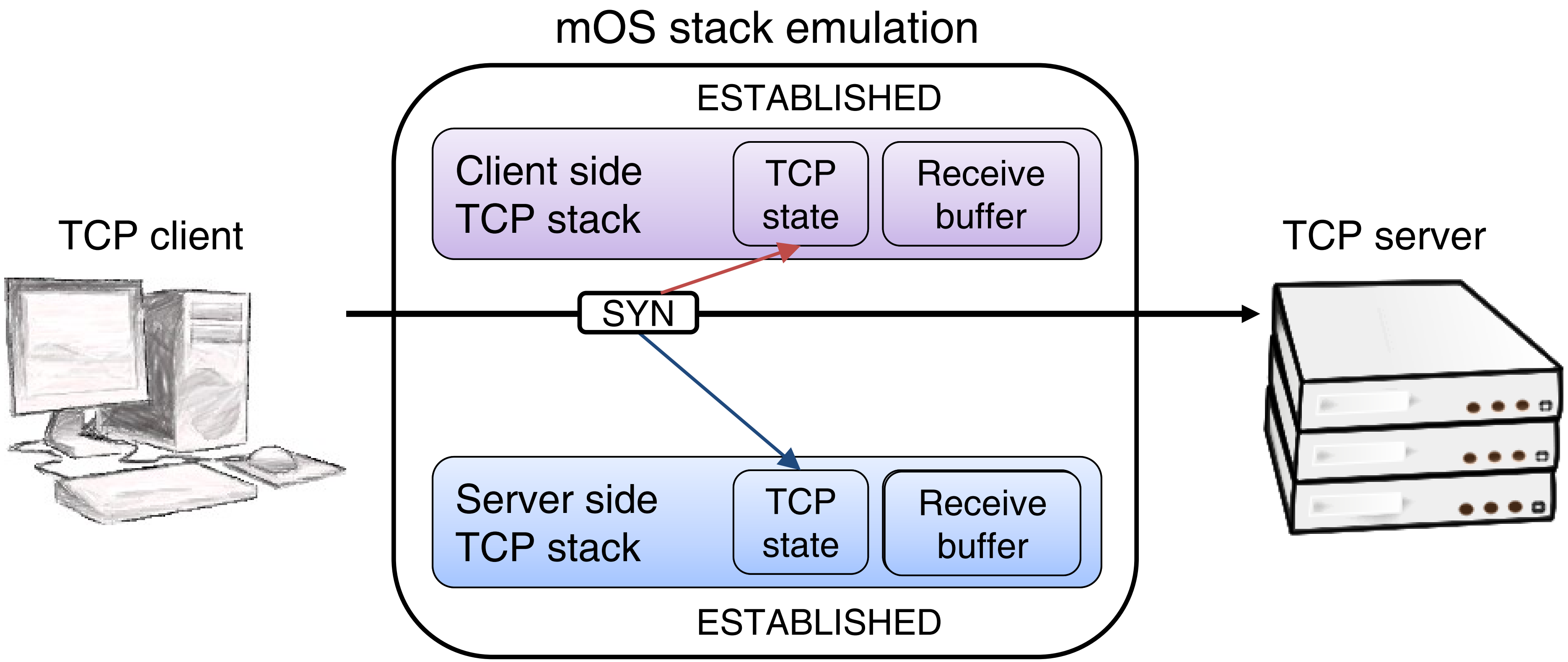
Key Abstraction: mOS Monitoring Socket

- Abstracts a non-terminating ***midpoint*** of a ongoing connection
  - Simultaneously manages the flow states of both end-hosts
  - For every incoming flow, a new mOS monitoring socket is created
- To monitor fine-grained TCP-layer operations and metadata
  - e.g., abnormal packet retransmission, out-of-flow packet arrival, abrupt connection termination, employment of weird TCP/IP options
- Read flow-reassembled data or non-contiguous fragments

- Modify/drop the last packet that raised the event

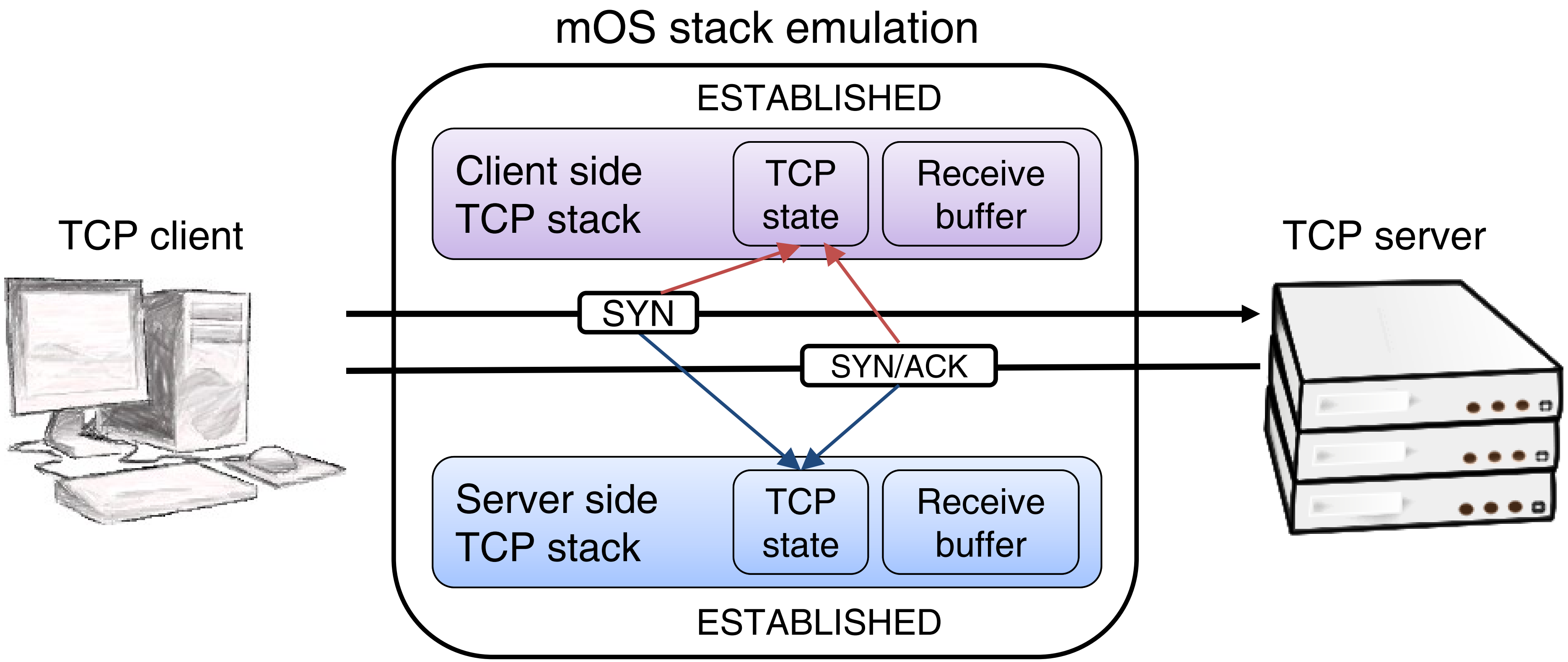
mOS Flow Management

- Dual TCP stack management
  - ***Infer*** the states of both client and server TCP stacks



mOS Flow Management

- Dual TCP stack management
  - ***Infer*** the states of both client and server TCP stacks



mOS stack emulation

TCP client

TCP server

ESTABLISHED

Client side TCP stack

TCP state

Receive buffer

SYN

SYN/ACK

Server side TCP stack

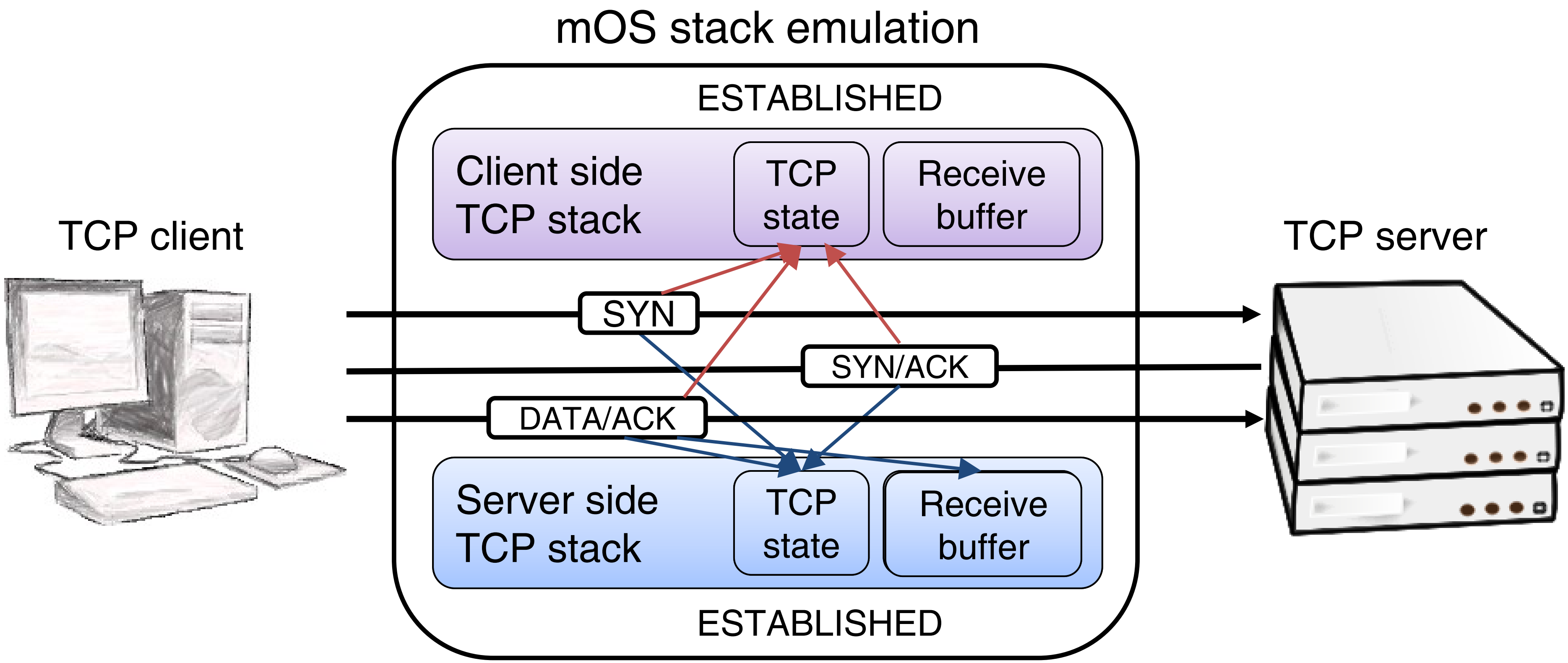
TCP state

Receive buffer

ESTABLISHED

mOS Flow Management

- Dual TCP stack management
  - ***Infer*** the states of both client and server TCP stacks



Scalable mOS Event Management

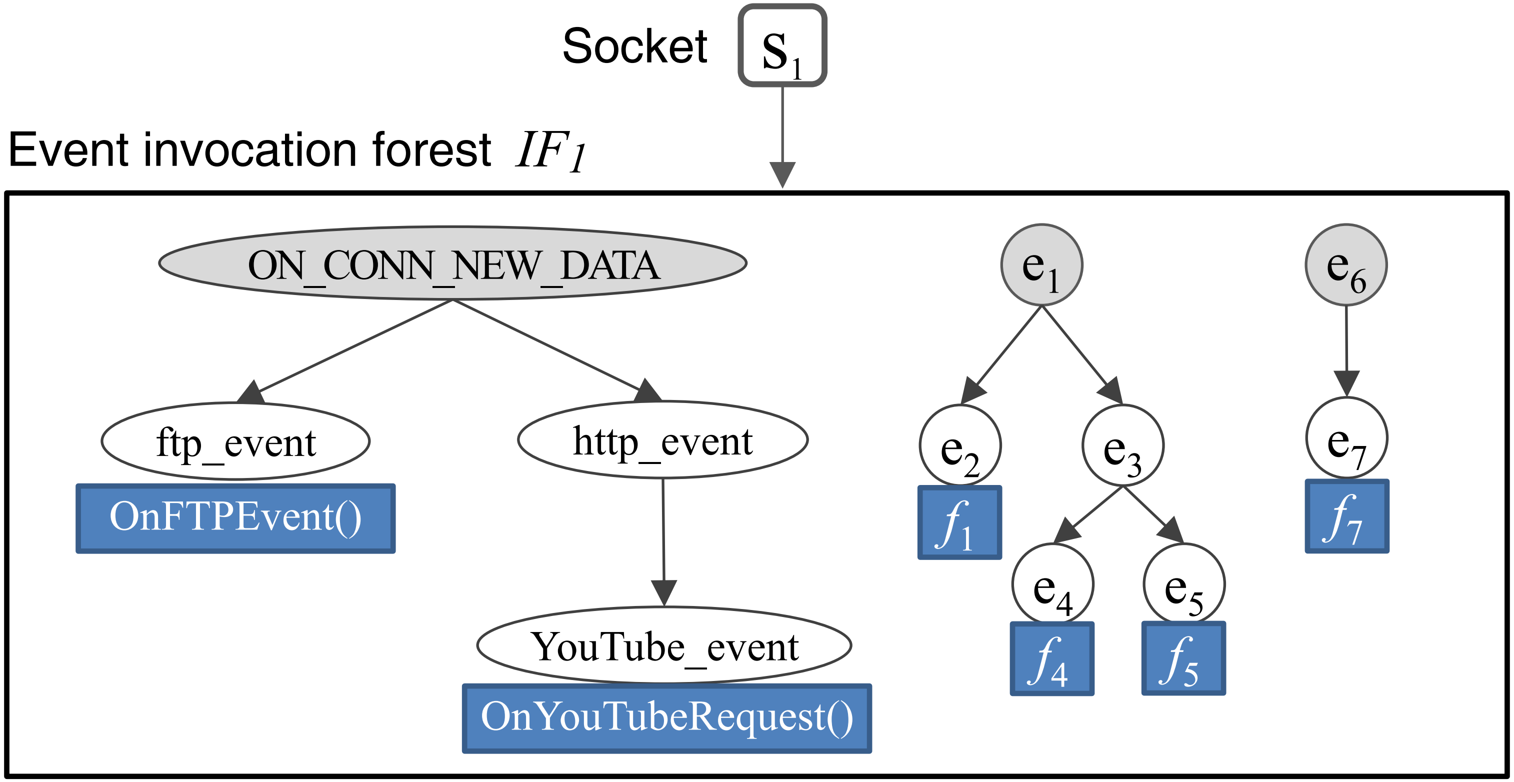
- Each flow can register/change its own set of events dynamically
  - Some flows may add or delete events
  - Some flows may change event handlers for registered events
- Scalability problem
  - How to efficiently manage event sets for 100K+ concurrent flows?
  - Naïve approach suffers from expensive copying of event sets
- Observation: the same event sets are shared by multiple flows
  - Reduces management overhead

Challenge

How to efficiently find/share the same event set?

Data Structures for Event Management

- Each socket points to an <u>event invocation forest</u> that records a set of flow events to wait on

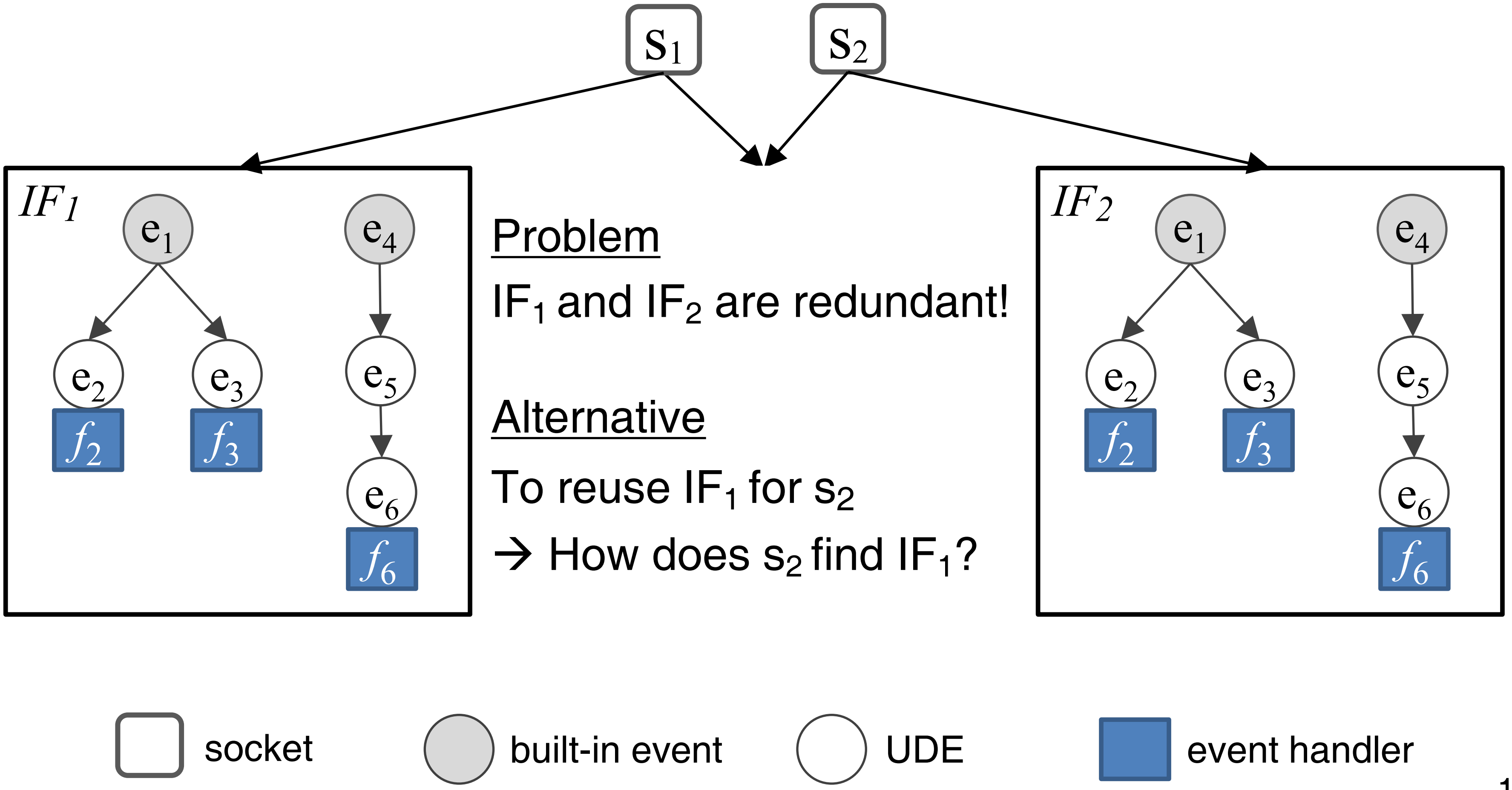


☐ socket ● built-in event ○ UDE ▨ event handler

Dynamic Event Registration Process

Naïve way

1. s_1 registers a new event <e_3, f_3> → IF_1 is created
2. s_2 also registers the same event <e_3, f_3> → IF_2 is created



Problem

IF_1 and IF_2 are redundant!

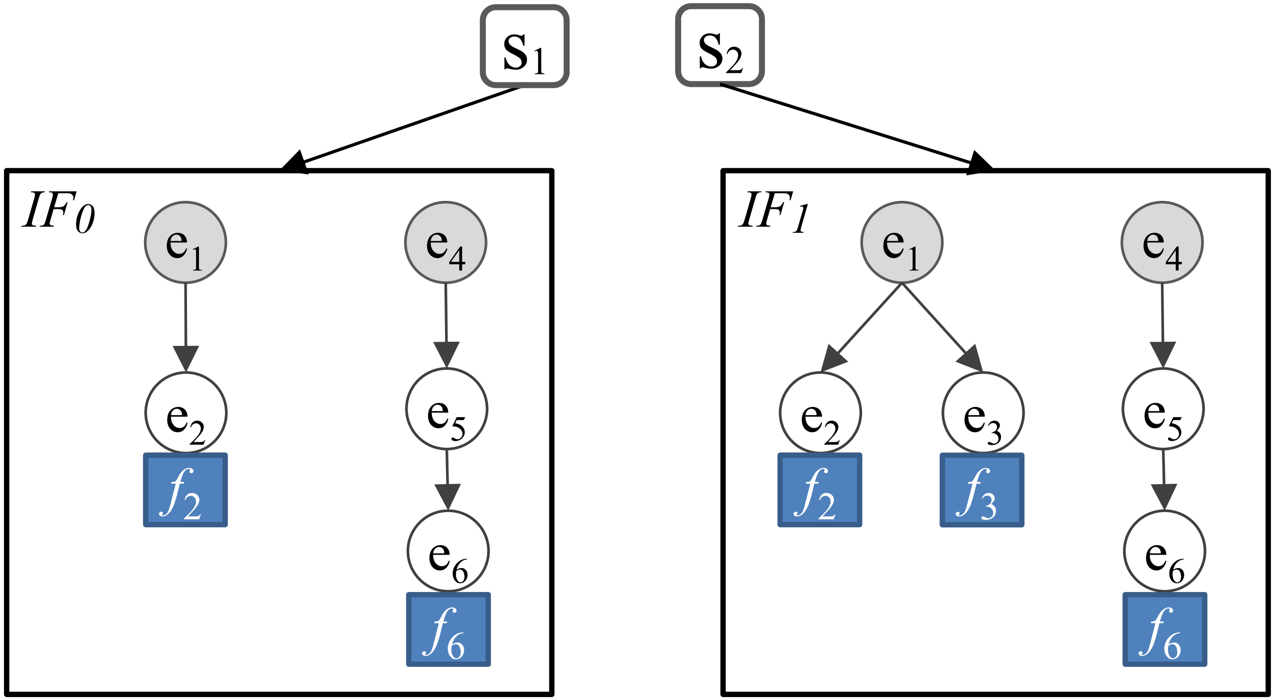
Alternative

To reuse IF_1 for s_2

→ How does s_2 find IF_1?

Efficient Search for Dynamic Registration

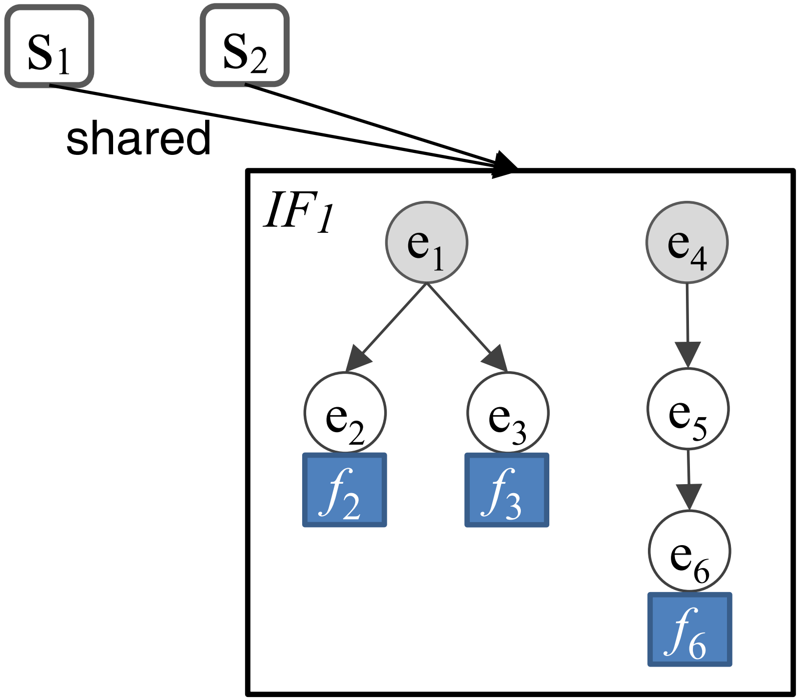
- Each event invocation forest has an ID (searchable via hashtable)
  - id (invocation forest) = XOR sum of hash (event + event handler)
- New invocation forest id after adding or deleting <e, f> from t
  - id (new forest) = id (old forest) $\oplus$ hash (e + f)



Efficient Search for Dynamic Registration

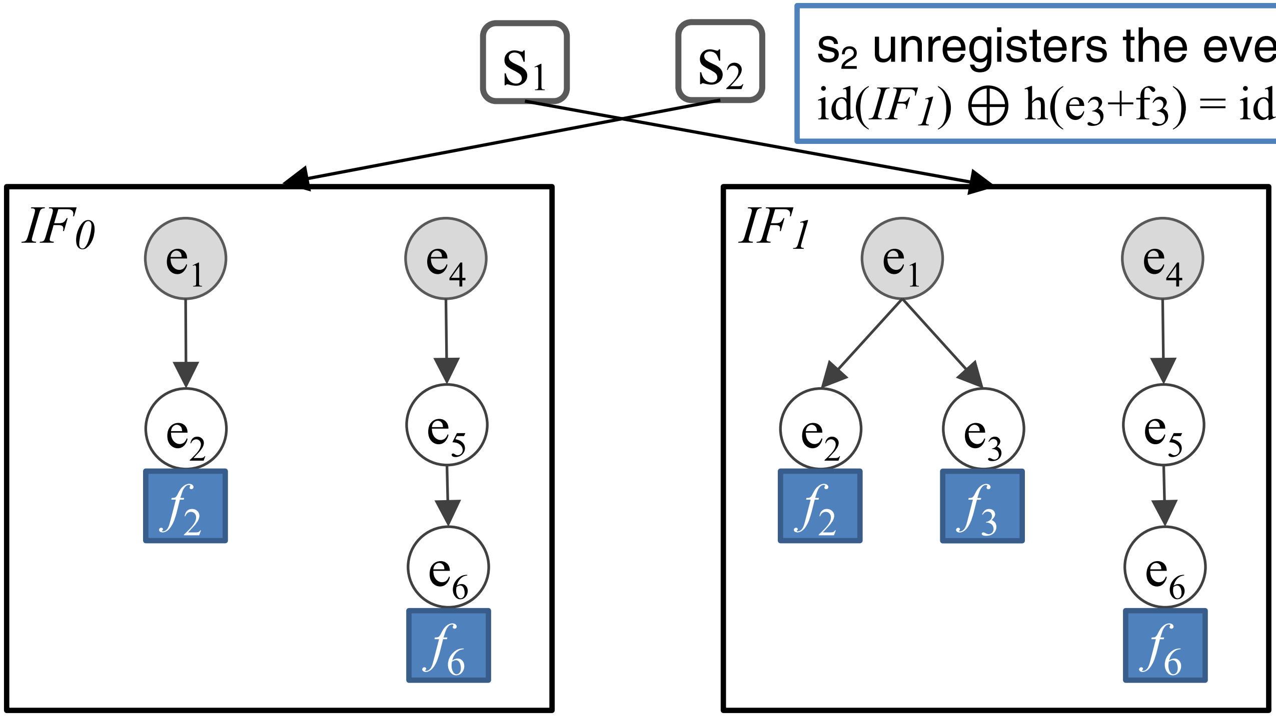
- Each event invocation forest has an ID (searchable via hashtable)
  - id (invocation forest) = XOR sum of hash (event + event handler)
- New invocation forest id after adding or deleting <e, f> from t
  - id (new forest) = id (old forest) $\oplus$ hash (e + f)

s_1 registers a new event $<e_3, f_3>$
$id(IF_0) \oplus h(e_3+f_3) = id(\boldsymbol{IF_1})$



Efficient Search for Dynamic Registration

- Each event invocation forest has an ID (searchable via hashtable)
  - id (invocation forest) = XOR sum of hash (event + event handler)
- New invocation forest id after adding or deleting <e, f> from t
  - id (new forest) = id (old forest) $\oplus$ hash (e + f)

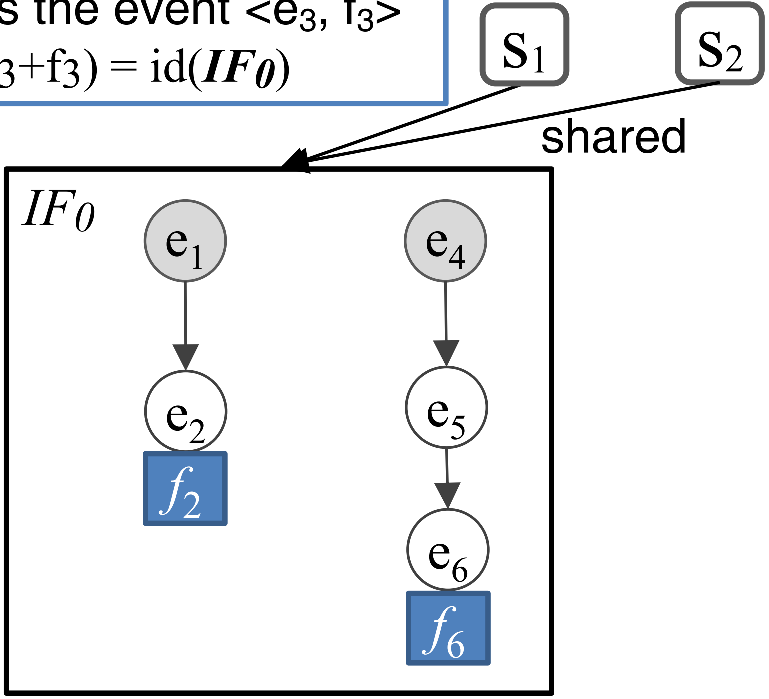


s_2 unregisters the event $<e_3, f_3>$
$id(IF_1) \oplus h(e_3+f_3) = id(\boldsymbol{IF_0})$

Efficient Search for Dynamic Registration

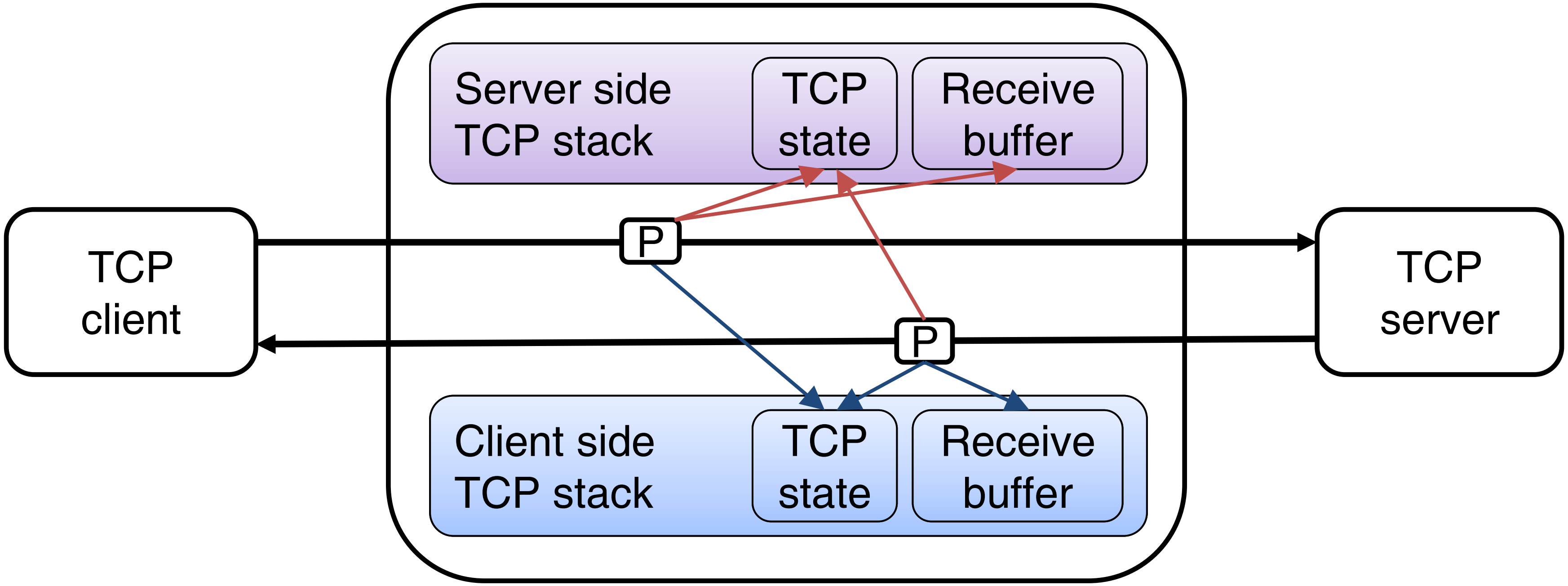
- Each event invocation forest has an ID (searchable via hashtable)
  - id (invocation forest) = XOR sum of hash (event + event handler)
- New invocation forest id after adding or deleting <e, f> from t
  - id (new forest) = id (old forest) $\oplus$ hash (e + f)

s_1 unregisters the event $<e_3, f_3>$
$id(IF_1) \oplus h(e_3+f_3) = id(\mathbf{IF_0})$



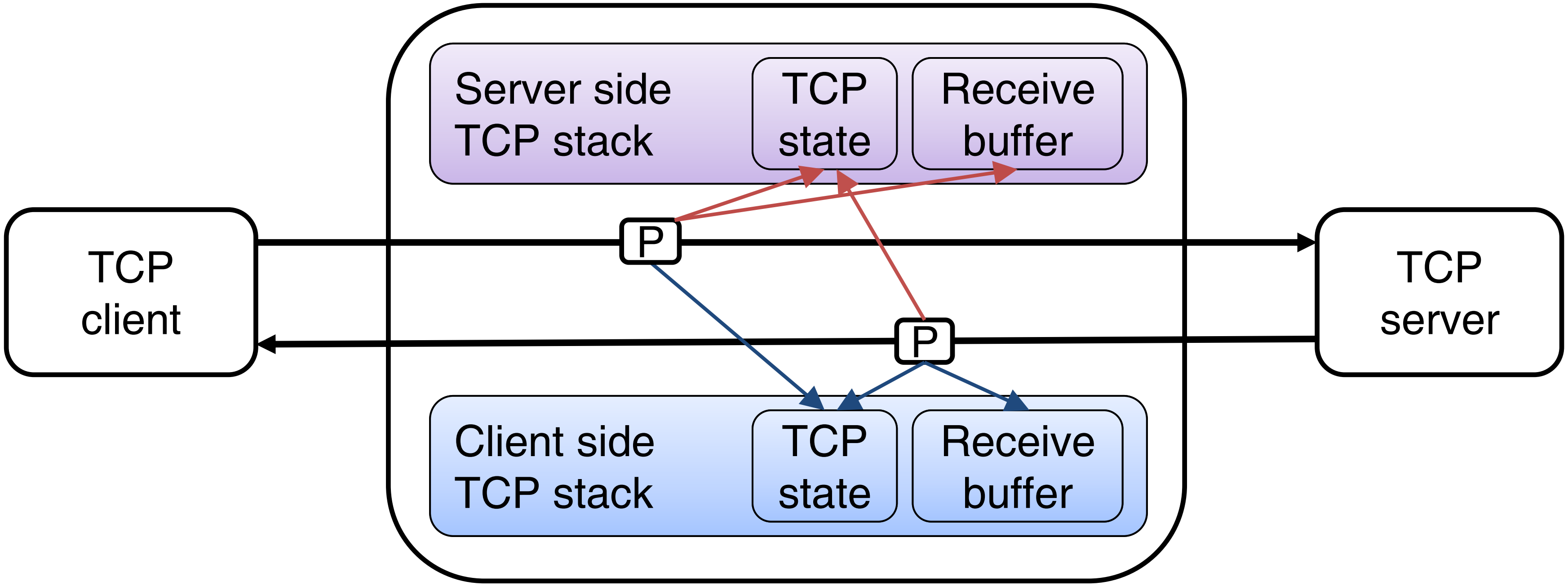
Fine-grained Resource Management in mOS

- Not all middleboxes require full features
  - Some middleboxes do not require flow reassembly



Fine-grained Resource Management in mOS

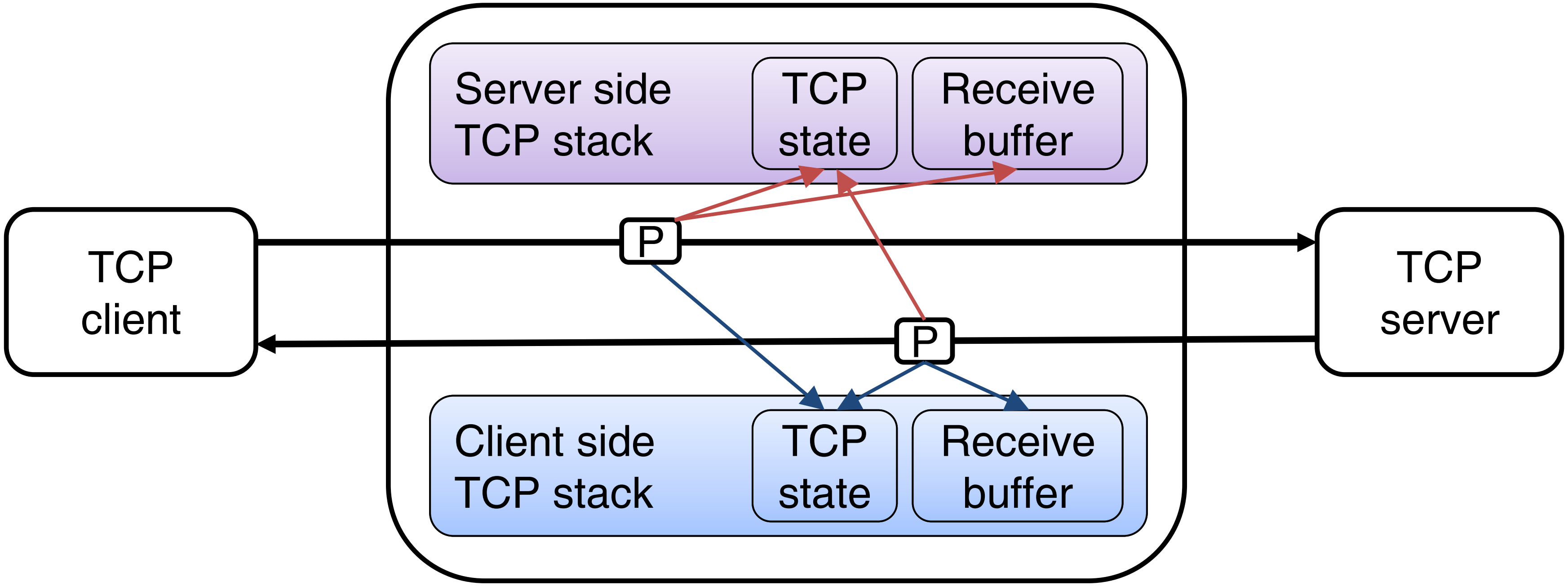
- Not all middleboxes require full features
    - Some middleboxes do not require flow reassembly
    - Some middleboxes monitor only client-side data



Fine-grained Resource Management in mOS

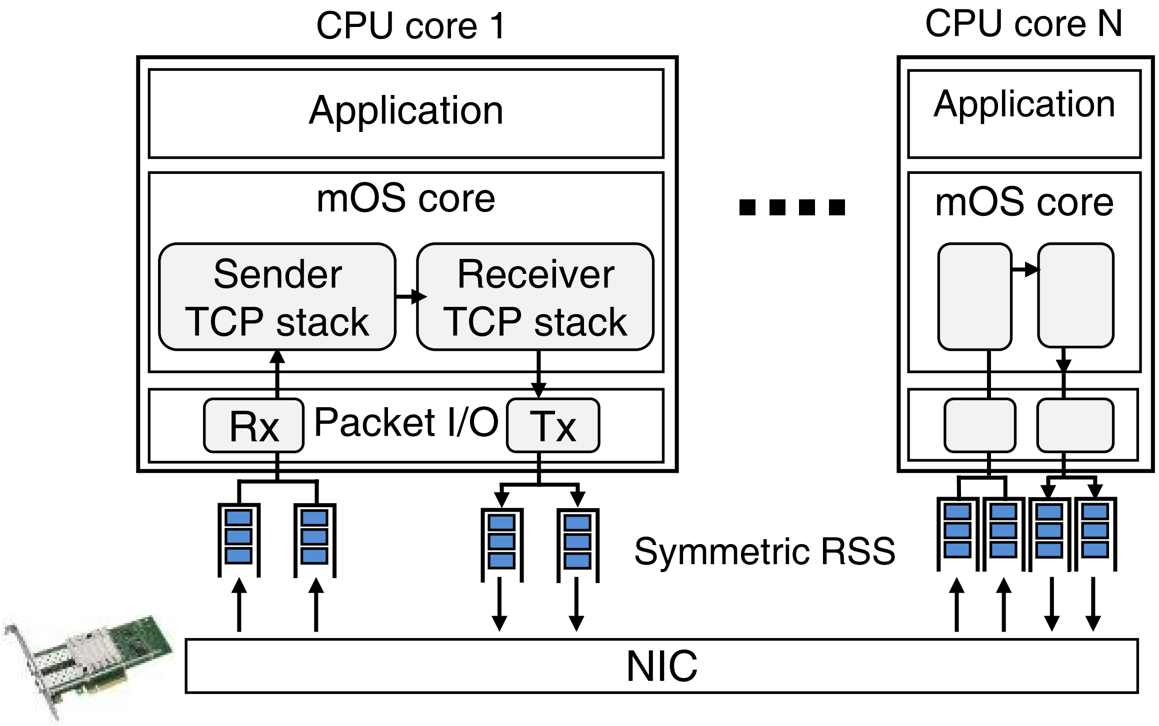
- Not all middleboxes require full features
  - Some middleboxes do not require flow reassembly
  - Some middleboxes monitor only client-side data
  - No more monitoring after handling certain events

Global or per-flow manipulation



mOS Stack Implementation

- Per-thread library TCP stack
    - ~26K lines of C code (mTCP [1] : ~11K lines)
- Shared nothing parallel architecture



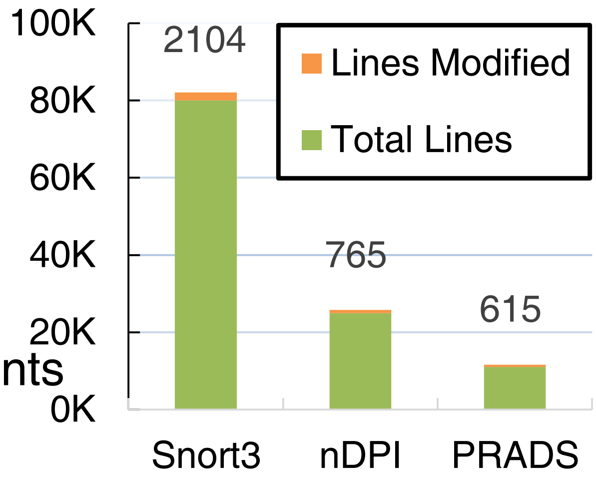
[1] “mTCP: a highly scalable user-level TCP stack for multicore systems”, NSDI'14

Evaluation

1. Does mOS API support diverse middlebox applications?
2. Does mOS promise high performance?

mOS API Evaluation

- Does the API support diverse range of middleboxes?
  - Snort3 (strip ~10K lines)
    - Snort with mOS flow management
    - Replaces HTTP/TCP inspection module
  - nDPI
    - L7 protocol parsing over flow content
  - PRADS
    - Signature pattern matching on flow content
- Lessons learnt
  - mOS simplifies code
  - mOS patches vulnerabilities (nDPI/PRADS)
    - Detects signature that spans multiple segments
  - mOS does not degrade performance
    - Perform on par with respective vanilla (DPDK) versions



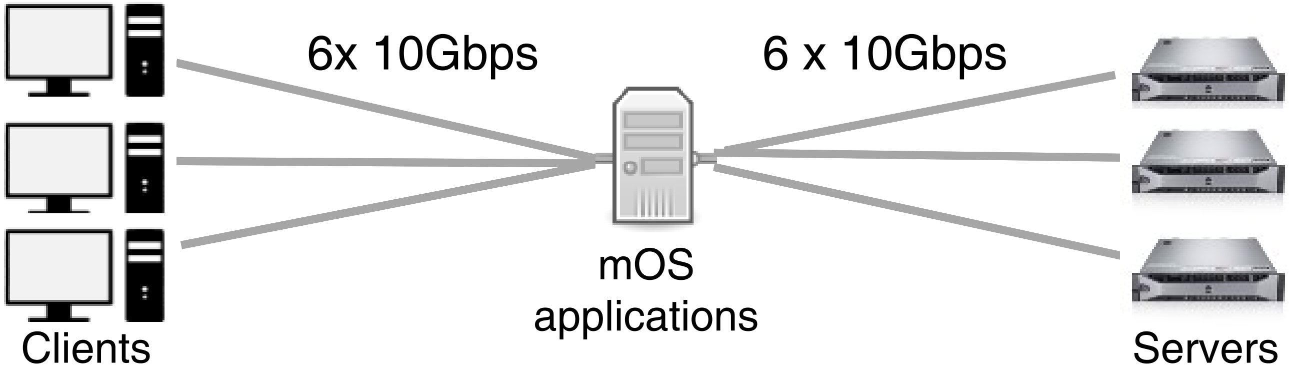
mOS API Evaluation (cont.)

- Does the API support diverse range of middleboxes?
  - Halfback proxy (128 lines)
    - Low latency proxy with proactive TCP retransmissions
  - Abacus (561 lines vs 4,091 lines)
    - Secure cellular data accounting system
  - Parallel NAT
    - High performance NAT
  - Midstat
    - netstat for middleboxes
  - L4 firewall
  - Etc.

- Applications ported to mOS: ~9x code line reduction

Performance Evaluation

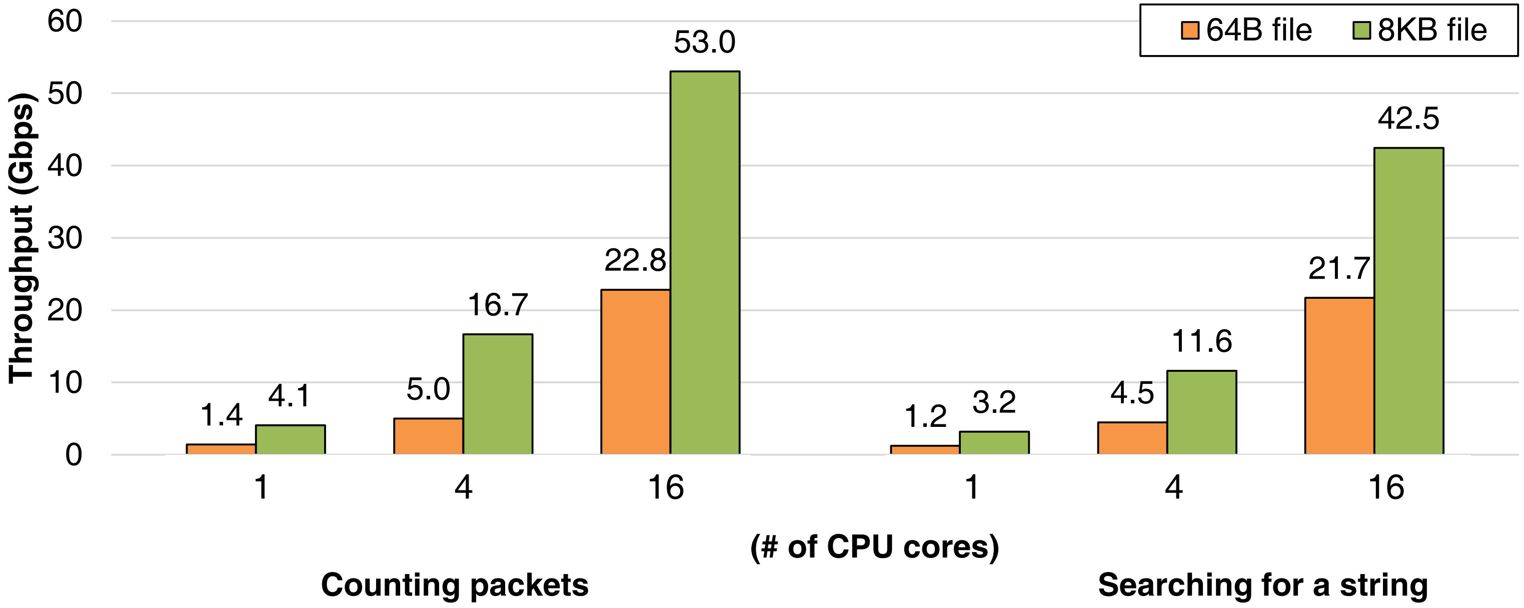
- Does mOS provide high performance?



- mOS applications in **inline** mode
  - Flow management and forwarding packets by their flows
  - 2 x Intel E5-2690 (16 cores, 2.9 GHz), 20 MB L3 cache size,
  - 132 GB RAM, 6 x 10 Gbps NICs
- Six pairs of clients and servers: 60 Gbps max
  - Intel E3-1220 v3 (4 cores, 3.1 GHz), 8 MB L3 cache size
  - 16 GB RAM, 1 x 10 Gbps NIC per machine

Performance Scalability on Multicores

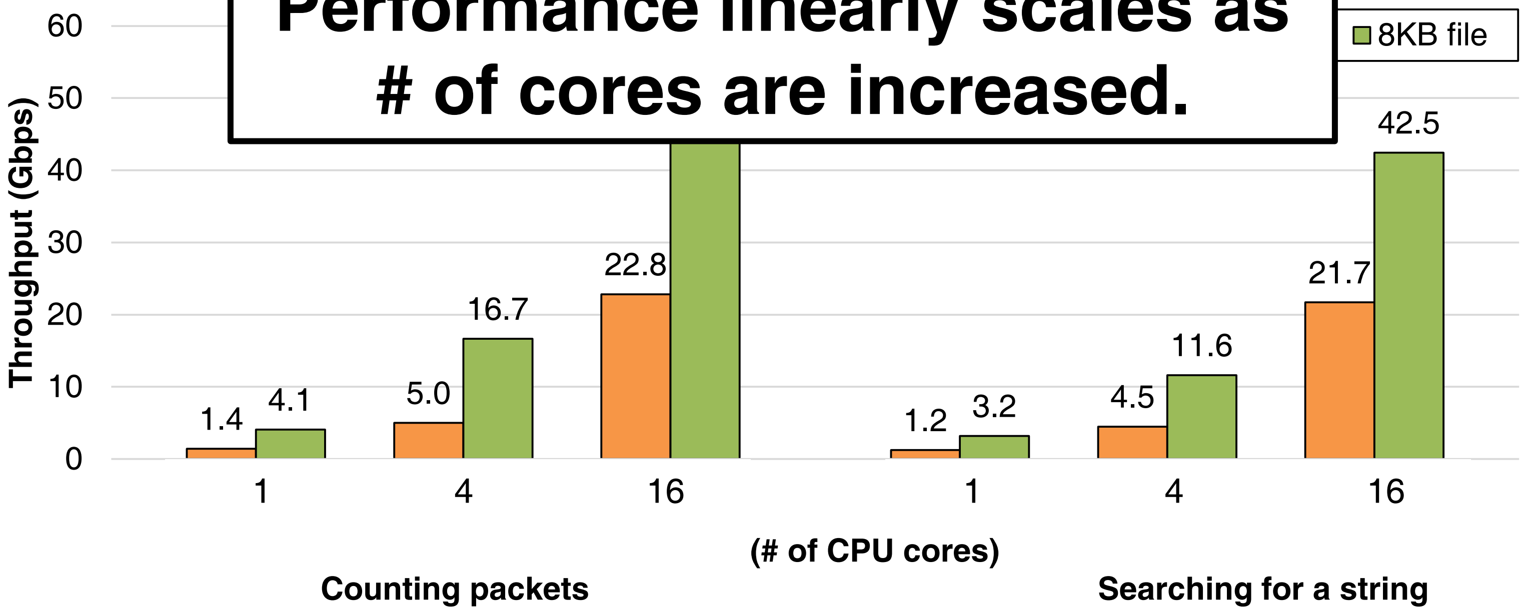
- File download traffic with 192K concurrent flows
  - Each flow downloads an X-byte content in one TCP connection
  - A new flow is spawned when a flow terminates
- Two simple applications
  - Counting packets per flow (packet arrival event)
  - Searching for a string in flow reassembled data (full flow reassembly & DPI)



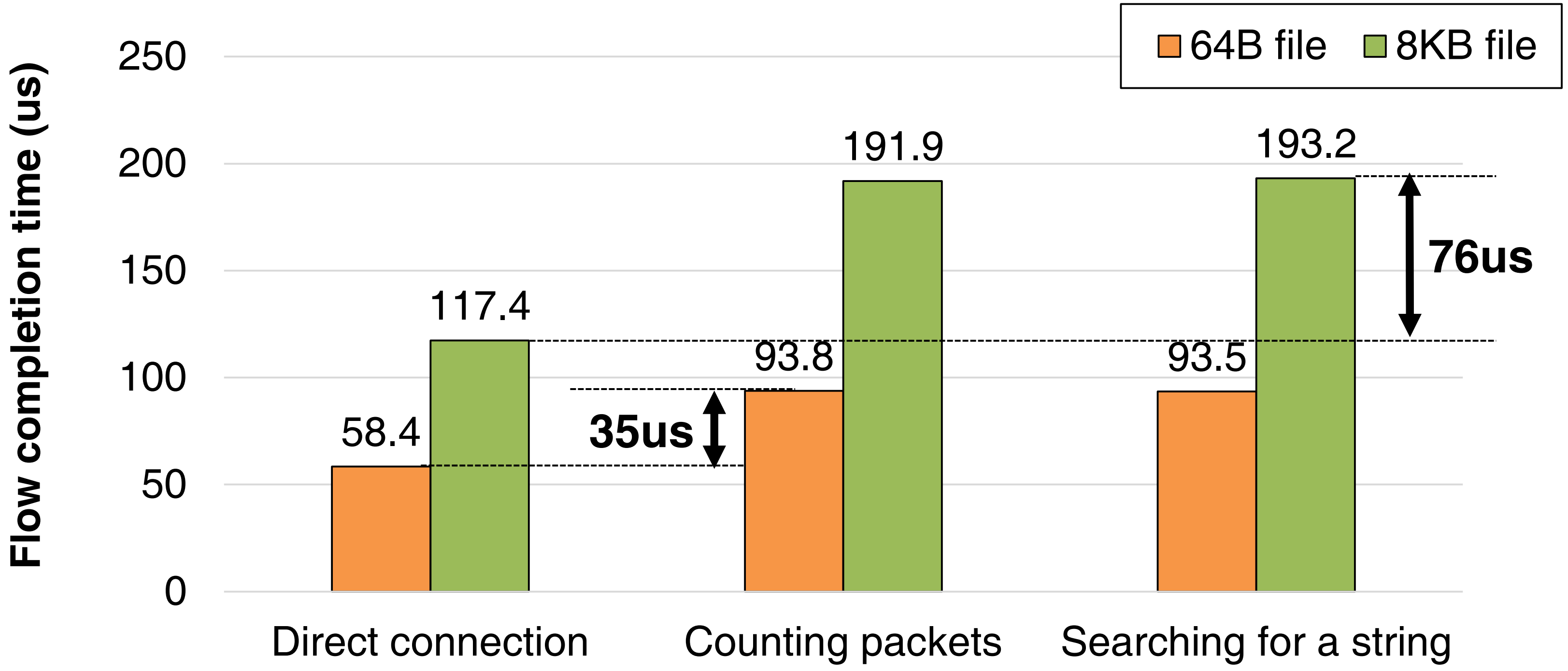
Performance Scalability on Multicores

- File download traffic with 192K concurrent flows
  - Each flow downloads an X-byte content in one TCP connection
  - A new flow is spawned when a flow terminates
- Two simple applications
  - Counting packets per flow (packet arrival event)
  - Searching for a string in flow reassembled data (full flow reassembly & DPI)

Performance linearly scales as # of cores are increased.

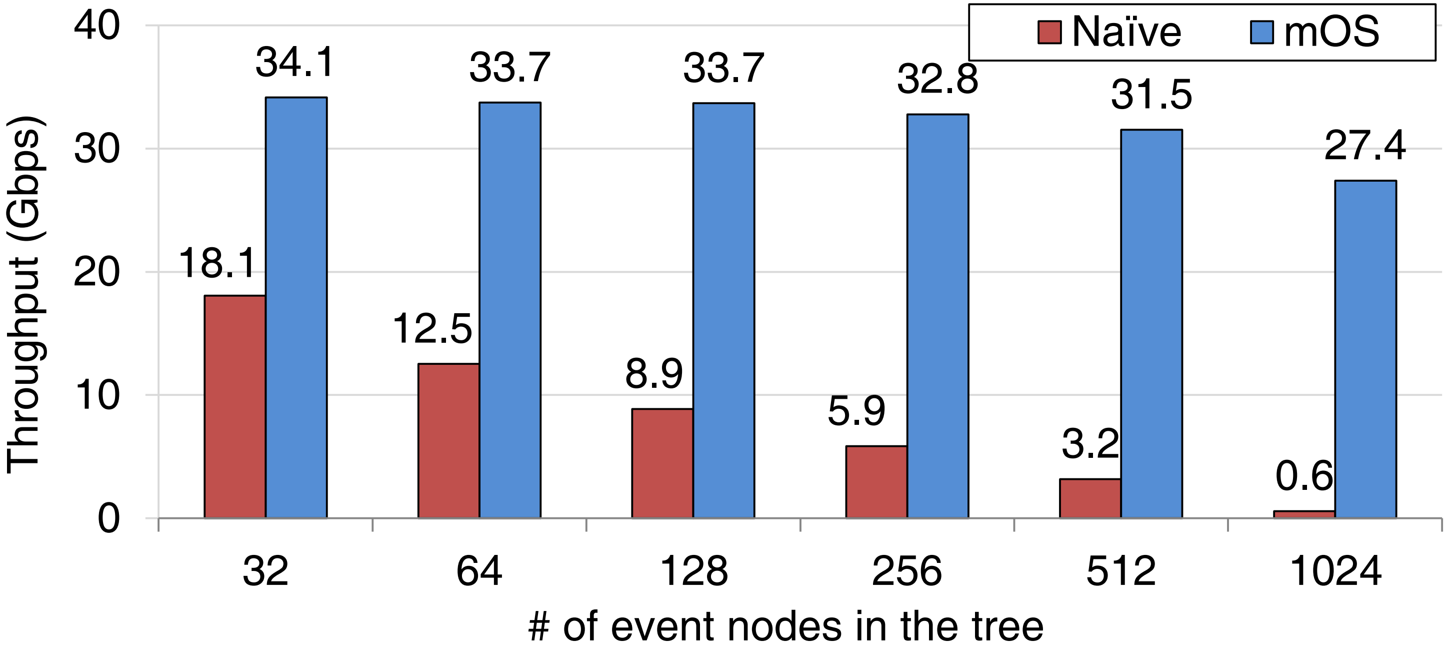


Latency overhead by mOS applications



Dynamic Event Registration Evaluation

- Monitor 192K concurrent flows
  - Flow size: 4KB
- Searching for a string in flow reassembled data
  - Dynamically register a new event when target string found
  - 50% client flows have target strings



Conclusion

- Software-based middleboxes have:
  - Modularity issues
  - Readability issues
  - Maintainability issues

➢ mOS stack: reusable networking stack for middleboxes
  - Programming abstraction with socket-based API
  - Event-driven middlebox processing
  - Efficient resource usage with dynamic resource composition

- mOS stack/API available @:

  https://github.com/ndsl-kaist/mOS-networking-stack

Thank You

Questions?

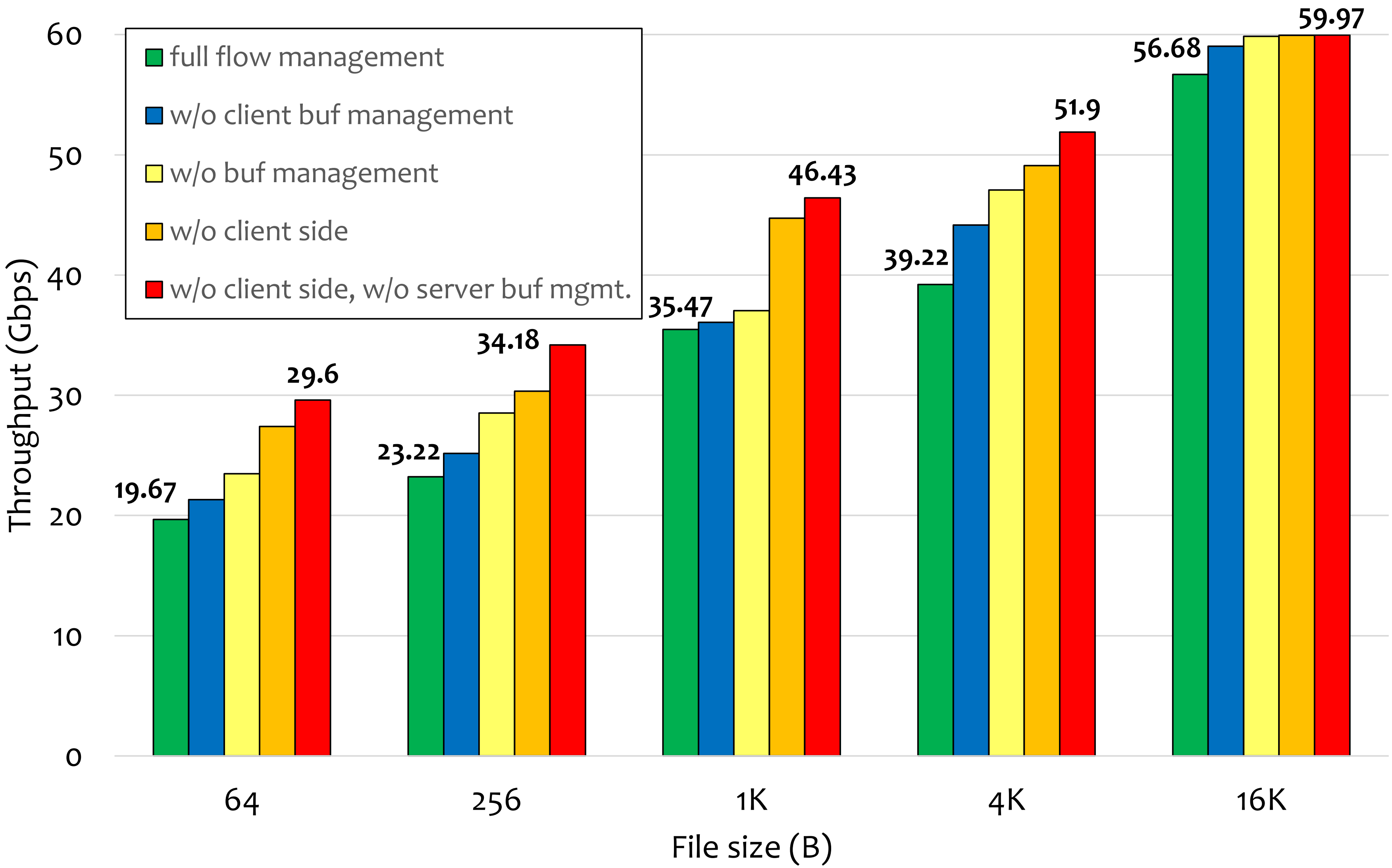
http://mos.kaist.edu/

https://github.com/ndsl-kaist/mOS-networking-stack

Appendix

Extra Slides

Performance under Selective Resource Consumption



Real applications performance

Application	original + pcap	original + DPDK	mOS port
Snort-AC	0.57 Gbps	8.18 Gbps	9.17 Gbps
Snort-DFC	0.82 Gbps	14.42 Gbps	15.21 Gbps
nDPIReader	0.66 Gbps	28.92 Gbps	28.87 Gbps
PRADS	0.42 Gbps	2.03 Gbps	1.90 Gbps

- Workload: real LTE packet trace (~67 GB)
- 4.5x ~ 28.9x performance improvement
- mOS brings code modularity & correct flow management

Events & Available Hooks

- Stream monitoring socket

Built-in event	MOS_HK_SND	MOS_HK_RCV
`MOS_ON_PKT_IN`	O	O
`MOS_ON_CONN_START`	O	O
`MOS_ON_CONN_END`	O	O
`MOS_ON_TCP_STATE_CHANGE`	O	O
`MOS_ON_REXMIT`	O	O
`MOS_ON_CONN_NEW_DATA`	X	X
`MOS_ON_ORPHAN`	X	X

- Raw monitoring socket

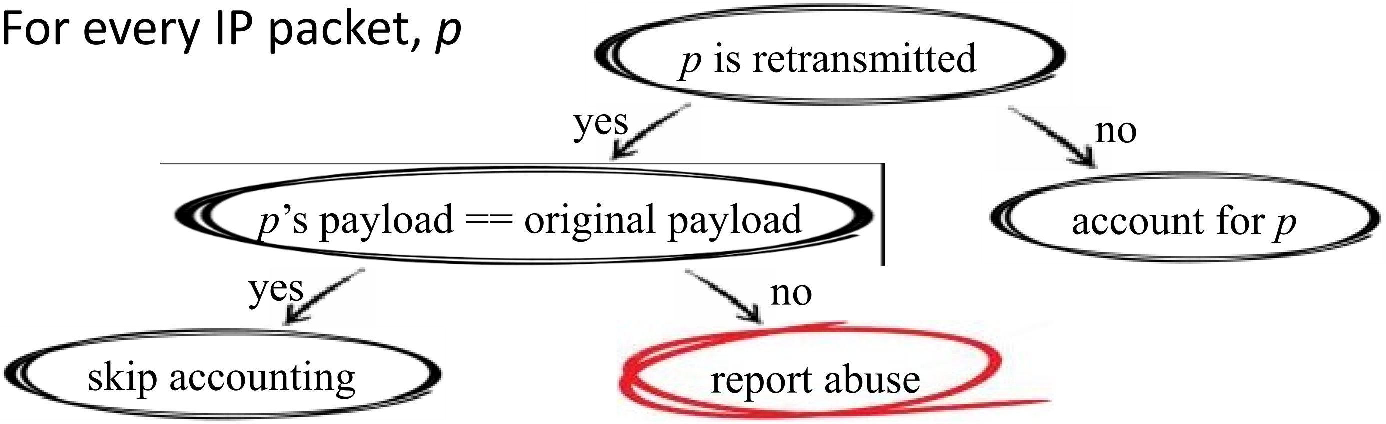
Built-in event	MOS_HK_SND	MOS_HK_RCV
`MOS_ON_PKT_IN`	X	X

Cellular Accounting with mOS Networking Stack

Core Logic +
Flow Mgmt

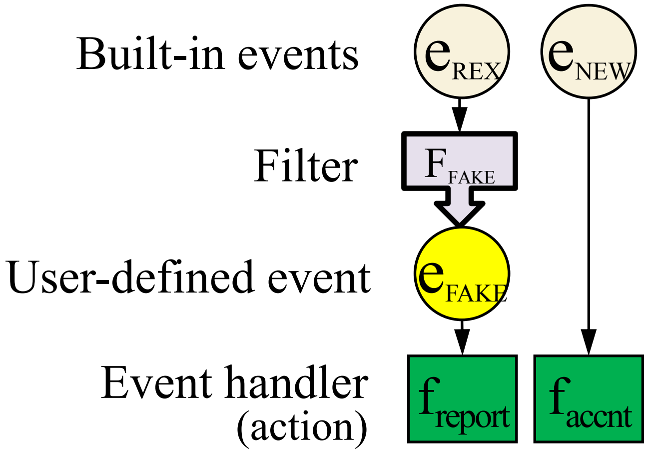
4,639
LoC

For every IP packet, *p*



Event-action

561
LoC



e_{REX}	MOS_ON_REXMIT
e_{NEW}	MOS_ON_CONN_NEW_DATA
F_{FAKE}	IsFakeRexmit()
e_{FAKE}	UDE_FAKE_REXMIT
f_{report}	ReportAbuse()
f_{accnt}	AccountDataUsage()

F_{FAKE} filter function ○ built-in event ● UDE 🟩 event handler