mOS Networking Stack: a Specialized Network Programming Library for Stateful Middleboxes

http://mos.kaist.edu/

KYOUNGSOO PARK

ASIM JAMSHED, YOUNGGYUN MOON, DONGHWI KIM & DONGSU HAN

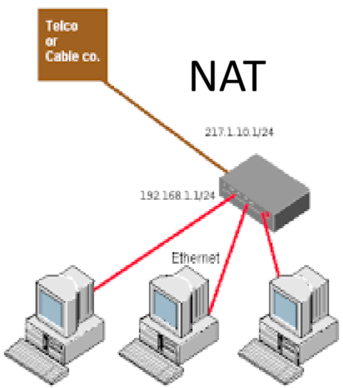
SCHOOL OF ELECTRICAL ENGINEERING, KAIST

Network Middlebox

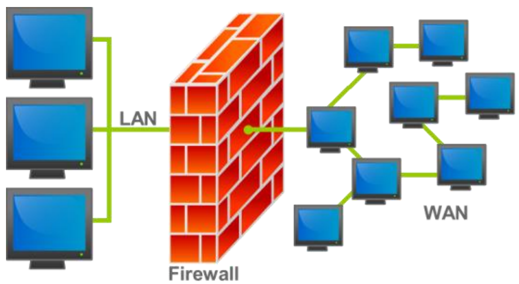
Networking devices that provide **extra** functionalities

- Switches/routers = L2/L3 devices
- All others are called middleboxes

NAT



Firewalls



Web/SSL proxies

IDS/IPS

SNORT®

SURICATA

L7 protocol analyzers

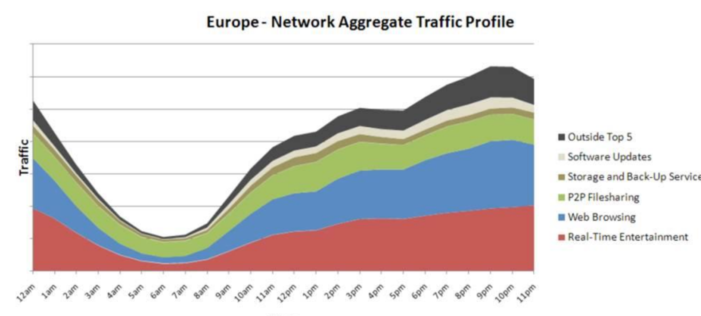


Figure 1. Network aggregate traffic profile. Source: Sandvine

Middleboxes are Increasingly Popular

Middleboxes are ubiquitous

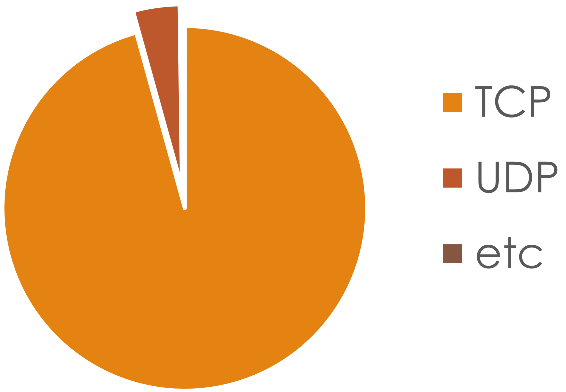
- Number of middleboxes =~ number of routers (Enterprise)
- Prevalent in cellular networks (e.g., NAT, firewalls, IDS/IPS)
- Network functions virtualization (NFV)
- SDN controls network functions

Provides key functionalities in modern networks

- Security, caching, load balancing, etc.
- Because original Internet design lacks many features

Most Middleboxes Deal with TCP Traffic

- TCP dominates the Internet
  - 95+% of traffic is TCP [1]
- Flow-processing middleboxes
  - Stateful firewalls
  - Protocol analyzers
  - Cellular data accounting
  - Intrusion detection/prevention systems
  - Network address translation
  - And many others!



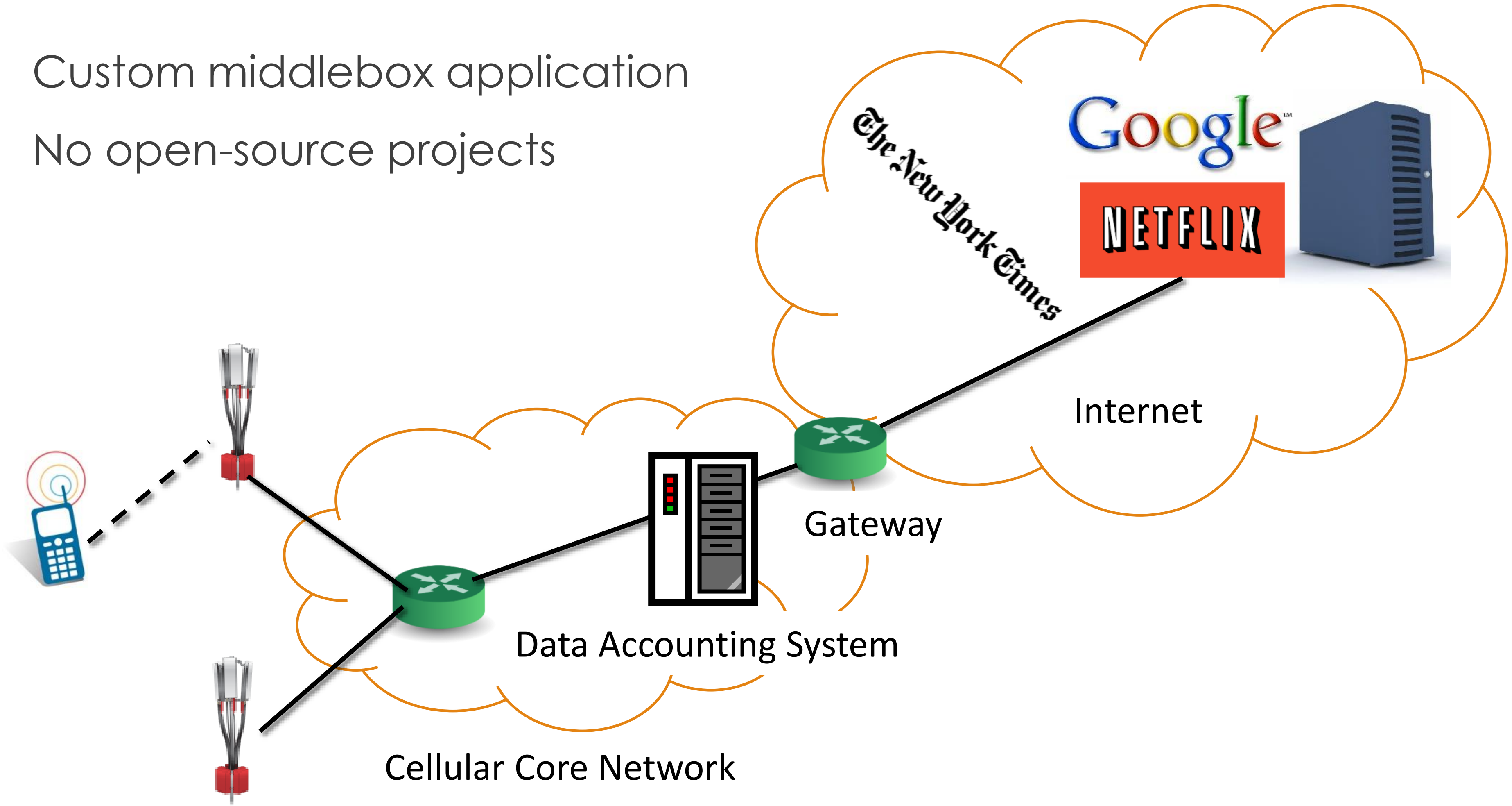
[1] "Comparison of Caching Strategies in Modern Cellular Backhaul Networks", ACM MobiSys 2013.

TCP state management is complex and error-prone!

Example: Cellular Data Accounting System

Custom middlebox application

No open-source projects



Develop Cellular Data Accounting System

```
For every IP packet, p

sub = FindSubscriber(p.srcIP, p.destIP);

sub.usage += p.length;
```

Charge for retransmission?

South Korea

```
For every IP packet, p
 if (p is not retransmitted){
   sub = FindSubscriber(p.srcIP, p.destIP);
   sub.usage += p.length;
 }
```

TCP tunneling attack? [NDSS'14]

Attack Detection

```
For every IP packet, p
 if (p is not retransmitted){
   sub = FindSubscriber(p.srcIP, p.destIP);
   sub.usage += p.length;
 } else { // if p is retransmitted
   if (p's payload != original payload) {
         report abuse by the subscriber;
   }
 }
```

Logically, simple process!

Cellular Data Accounting Middlebox

Core logic

- Determine if a packet is retransmitted
- Remember the original payload (e.g, by sampling)
- Key: TCP flow management

How to implement?

- Borrow code from open-source IDS (e.g., Snort/Suricata)
- Problem: 50~100K code lines tightly coupled with their IDS logic

Another option?

- Borrow code from open-source kernel (e.g., Linux/FreeBSD)
- Problem: kernel is for one end, so it lacks middlebox semantics

What is the common practice? state-of-the-art?

- Implement your own flow management
- Problem: repeat it for every custom middlebox

Programming TCP End-Host Application

- Typical TCP end-host applications

TCP application | User level
Berkeley Socket API
TCP/IP stack | Kernel level

- Typical TCP middleboxes?

  - Middlebox logic
  - Packet processing
  - Flow tracking
  - Flow reassembly
  - Spaghetti code?

No clear separation!

Berkeley socket API
- Nice abstraction that separates flow management from application
- Write better code if you know TCP internals
- ***Never*** requires you to write *TCP stack itself*

mOS Networking Stack

Reusable networking stack for middleboxes

- Programming abstraction and APIs to developers

Key concepts

- Separation of flow management from custom logic
- Event-based middlebox development (event/action)
- Per-flow flexible resource consumption

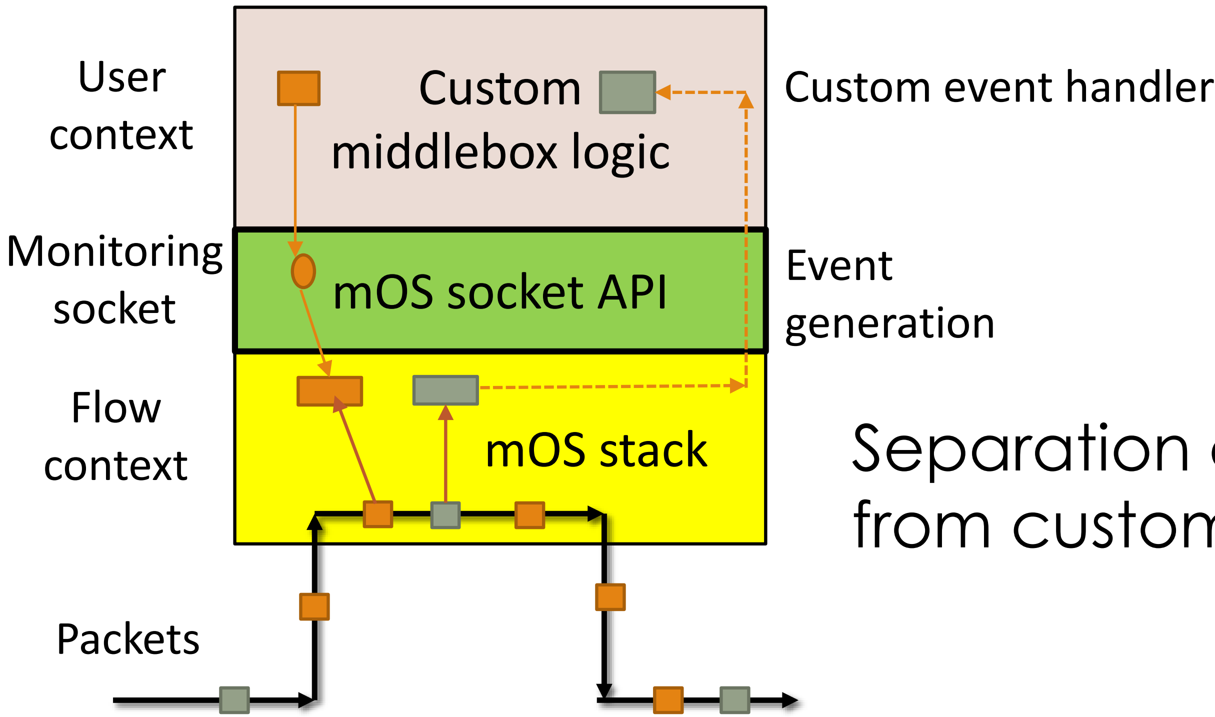
Benefits

- Clean, modular development of stateful middleboxes
- Developers focus on core logic rather than flow management
- High performance flow management on mTCP stack

Key Abstraction: mOS Monitoring Socket

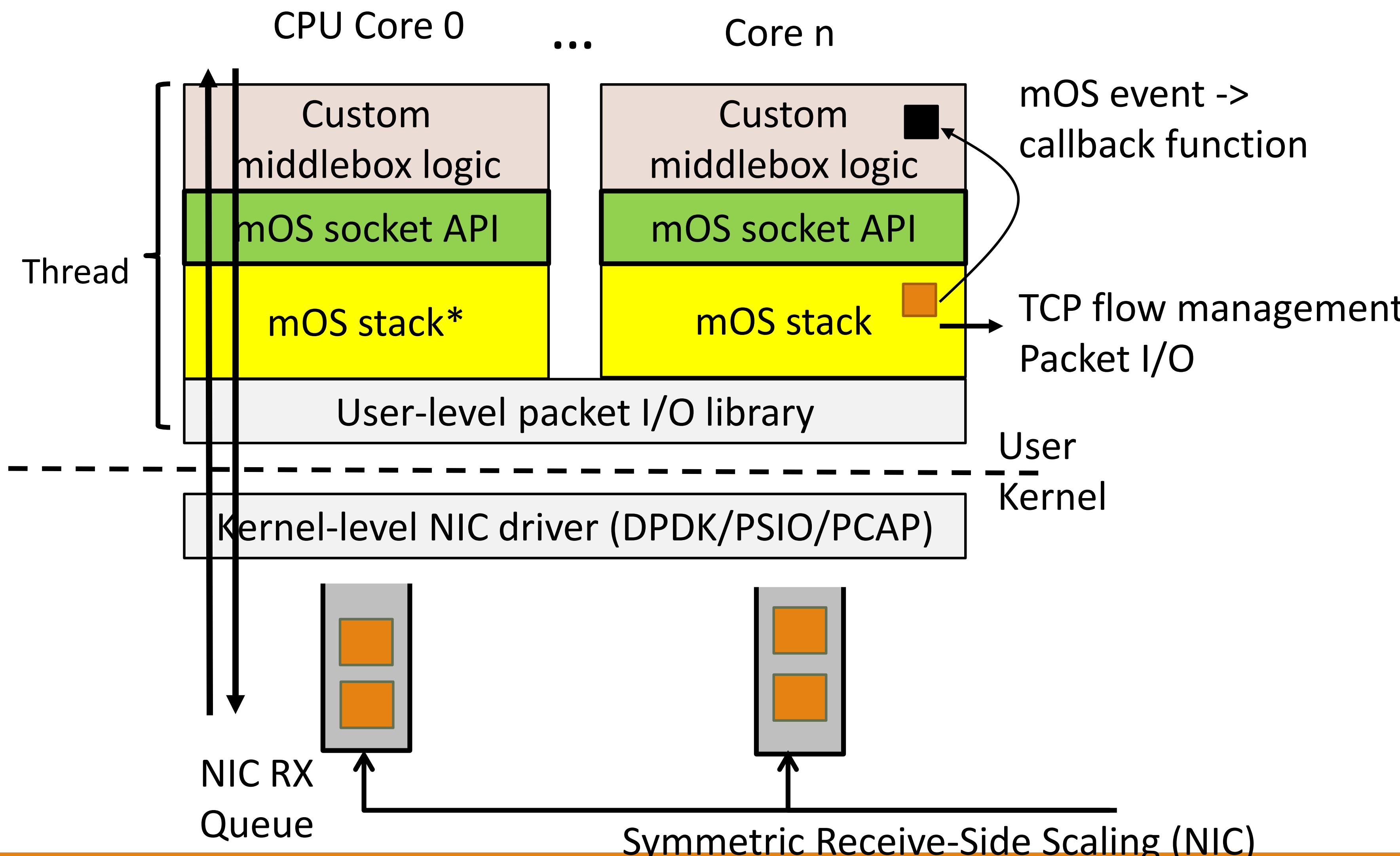
Represents the middlebox viewpoint on network traffic

- Monitors both TCP connections and IP packets
- Provides similar API to the Berkeley socket API

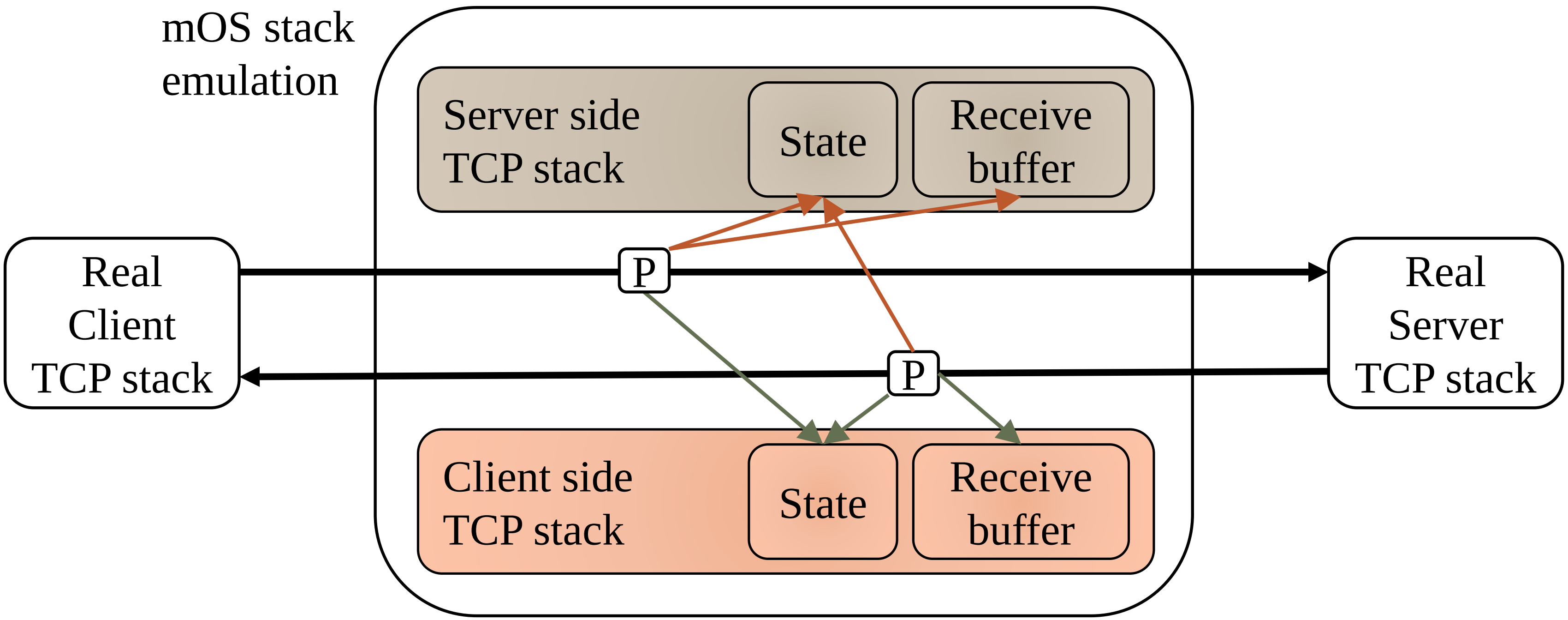


Separation of flow management from custom middlebox logic!

Shared-Nothing Parallel Architecture



mOS Flow Management



Dual TCP stack management

- Track the TCP states of both client and server TCP stacks

Example: a client sends a SYN packet

- Client-side state changes from CLOSED to SYN_SENT
- Server-side state changes from LISTEN to SYN_RECEIVED

mOS Event

Notable condition that merits middlebox processing
- Different from TCP socket events

Built-in event (BE)
- Events that happen naturally in TCP processing
- e.g., packet arrival, TCP connection start/teardown, retransmission, etc.

User-defined event (UDE)
- User can define their own event
- UDE = base event + filter function
  - Raised when base event triggers and filter evaluates to TRUE
  - Nested event: base event can be either BE or UDE
  - e.g., HTTP request, 3 duplicate ACKs, malicious retransmission

Middlebox logic = a set of <event, event handler> tuples

Sample Code: Initialization

```
static void
thread_init(mctx_t mctx)
{
  monitor_filter ft ={0};
  int msock; event_t http_event;

  msock = mtcp_socket(mctx, AF_INET, MOS_SOCK_MONITOR_STREAM, 0);

  ft.stream_syn_filter = "dst net 216.58 and dst port 80";
  mtcp_bind_monitor_filter(mctx, msock, &ft);

  mtcp_register_callback(mctx, msock, MOS_ON_CONN_START, MOS_HK_SND, on_flow_start);

  http_event = mtcp_define_event(MOS_ON_CONN_NEW_DATA, chk_http_request);
  mtcp_register_callback(mctx, msock, http_event, MOS_HK_RCV, on_http_request);
}
```

Sets up a traffic filter in Berkeley packet filter (BPF) syntax

Defines a user-defined event that detects an HTTP request

Uses a built-in event that monitors each TCP connection start event

UDE Filter Function

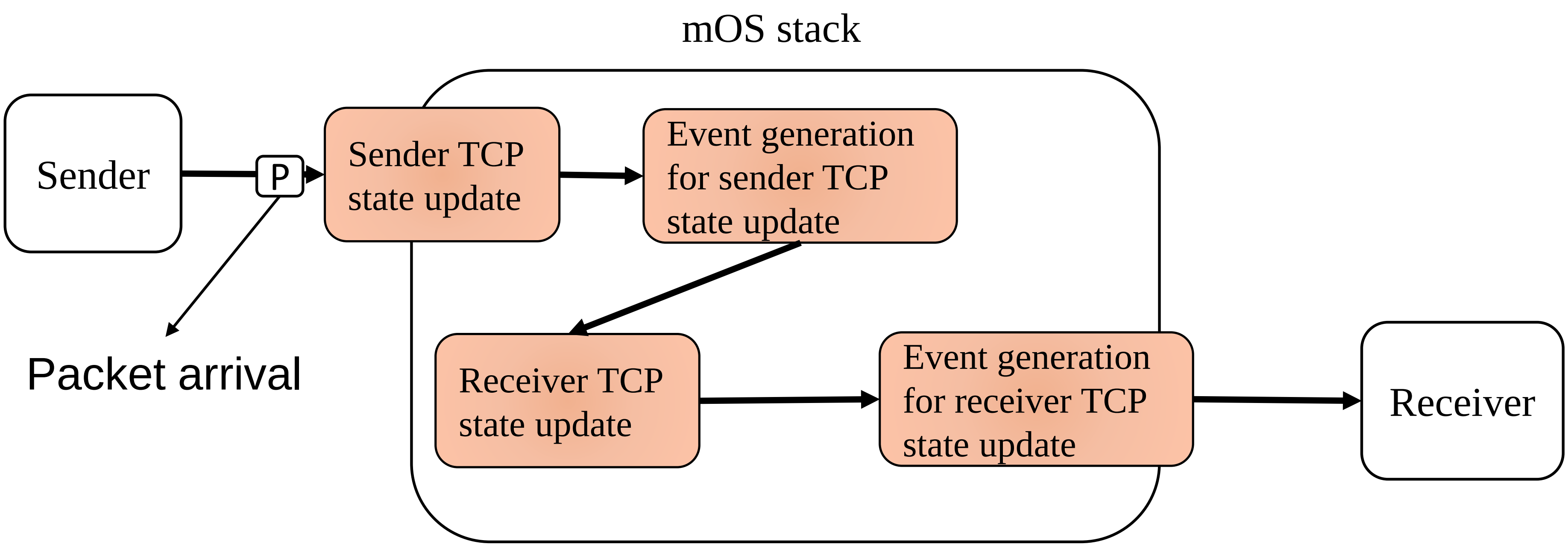
```
static bool chk_http_request(mctx_t m, int sock, int side, event_t event)
{
   struct httpbuf *p;
   u_char* temp; int r;

   if (side != MOS_SIDE_SVR) // monitor only server-side buffer
      return false;
   if ((p = mtcp_get_uctx(m, sock)) == NULL) {
       p = calloc(1, sizeof(struct httpbuf));
       mtcp_set_uctx(m, sock, p);
   }
   r = mtcp_peek(m, sock, side, p->buf + p->len, REQMAX - p->len - 1);
   p->len += r;  p->buf[p->len] = 0;
   if ((temp = strstr(p->buf, "\n\n")) ||(temp = strstr(p->buf, "\r\n\r\n"))) {
        p->reqlen = temp - p->buf;
        return true;
   }
   return false;
}
```

Called whenever the base event is triggered

If it returns TURE, UDE callback function is called

Event Generation Process



Carefully reflects what a middlbox sees and operates on

Based on the estimation of sender/receiver's TCP states
- Packet arrival: sender's state has already been updated
- Infer the receiver stack update with a new packet

Scalable Event Management

Each flow subscribes to a set of events

Each flow can change its own set of events over time

- Some flow ***adds*** a new event or delete an event
- Some flow ***changes*** the event handler for an event

Scalability problem

- How to manage event sets for 100+K concurrent flows?

Observation: the same event sets are shared by multiple flows

How to represent the event set for a flow?

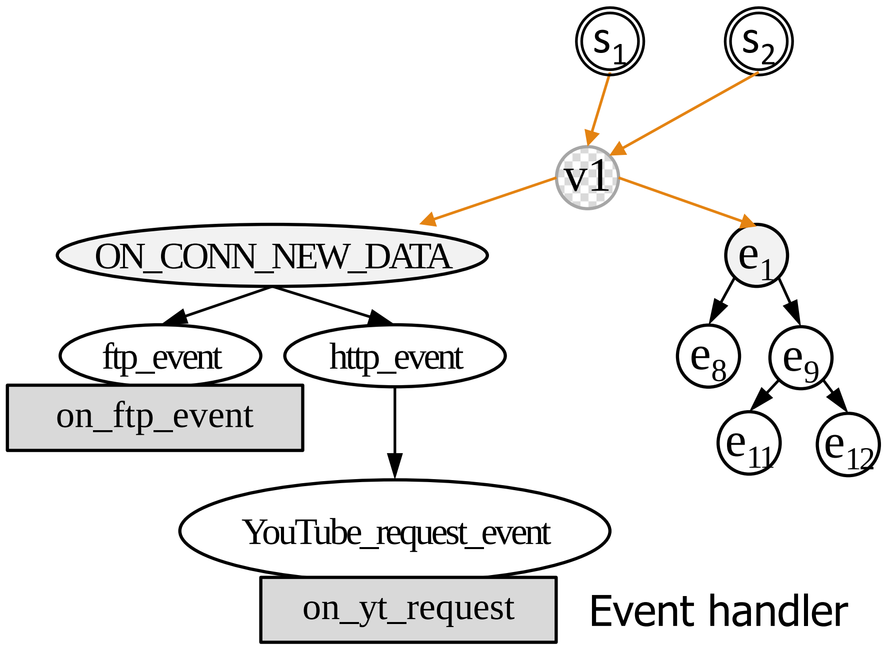
How to efficiently find the same event set?

- When a flow updates its set of events?

Event Dependency Tree

Represents how a UDE is defined

Start from a built-in event as root

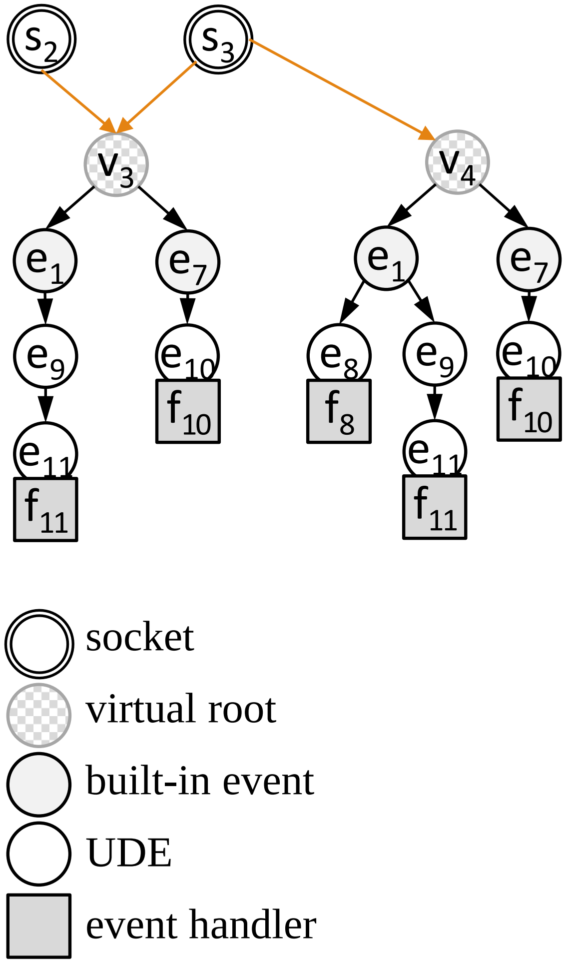


Event handler

New flow

Points to a virtual root that has a set of dependency trees

Update on Event Dependency Tree



s3 adds a new event <e8, f8> to v3

v4 is created with a new event and s3 points to it

s2 adds the same event <e8, f8> to v3

v4 already exists, but how does s2 find v4?

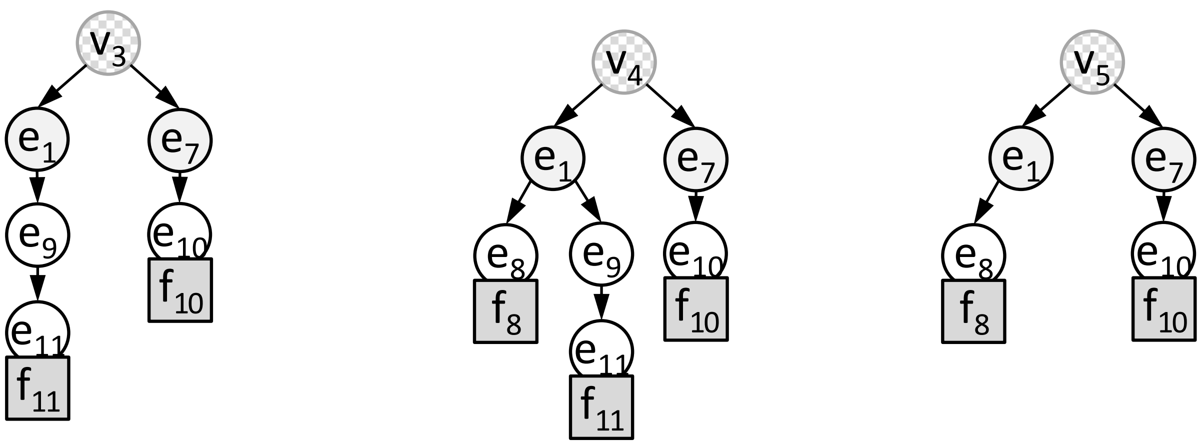
Efficient Search for an Event Dependency Tree

Each event dependency tree has an ID

- id (virtual root) = XOR sum of hash (event + event handler)
- id (v3) = hash (e11 + f11) $\oplus$ hash (e10 + f10)

New tree id after adding or deleting <e, f> from t

- id (t') = id (t) $\oplus$ hash (e + f)
- Add <e8, f8> to v3?
  - id(v4) = id(v3) $\oplus$ hash (e8 + f8)
- Remove <e10, f10> from v4?
  - id (v5) = id(v4) $\oplus$ hash (e11 + f11)



Current mOS stack API

```
Socket creation and traffic filter
int      mtcp_socket(mctx_t mctx, int domain, int type, int protocol);
int      mtcp_close(mctx_t mctx, int sock);
int      mtcp_bind_monitor_filter(mctx_t mctx, int sock, monitor_filter_t ft);
User-defined event management
event_t mtcp_define_event(event_t ev, FILTER filt);
int      mtcp_register_callback(mctx_t mctx, int sock, event_t ev, int hook, CALLBACK cb);
Per-flow user-level context management
void *   mtcp_get_uctx(mctx_t mctx, int sock);
void     mtcp_set_uctx(mctx_t mctx, int sock, void *uctx);
Flow data reading
ssize_t mtcp_peek(mctx_t mctx, int sock, int side, char *buf, size_t len);
ssize_t mtcp_ppeek(mctx_t mctx, int sock, int side, char *buf, size_t count, off_t
seq_off);
```

Current mOS stack API

```
Packet information retrieval and modification

int     mtcp_getlastpkt(mctx_t mctx, int sock, int side, struct pkt_info *pinfo);

int     mtcp_setlastpkt(mctx_t mctx, int sock, int side, off_t offset, byte *data, uint16_t
datalen, int option);

Flow information retrieval and flow attribute modification

int     mtcp_getsockopt(mctx_t mctx, int sock, int l, int name, void *val, socklen_t *len);

int     mtcp_setsockopt(mctx_t mctx, int sock, int l, int name, void *val, socklen_t len);

Retrieve end-node IP addresses

int     mtcp_getpeername(mctx_t mctx, int sock, struct sockaddr *addr, socklen_t *addrlen);

Per-thread context management

mctx_t  mtcp_create_context(int cpu);

int     mtcp_destroy_context(mctx_t mctx);

Initialization

int     mtcp_init(const char *mos_conf_fname);
```

Fine-grained Resource Allocation

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

Fine-control resource consumption

- Disable flow reassembly but keep only metadata
- Enable flow monitoring for one side
- Stop flow monitoring in the middle
- Per-flow manipulation with setsockopt()

```
// disabling receive buffers for both client and server stacks
int zero = 0;
if (!(config_monitor_side & MOS_SIDE_CLI))
    mtcp_setsockopt(mctx, sock, SOL_MONSOCKET, MOS_CLIBUF, &zero, sizeof(zero));
 if (!(config_monitor_side & MOS_SIDE_SVR))
    mtcp_setsockopt(mctx, sock, SOL_MONSOCKET, MOS_SVRBUF, &zero, sizeof(zero));
```

mOS Networking Stack Implementation

Per-thread library TCP stack
- ~26K lines of C code (mTCP: ~11K lines)
- Based on mTCP user level TCP stack [NSDI '14]
- Exploits parallelism on multicore systems

User-defined event implementation
- Designed to scale to arbitrary number of events
- Identical events are automatically shared by multiple flows

Applications ported to mOS: ~9x code line reduction

Application	Modified	SLOC	Output
Snort	884	79,889	HTTP/TCP inspection
nDPI	765	25,483	Stateful session management
PRADS	615	10,848	Stateful session management
Abacus	-	4,091→486	Detect out-of-order packet retransmission

Evaluation: Experiment Setup

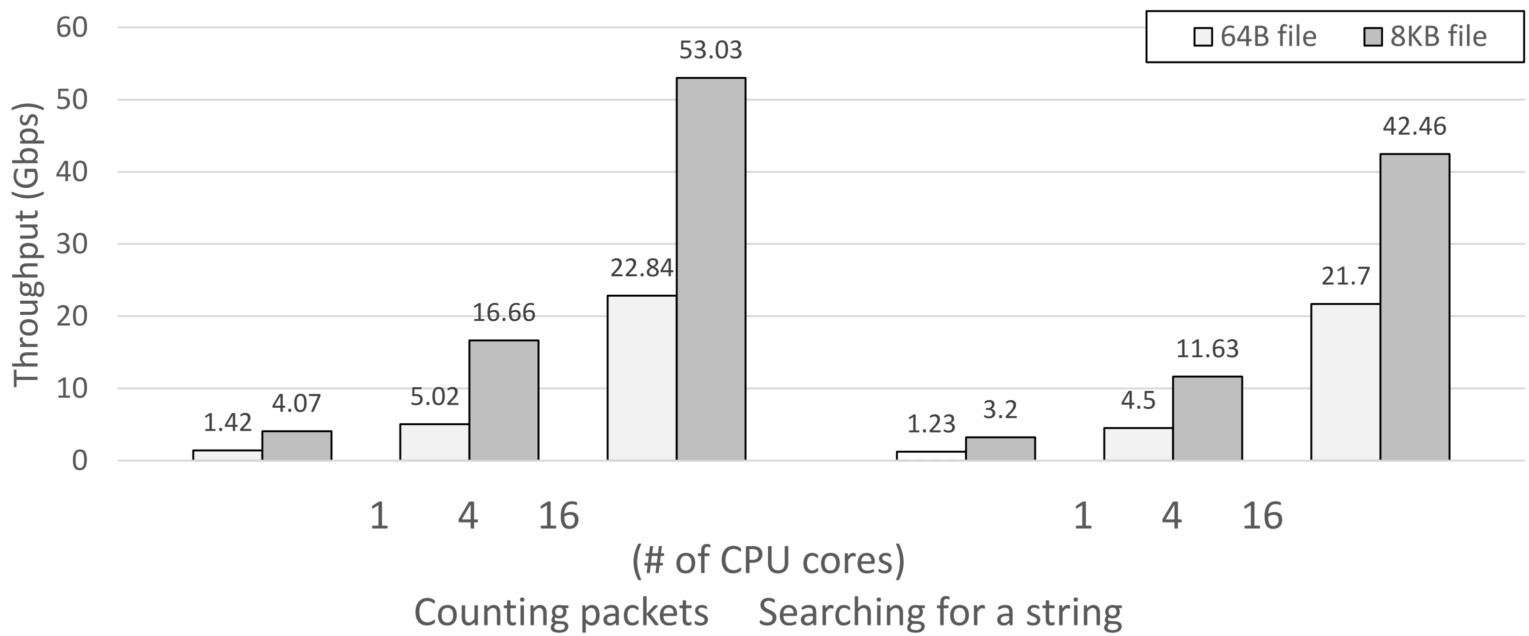
Operating as **in-line** mode: clients ⇔ mOS applications ⇔ servers

mOS applications with mOS stream sockets
- 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

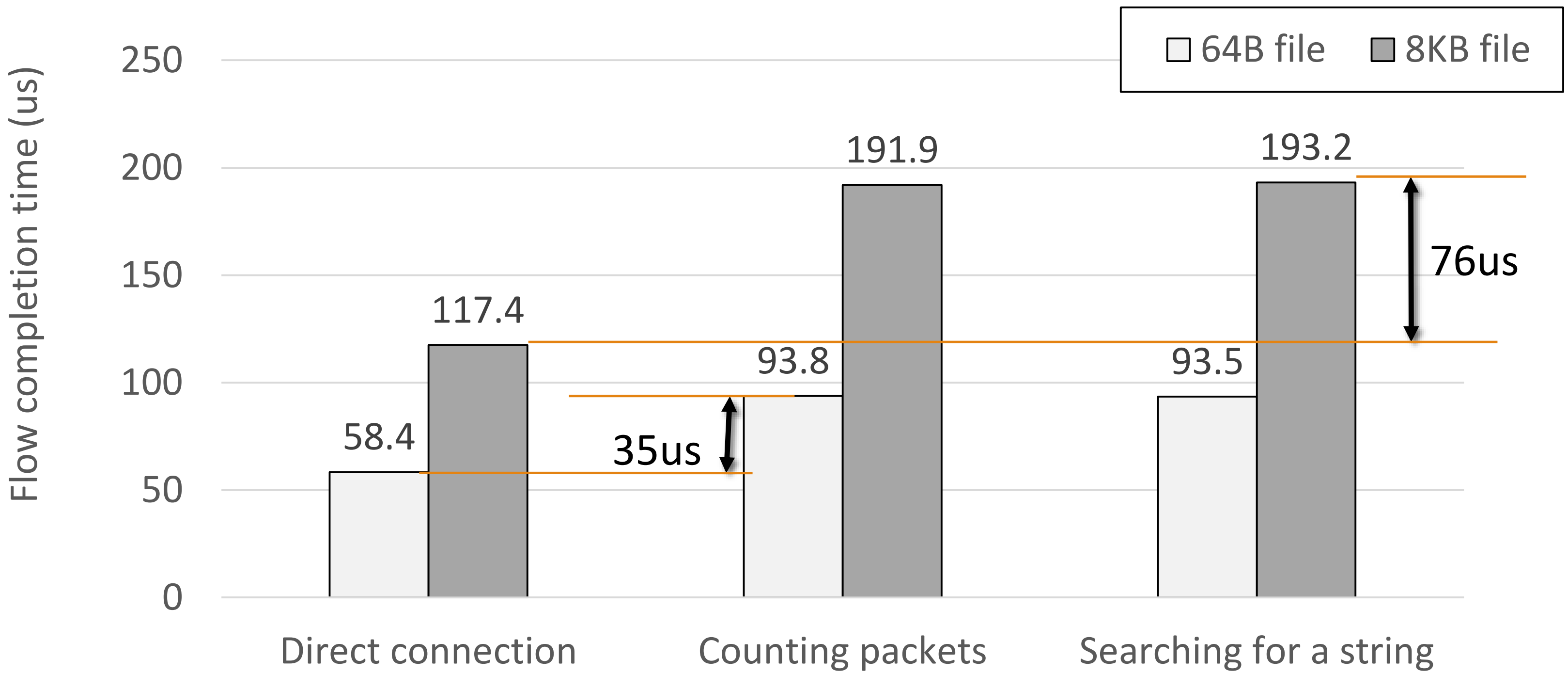
Six pairs of clients and servers: 60 Gbps max
- Intel E3-1220 v3 (4 cores, 3.1 GHz)
- 8 MB L3 cache size
- 16 GB RAM
- 1 x 10 Gbps NIC per machine

Performance Scalability on Multicores

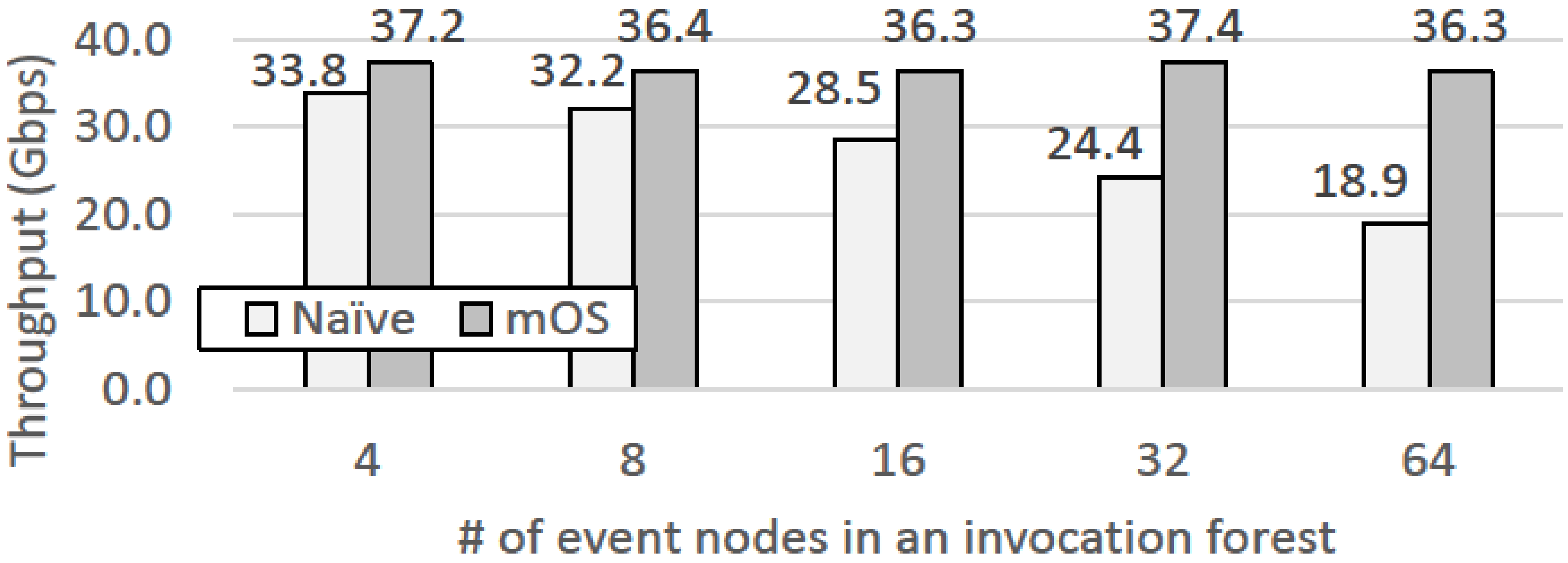
- File download traffic with 192,000 concurrent flows
  - Each flow downloads an X-byte content in one TCP connection
  - A new flow is spawned when a flow terminates
- Two simple applications
  - Counting packets per flow (packet arrival event)
  - Searching for a string in flow reassembled data (full flow reassembly & DPI)



Latency Overhead by mOS Applications



Event Management Performance



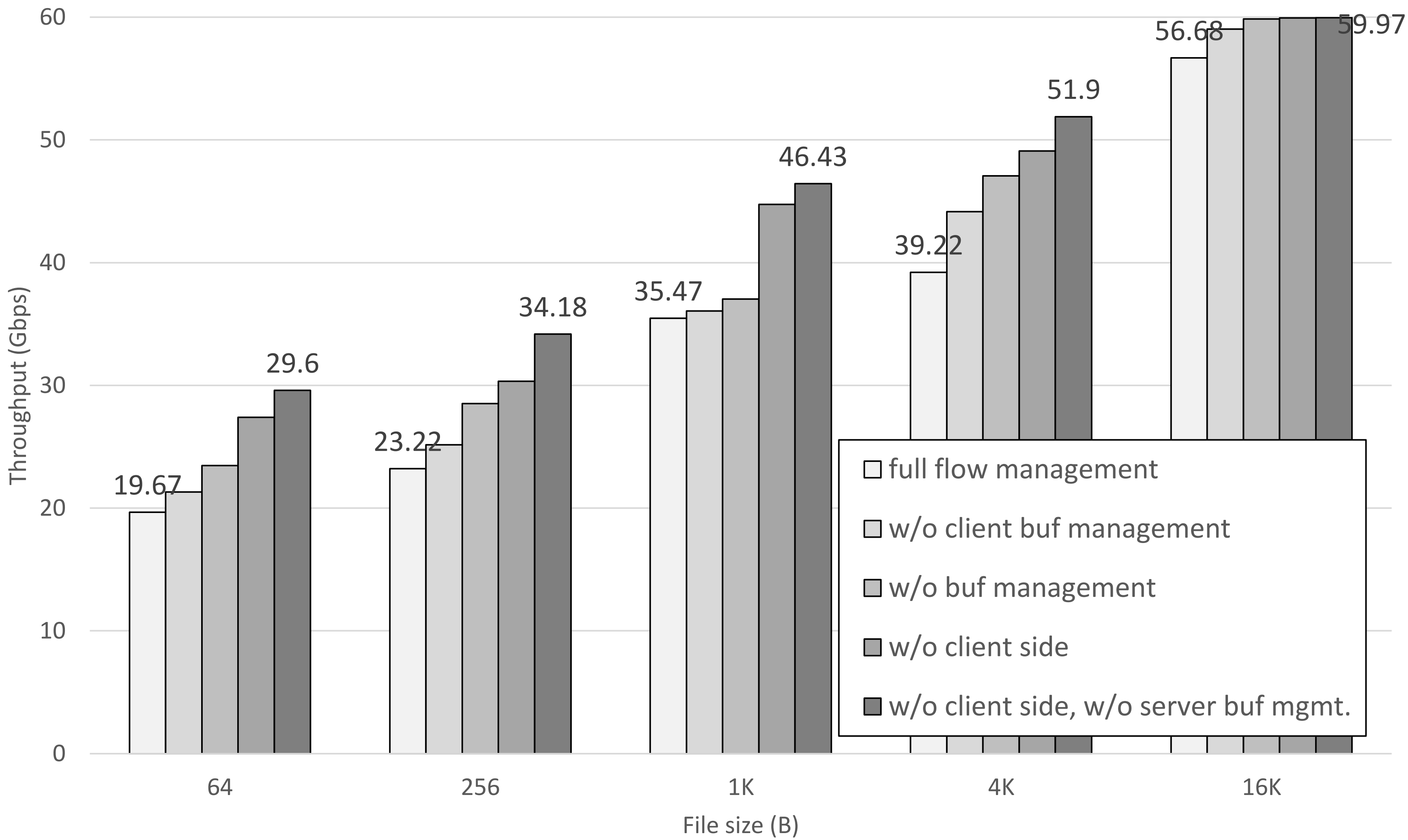
192,000 concurrent flows downloading large files

mOS application searches for a string, dynamically adds a new event

Increases the number of events per flow (4 to 64)

mOS improves the performance by 3.5 to 17.3 Gbps

Performance Under Selective Resource Consumption



Real Application 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
- Mostly due to multi-core aware packet processing (DPDK)
- mOS brings code modularity and correct flow management

Conclusion

Current middlebox development suffers from

- Lack of modularity
- Lack of readability
- Lack of maintainability

Solution: reusable networking stack for middleboxes

mOS stack: abstraction for flow management

- Programming abstraction with socket-based API
- Event-driven middlebox processing
- Efficient resource usage with dynamic resource composition

mOS Stack Is Open-Sourced

Public release of mOS stack/API at github

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

mOS online manual

- http://mos.kaist.edu/guide/

Thank you!

mOS project page

http://mos.kaist.edu/

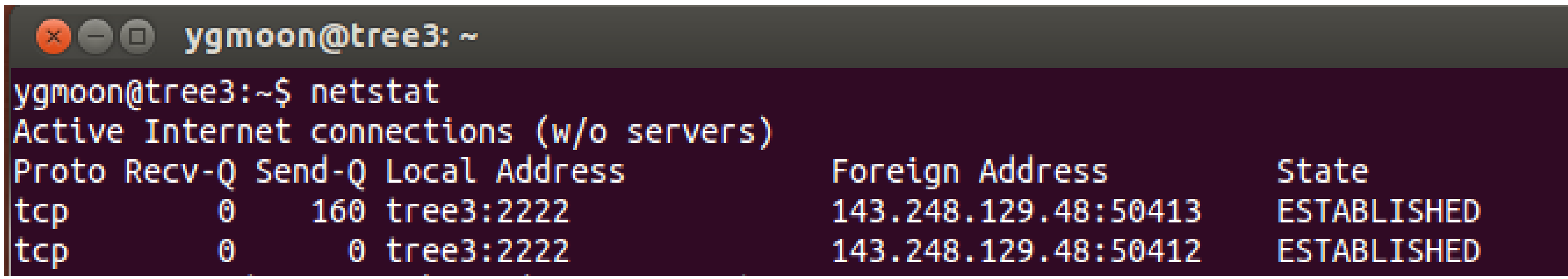
Backup Slides: Sample mOS applications

midstat:
netstats as middlebox

netstat

A command-line tool that displays ***net****work* ***stat****istics*

- Active TCP/UDP connections statistics (both outgoing & incoming)



Prints out the statistics of end-host kernel networking stack (Available on Linux, BSD, Solaris, and Windows)

- Used for finding problems in network or measuring traffic amount

midstat

A monitoring tool that tracks flow statistics of each side of ongoing connections

- Shows IP addresses, port numbers, and TCP states

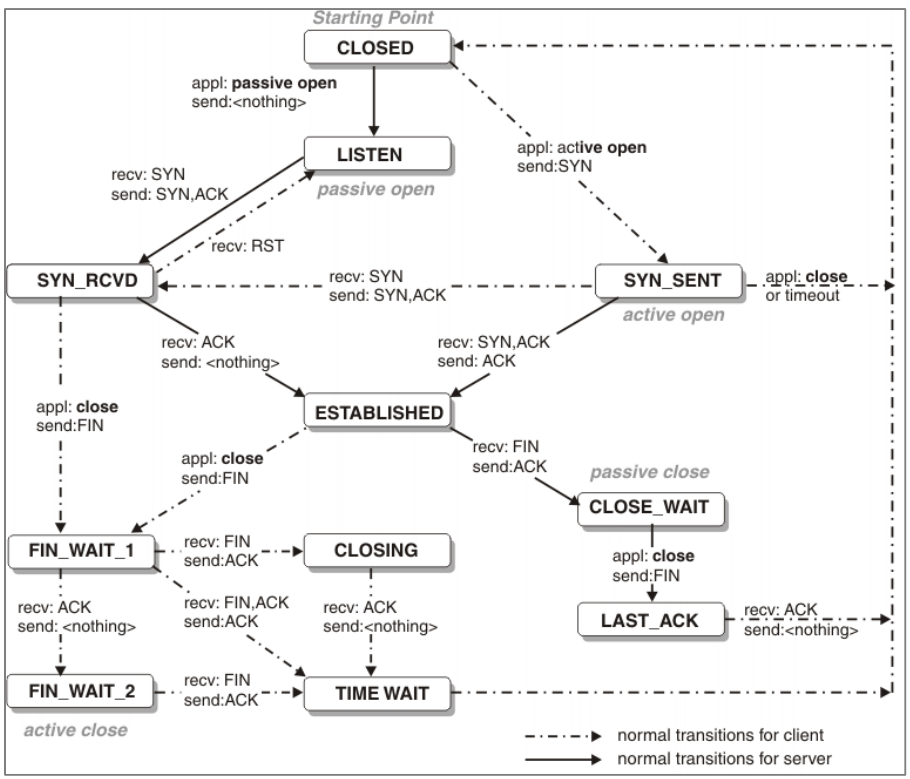
Proto	CPU	Client Address	Client State	Server Address	Server State
tcp	0	10.0.0.12:39476	LAST_ACK	10.0.0.10:80	CLOSING
tcp	0	10.0.0.12:42148	CLOSE_WAIT	10.0.0.10:80	FIN_WAIT_1
tcp	0	10.0.0.12:3420	LAST_ACK	10.0.0.10:80	FIN_WAIT_1
tcp	0	10.0.0.12:17591	CLOSE_WAIT	10.0.0.10:80	FIN_WAIT_1
tcp	0	10.0.0.12:22541	CLOSE_WAIT	10.0.0.10:80	FIN_WAIT_1
tcp	0	10.0.0.12:22784	CLOSE_WAIT	10.0.0.10:80	FIN_WAIT_1
tcp	0	10.0.0.12:27281	CLOSE_WAIT	10.0.0.10:80	FIN_WAIT_1
tcp	0	10.0.0.12:33422	CLOSE_WAIT	10.0.0.10:80	FIN_WAIT_1
tcp	0	10.0.0.12:1032	CLOSE_WAIT	10.0.0.10:80	FIN_WAIT_1
tcp	0	10.0.0.12:37428	LAST_ACK	10.0.0.10:80	CLOSING

--- and 4187 more flows ---

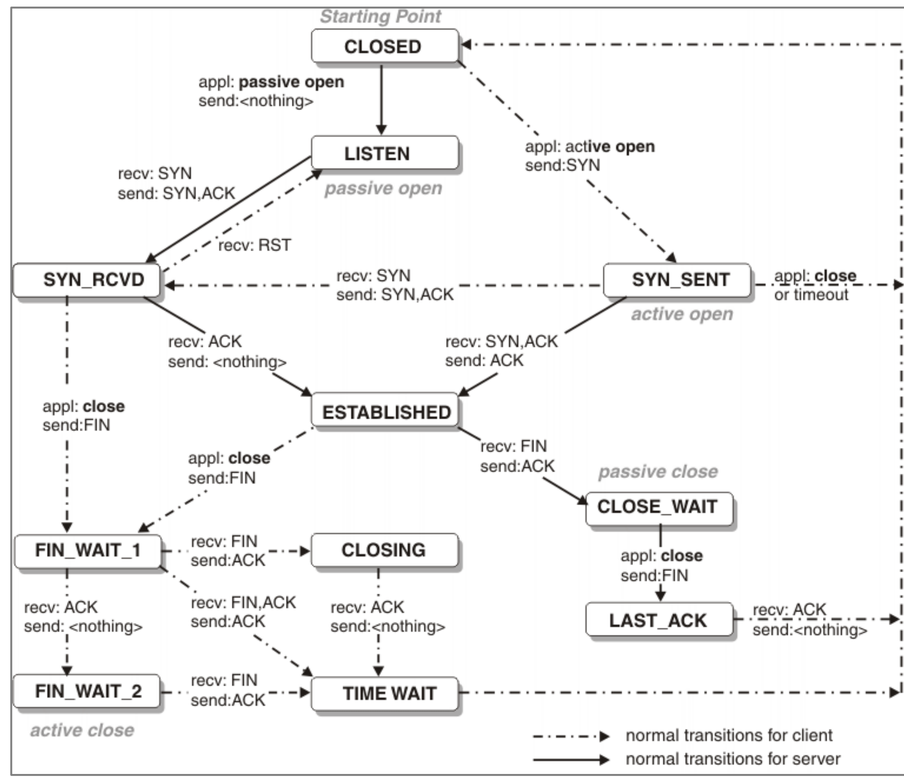
Implementing midstat without mOS

How to track the TCP states of both server and client?

- Without mOS, the app should maintain complex TCP state machines



Client-side TCP state

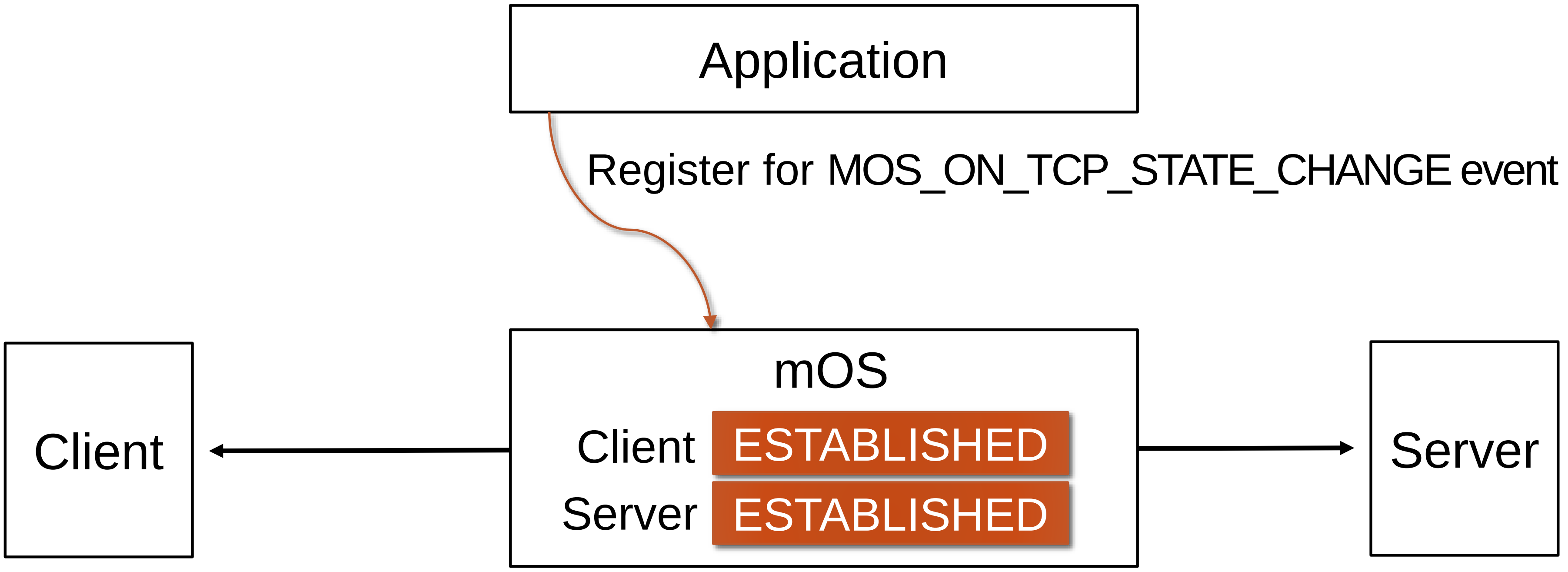


Server-side TCP state

Implementing midstat with mOS

mOS tracks the TCP states of both client- and server-side

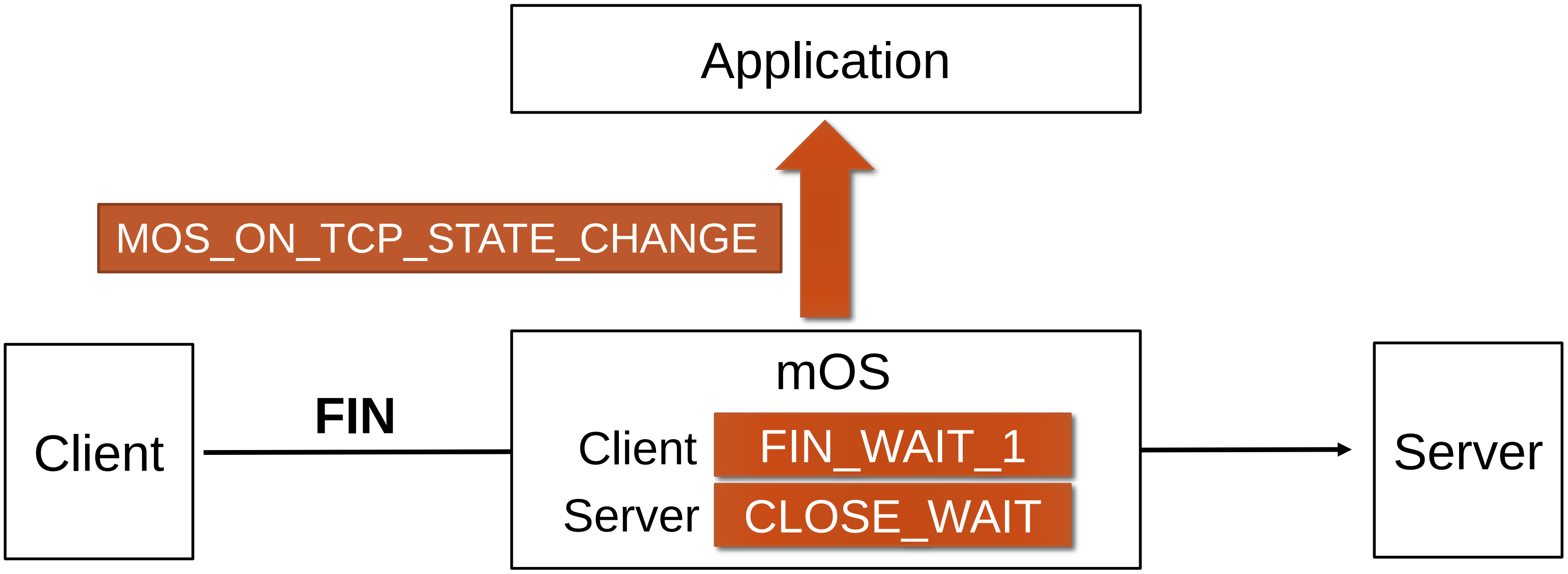
- Notifies the application when there is any TCP state update



Implementing midstat with mOS

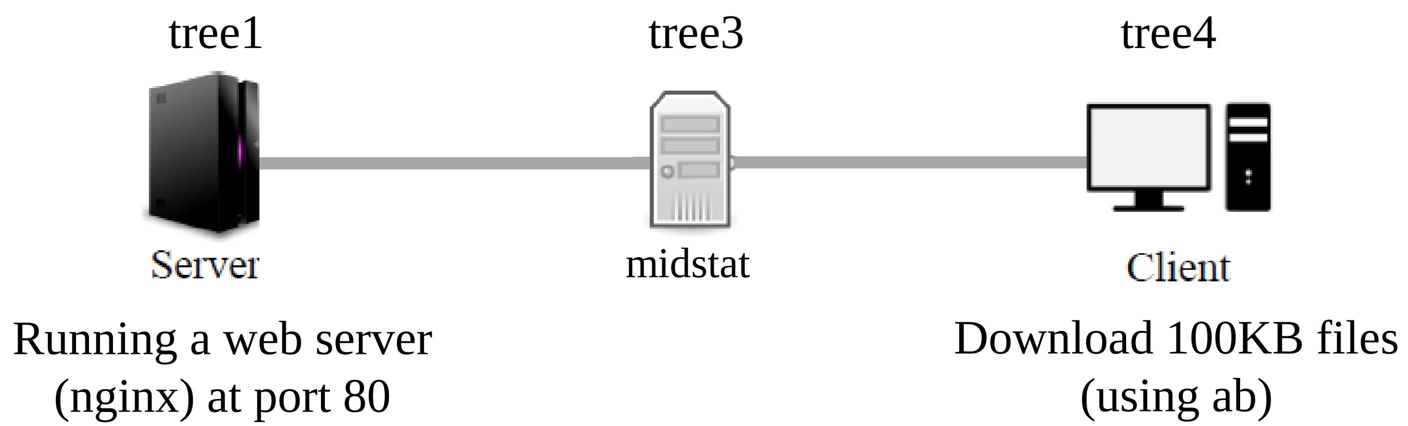
mOS tracks the TCP states of both client- and server-side

- Notifies the application when there is any TCP state update



midstat Demo

Test environment

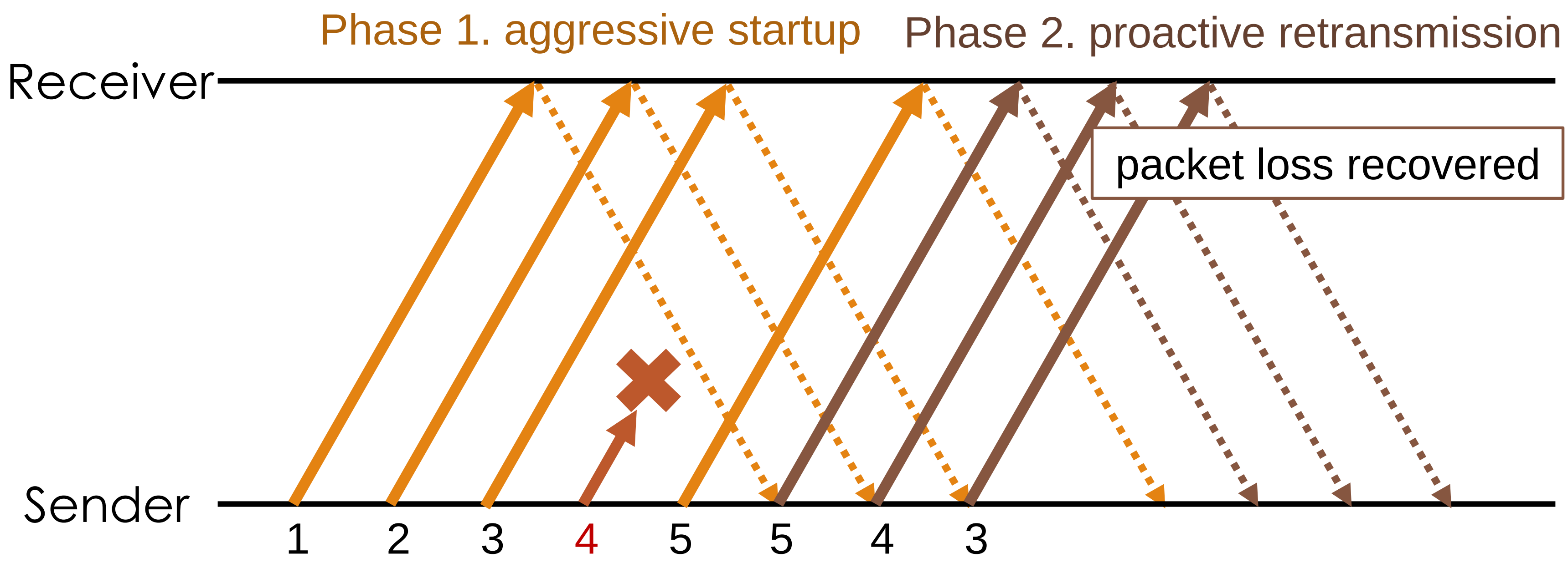


mHalfback

Halfback

A transport-layer scheme designed for optimizing the flow completion time (FCT) *[CoNEXT '15]*

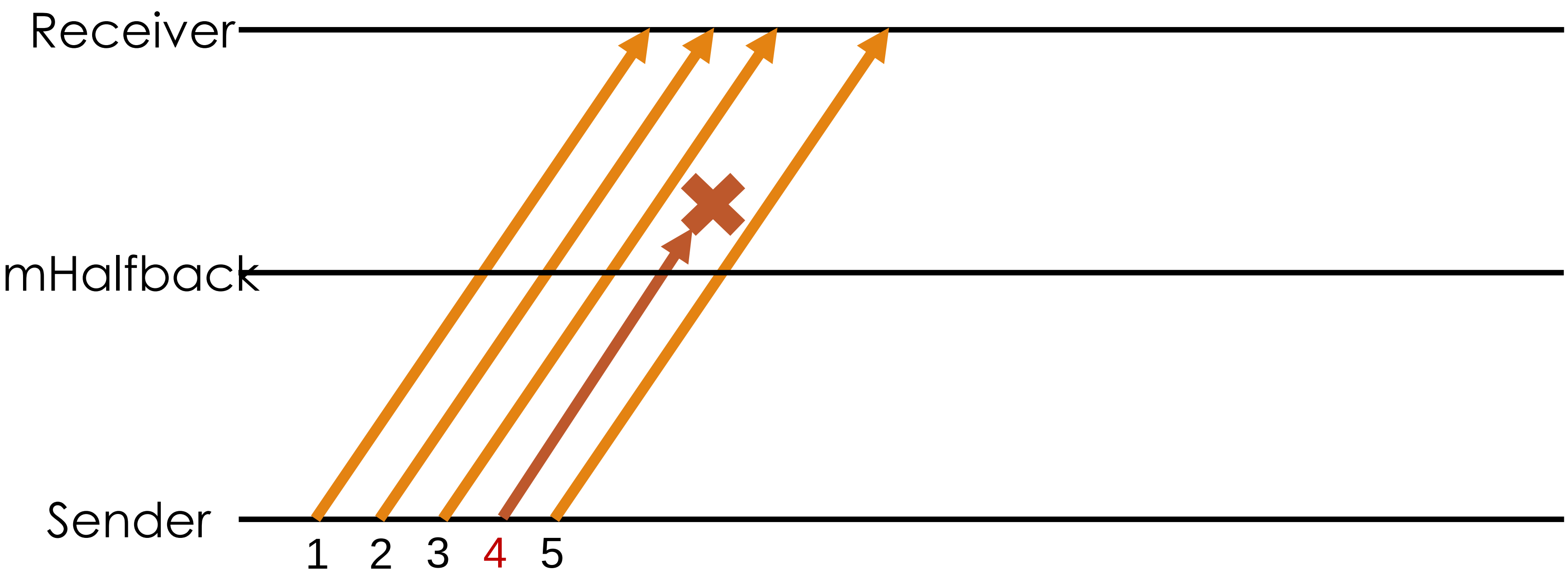
- Skips the TCP slow start phase to pace up transmission rate at start
- Performs proactive retransmission for fast packet loss recovery



mHalfback Proxy

We design mHalfback, a middlebox for fast packet loss recovery
- Transparently reduces the FCT without modifying end-host stacks

The main logic of mHalfback
- 1) For each TCP data packet arrival, hold a copy of the packet



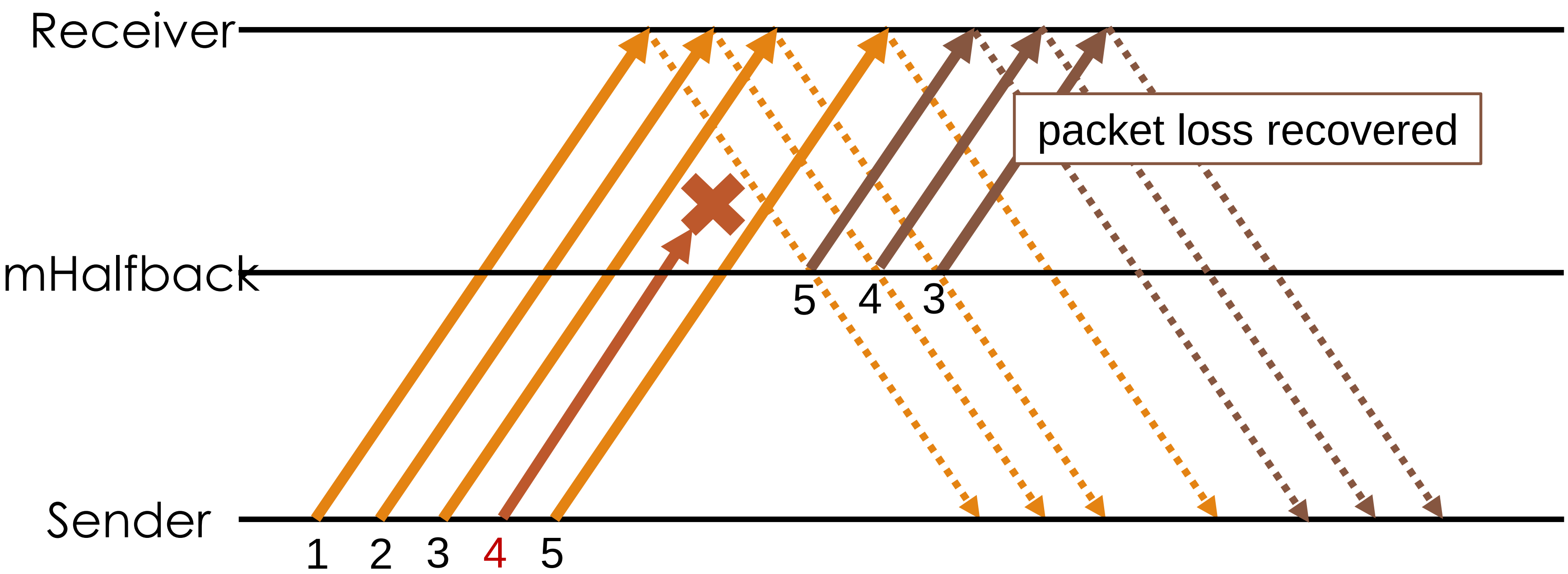
mHalfback Proxy

We design mHalfback, a middlebox for fast packet loss recovery

- Transparently reduces the FCT without modifying end-host stacks

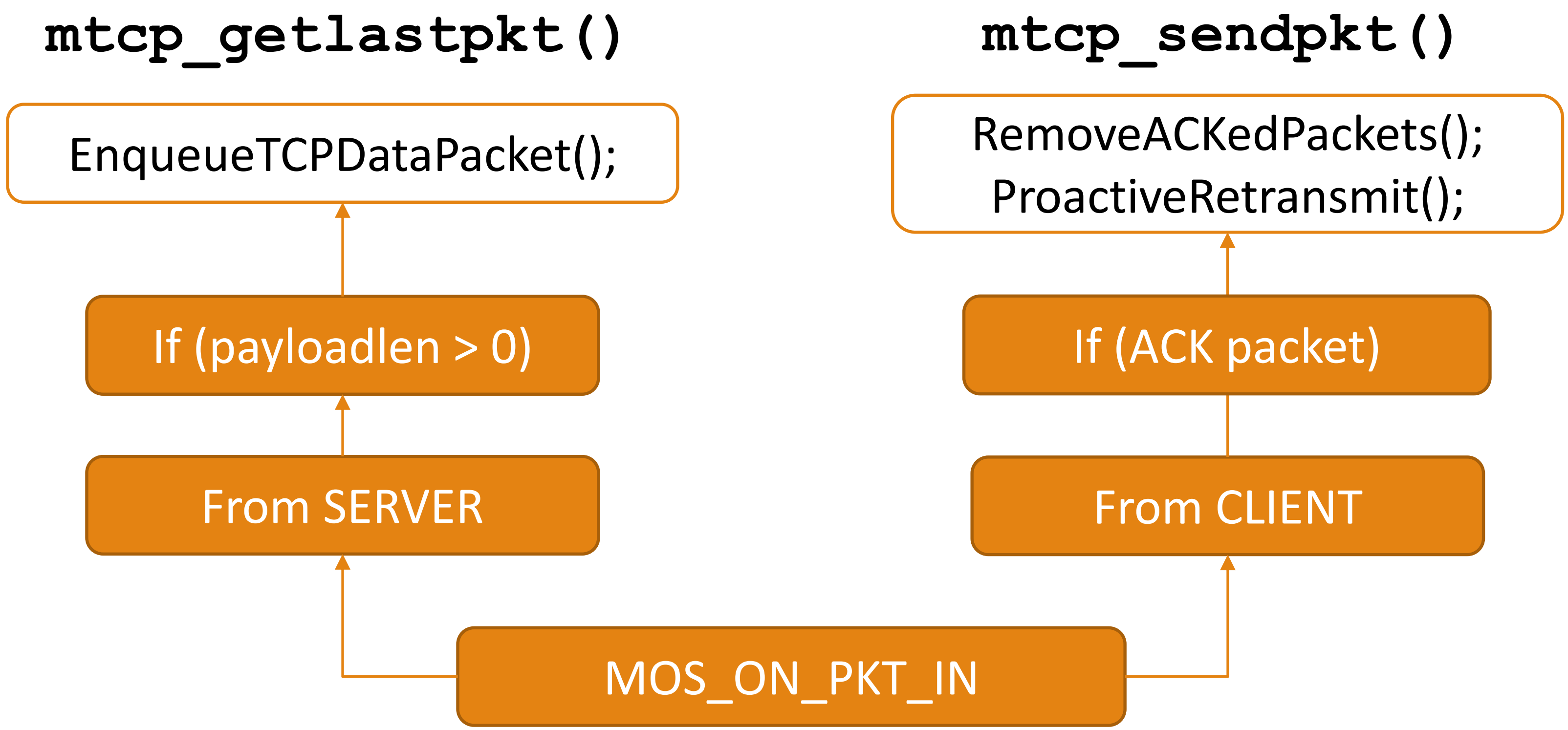
The main logic of mHalfback

- 2) When an ACK packet comes from the receiver, do retransmission



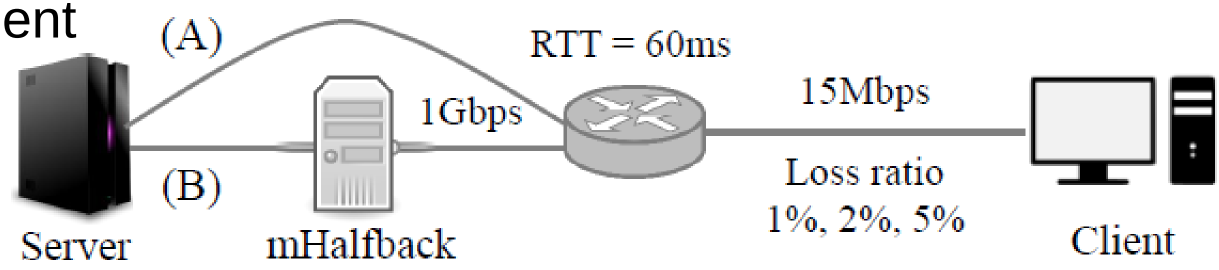
Implementing mHalfback with mOS

How to perform proactive retransmission?

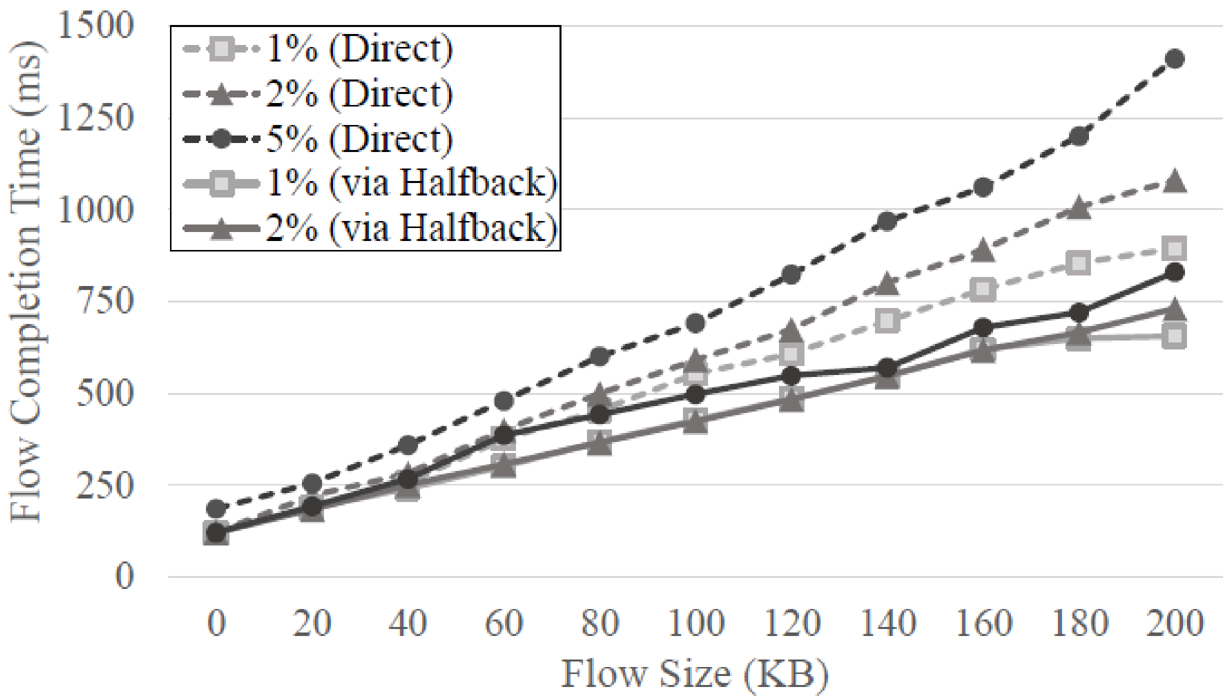


mHalfback Demo

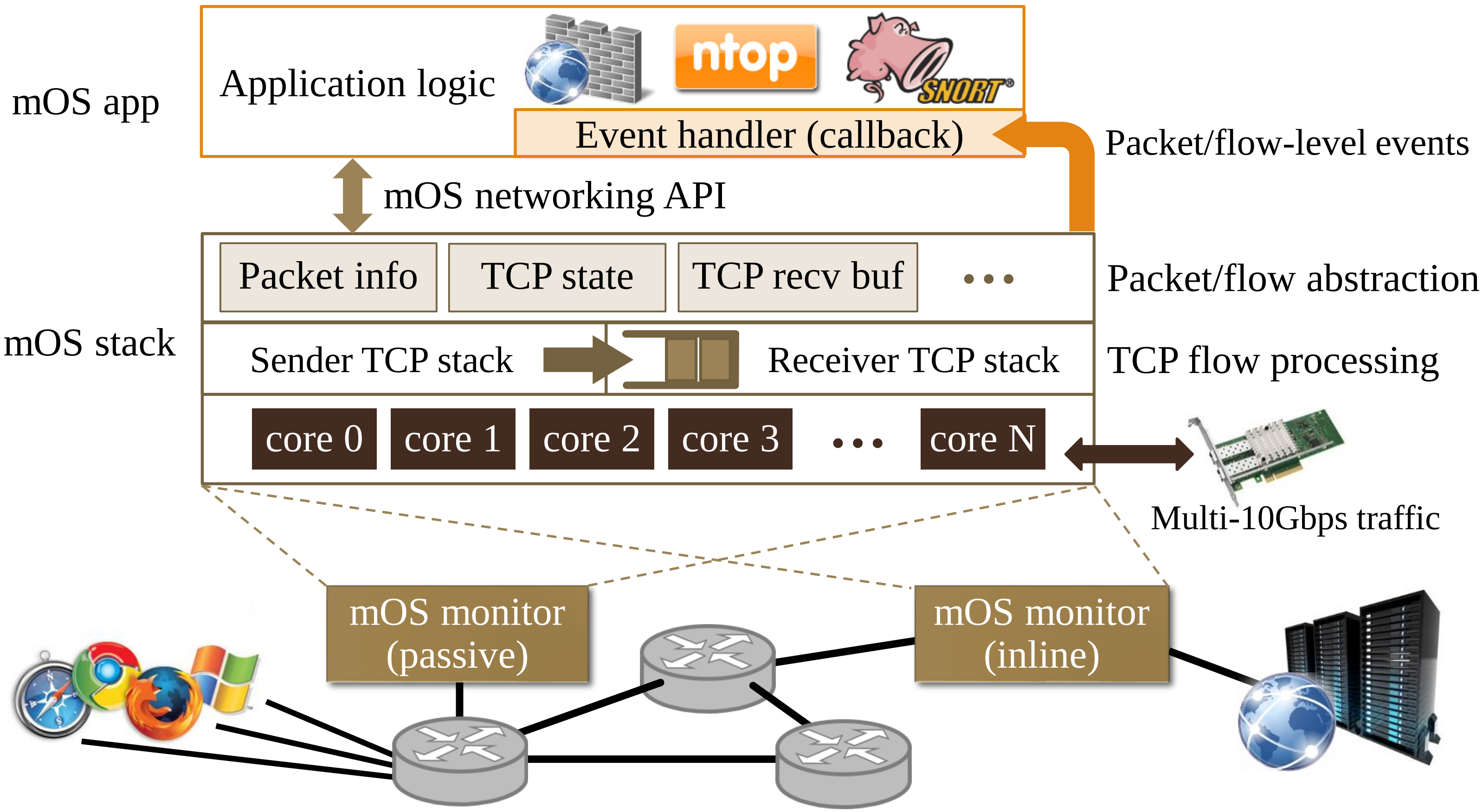
- Test environment



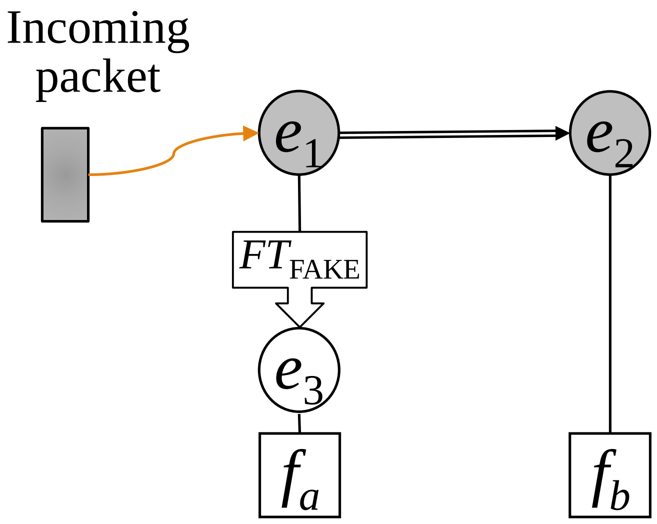
- Results



Operation Scenarios of mOS Applications



Cellular Data Accounting with Events



$e1$: packet retransmission event

FT_{FAKE}: see if *e1* has a packet with malicious content
- Called a filter function (boolean function)

$e3$: fake retransmission event
- Triggered when ***e1* is raised && *FT* returns true**

f_a: action for handling $e3$
- Called an event handler for $e3$

$e2$: new data event (no retransmission)

f_b: action for handling $e2$

Developer: defines a custom event (e3) and provides an action

System: provides regular events (e1, e2) and executes event handlers