A practical course on

# Advanced systems programming in C/Rust

Dimitra Giantsidi

Today's topic!

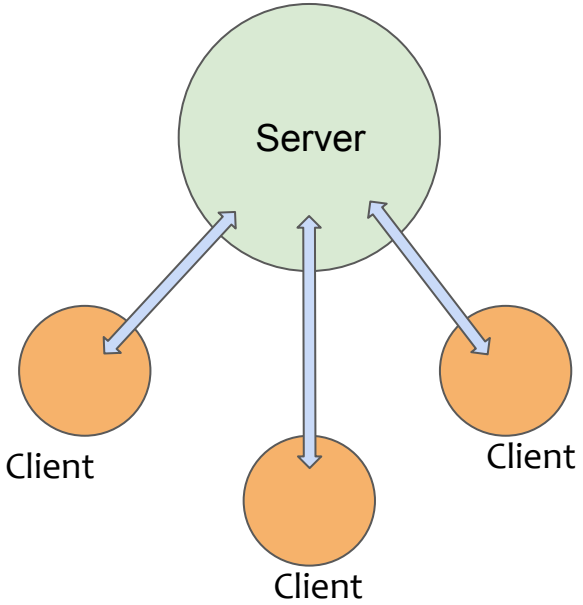
# Network Programming

# Outline

- Basics on networking
  - Fundamentals on communication
  - Networking protocols and the OSI layer
- Sockets
  - Stream/datagram sockets
  - Sockets API
  - Client/server example
- Server design
  - I/O multiplexing (select(), poll(), epoll())
  - Asynchronous I/O
- Tools
  - netstat , tcpdump
- Userspace networking

# Fundamentals - Client/server model

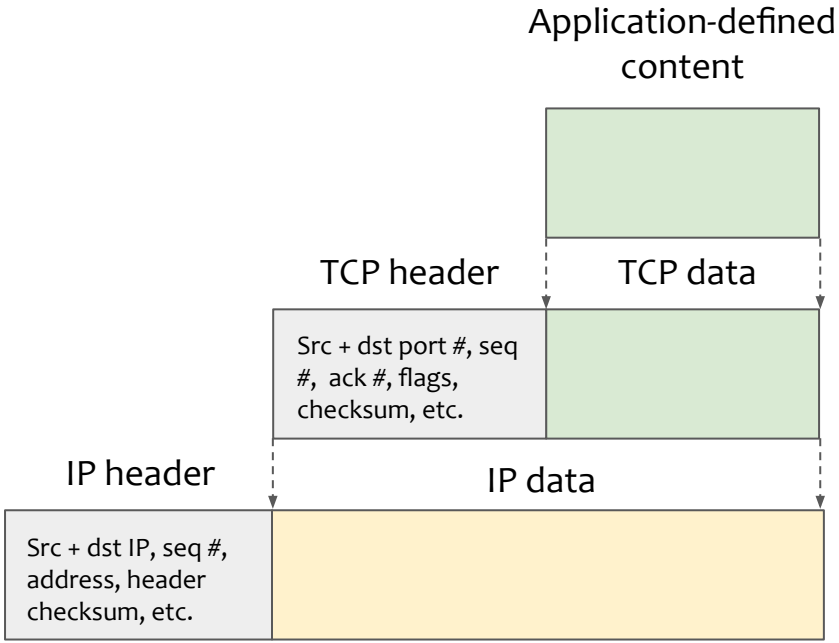
- Server
  - Usually a long running process (*daemon process*)
  - Manages some resources
  - Receives and process requests
- Client(s)
  - Sends one or more requests to the server
  - Wait for the server's reply
- Transport layer
  - Network medium
  - Transfers the data



# Networking Protocols and the OSI Layer

- Networking protocols:
  - Set of rules for data transmission
  - TCP/IP protocol (best-effort protocol)

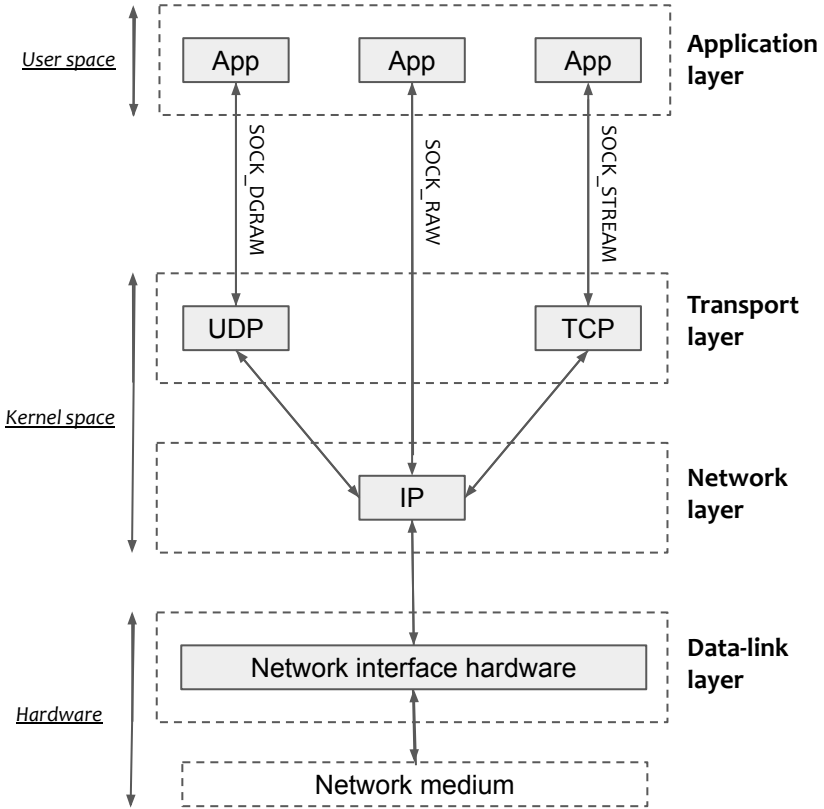
- OSI layer:
  - Encapsulation
  - Application programmers only pass the data down
  - Sockets is the API to access the transport layer functions



# Data-link and Network Layers

- Data-link layer
  - Device drivers and network card
  - Transfers frames
  - Maximum transmission unit (MTU)

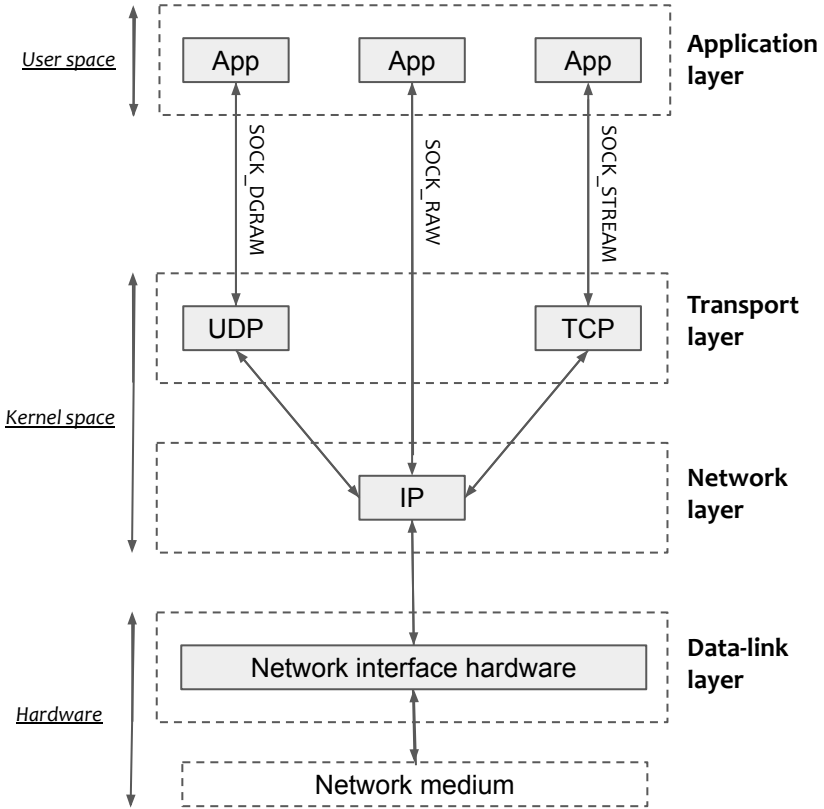
- Network layer (IP)
  - Transfers packets (fragmentation, routing, etc.)
  - Connectionless and unreliable (best-effort)



# Stream/Datagram Sockets

- Stream sockets
  - Implemented on top of TCP
  - reliable, bidirectional, **byte-stream** communication channel

- Datagram sockets
  - Implemented on top of UDP
  - Not reliability; messages might be lost, duplicated or re-ordered
  - The receiver will drop the datagram in case of a queue overflow

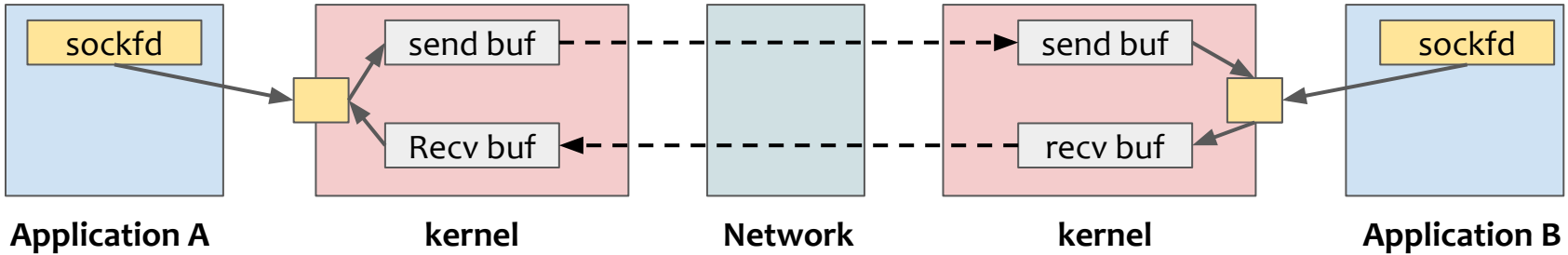


# Sockets API

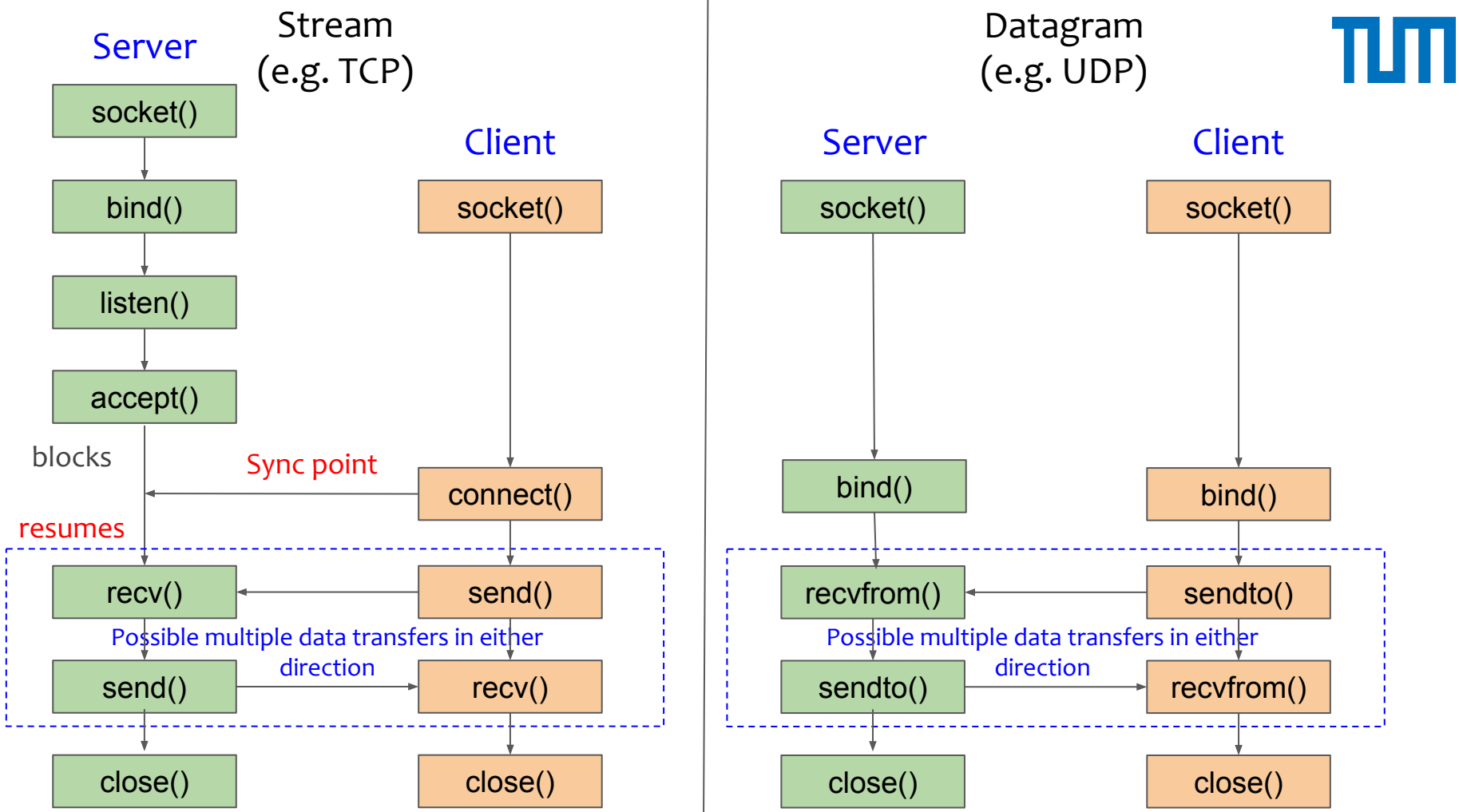
| syscall | Description |
|---|---|
| socket() | Create a new communication endpoint |
| bind() | Attach a local address to a socket |
| listen() | Mark the socket as passive; can accept incoming connections |
| accept() | Accept a received connection request |
| connect() | Actively attempt to establish a connection |
| send(), sendto(), write() | Send **some** data over the connection |
| recv(), recvfrom(), read() | Receive **some** data over the connection |
| close() | Release the connection |

# What is a socket in the end?

- An endpoint of communication (kernel)
- A file descriptor that lets us read/write data to network (application)




- Client/server communication is nothing more that reading and writing to socket descriptors
- Main distinction between regular file I/O and socket I/O is how application “opens” socket descriptors


Stream
(e.g. TCP)
Server
socket()
bind()
listen()
accept()
blocks
Client
socket()
Sync point
connect()
resumes
recv()
send()
Possible multiple data transfers in either direction
send()
recv()
close()
close()
Datagram
(e.g. UDP)
Server
socket()
bind()
recvfrom()
Possible multiple data transfers in either direction
sendto()
recvfrom()
close()
Client
socket()
bind()
sendto()
close()

# Non-blocking Sockets and Asynchronous I/O

- Blocking I/O is convenient but no sufficient;
  - Multiple connections
  - Simultaneous sends/receives
  - Simultaneously doing other-work

- Non-blocking sockets
  - If an operation is going to block, return failure (-1)
  - fcntl(), MSG_DONTWAIT, etc.
  - Require polling

- Asynchronous I/O
  - OS signals the applications when a syscall is completed
  - io_uring, std::async(), std::launch(), boost::asio library, etc.

# Server design (1/2)

- Iterative server
  - Handles **one client** at a time
  - Clients might experience long delays
  - Example: iPerf, etc.

- Multitasking server
  - A per-client process/thread
  - Allows parallelism
  - Process/threads incur overheads (e.g. creation, scheduling, context switching, etc.)
  - Not scalable
  - Examples: apache httpd server, MySQL, etc.

- Multiplexing I/O
  - Supports more than one I/O channels
  - Each thread/process handles more than one connections
  - Requires non-blocking sockets to be effective
  - select(), poll()
  - Examples: nginx, nodejs, redis, etc.

# Server design (2/2)

- select()*, poll()*
  - Monitors multiple file descriptors to see if I/O is possible on any of them
  - Blocks until one of the descriptors is ready or until timeouts
  - Returns which descriptors are ready for reading/writing
- epoll()*
  - Similar to poll() but faster and scales better
  - Can also be used either as an edge-triggered or a level-triggered interface:
    - edge-triggered mode: a call to epoll_wait() returns only when a new event is enqueued with the epoll object
    - level-triggered mode: epoll_wait() returns as long as the condition holds
    - **Example**: a pipe registered with epoll receives data. A call to epoll_wait() will return. The reader only consumes some data from the buffer.
      - level-triggered mode: further calls to epoll_wait() return immediately
      - edge-triggered mode: epoll_wait() will return only once new data is written

* *for more info check the man pages*

# Network Byte Order and Data Representation

- Network byte order
  - Different machines/OS's have different word orderings; little-endian (lower bytes first), big-endian (higher bytes first)
  - The byte ordering used by **the network is always big-endian**
  - htonl(), htons(), ntohl(), ntohs()

- Data representation
  - Heterogeneous architectures/applications
  - Encode text (marshalling)
  - Serialization protocol (google protobufs, etc.)

- Internet socket addresses
  - IPv4 vs IPv6

# Tools (1/3)

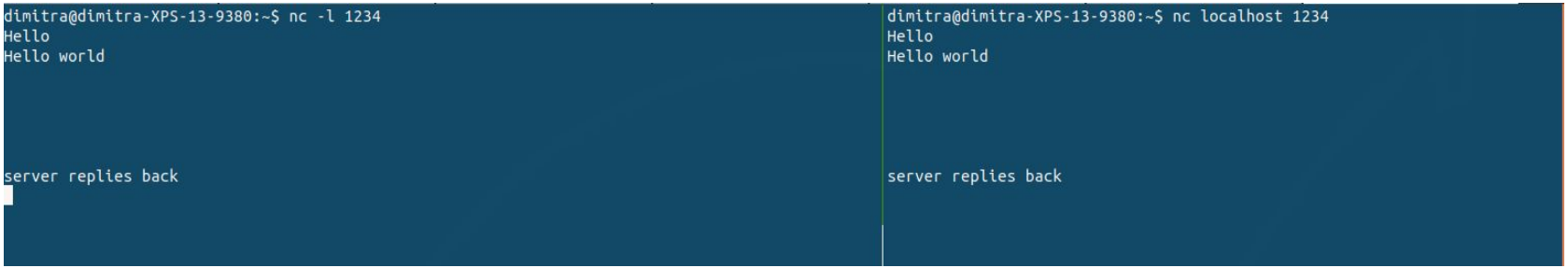
- tcpdump
  - Monitor traffic on a network
  - Example: dump (any) 4 packets and then exit (verbose on)

```
→ ~ sudo tcpdump -a -c 4 -v
tcpdump: listening on tinc.retiolum, link-type RAW (Raw IP), capture size 262144 bytes
11:14:34.340476 IP (tos 0x48, ttl 64, id 62334, offset 0, flags [DF], proto TCP (6), length 176)
    amy.r.ssh > dimitra.r.34254: Flags [P.], cksum 0x539c (correct), seq 3750102806:3750102930, ack 2329784408, win 502, options [nop,nop,TS val 2256225484 ecr 277803412], length 124
11:14:34.340582 IP (tos 0x48, ttl 64, id 62335, offset 0, flags [DF], proto TCP (6), length 184)
    amy.r.ssh > dimitra.r.34254: Flags [P.], cksum 0xece4 (correct), seq 124:256, ack 1, win 502, options [nop,nop,TS val 2256225484 ecr 277803412], length 132
11:14:34.340849 IP (tos 0x48, ttl 64, id 62336, offset 0, flags [DF], proto TCP (6), length 536)
    amy.r.ssh > dimitra.r.34254: Flags [P.], cksum 0xde7f (correct), seq 256:740, ack 1, win 502, options [nop,nop,TS val 2256225484 ecr 277803412], length 484
11:14:34.340936 IP (tos 0x48, ttl 64, id 62337, offset 0, flags [DF], proto TCP (6), length 352)
    amy.r.ssh > dimitra.r.34254: Flags [P.], cksum 0x7f85 (correct), seq 740:1040, ack 1, win 502, options [nop,nop,TS val 2256225484 ecr 277803412], length 300
4 packets captured
5 packets received by filter
0 packets dropped by kernel
→ ~
```

  - Filter packets (e.g. filters based on the src or dst IP, hostname, port, protocol, etc.)
  - Option `-A` to monitor packet content, etc.

# Tools (2/3)

- netcat
  - Anything about TCP/UDP and UNIX-domain sockets
  - Example: bidirectional TCP Server/client
    - Server listens to localhost:1234
    - Client establishes a connection to localhost:1234



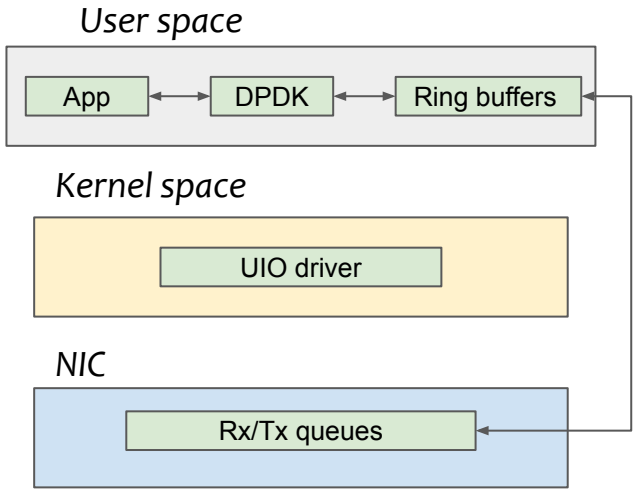

  - Open TCP connections, send UDP packets, listen on TCP/UDP ports, port scanning, etc.
  - Example: nc -v -n <IP> <port range>

# Tools (3/3)

- More complex protocols
  - tshark/Wireshark

# Userspace networking

- Sockets cross the kernel
  - Overheads due to privilege checking, cpu mode switch, data moving, etc.




- Networking libraries that bypass the kernel
  - DPDK, RDMA, etc.

# Assignment

**Tasks:**

- Implement your own client/server application!
- Make use of socket API to design a multiplexing I/O server.
  - Non-blocking sockets, select(), send(), recv(), etc.
- Make use of google::protobufs as a serialization protocol for the messages.

# Thank you for listening!

See you in the Q&A session