Demultiplexing & Error Detection

Lecture 10

http://www.cs.rutgers.edu/~sn624/352-S22

Srinivas Narayana

RUTGERS
UNIVERSITY | NEW BRUNSWICK

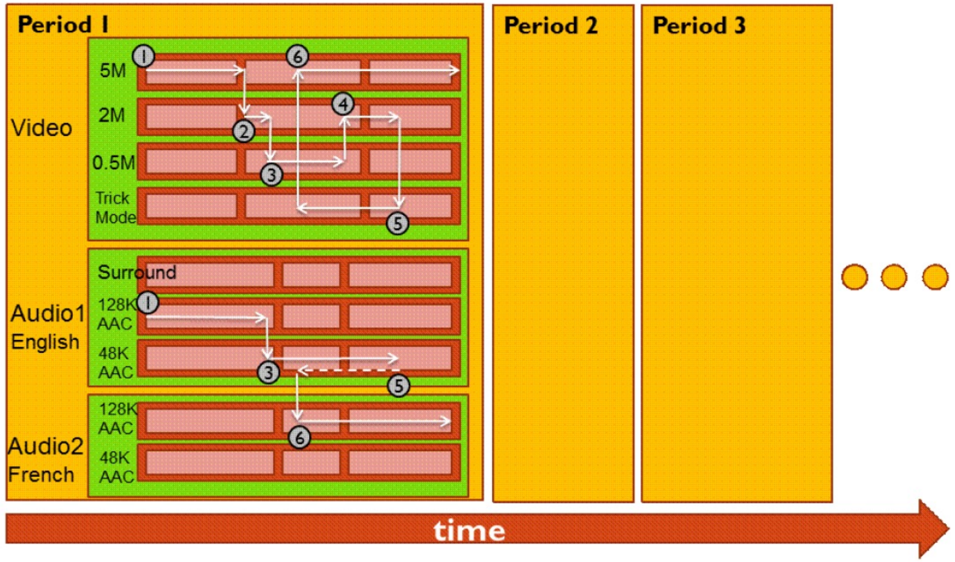
Quick recap of concepts

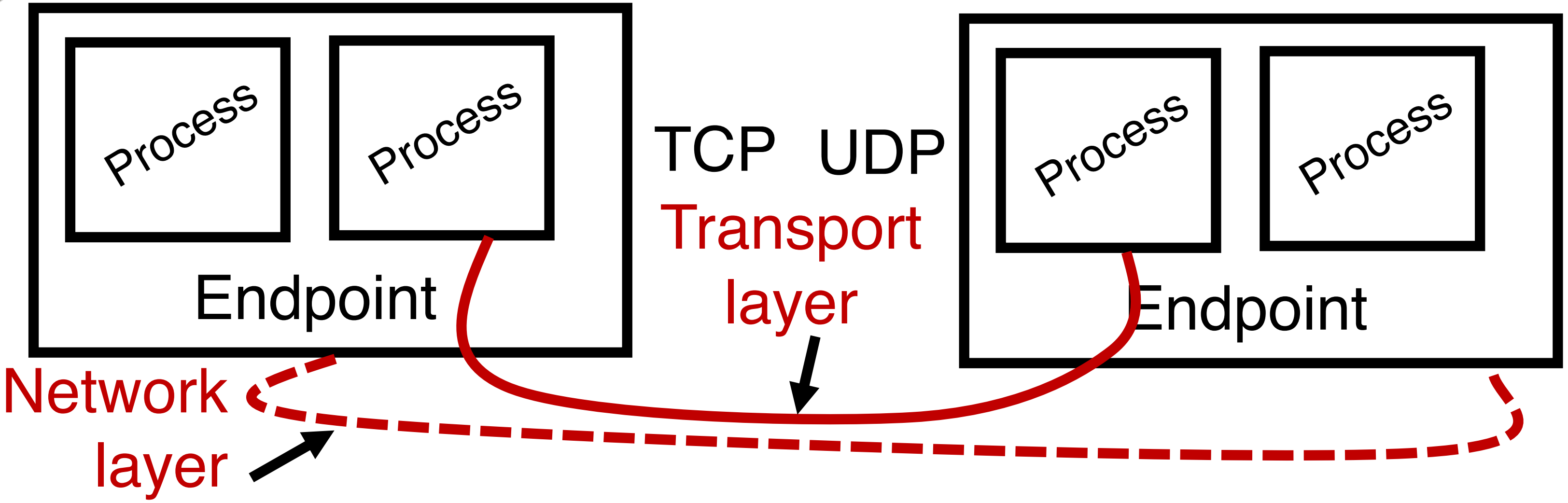
DASH

Video streaming over HTTP

Varying quality, varying sources, over the duration of the video

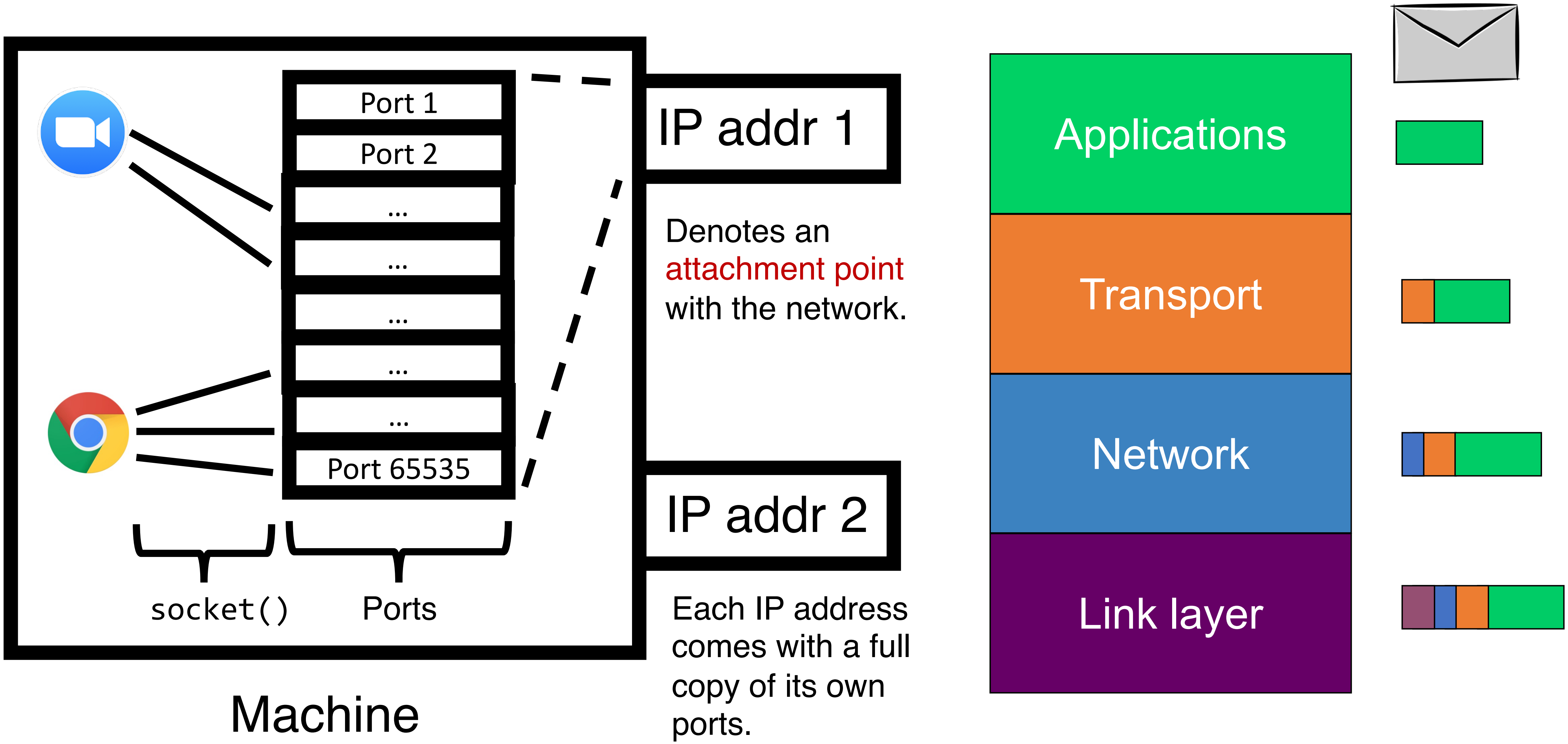
Can use CDNs!





Demultiplexing Packets

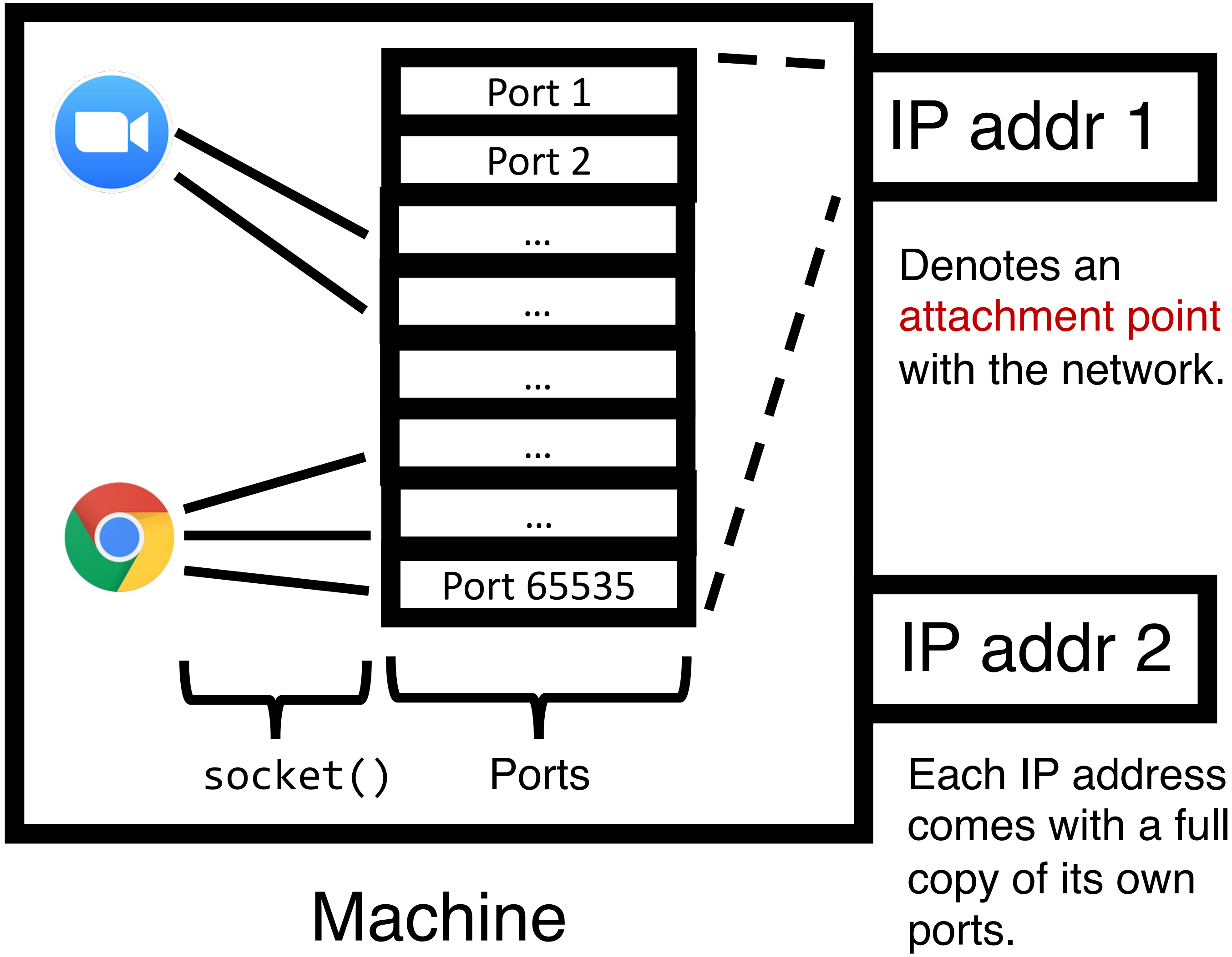
Demultiplexing



Denotes an attachment point with the network.

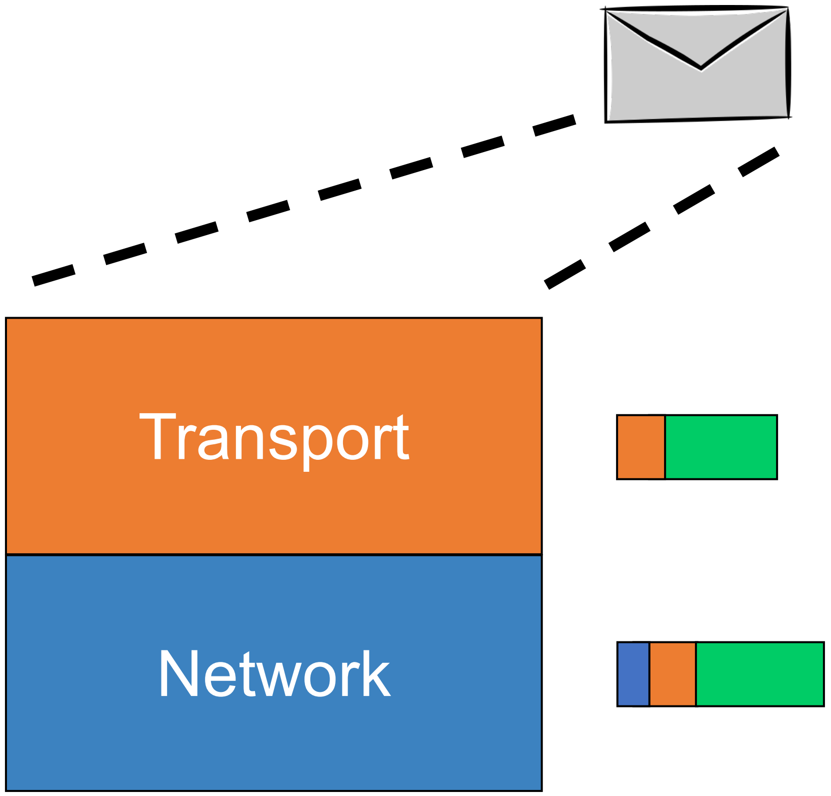
Each IP address comes with a full copy of its own ports.

Demultiplexing

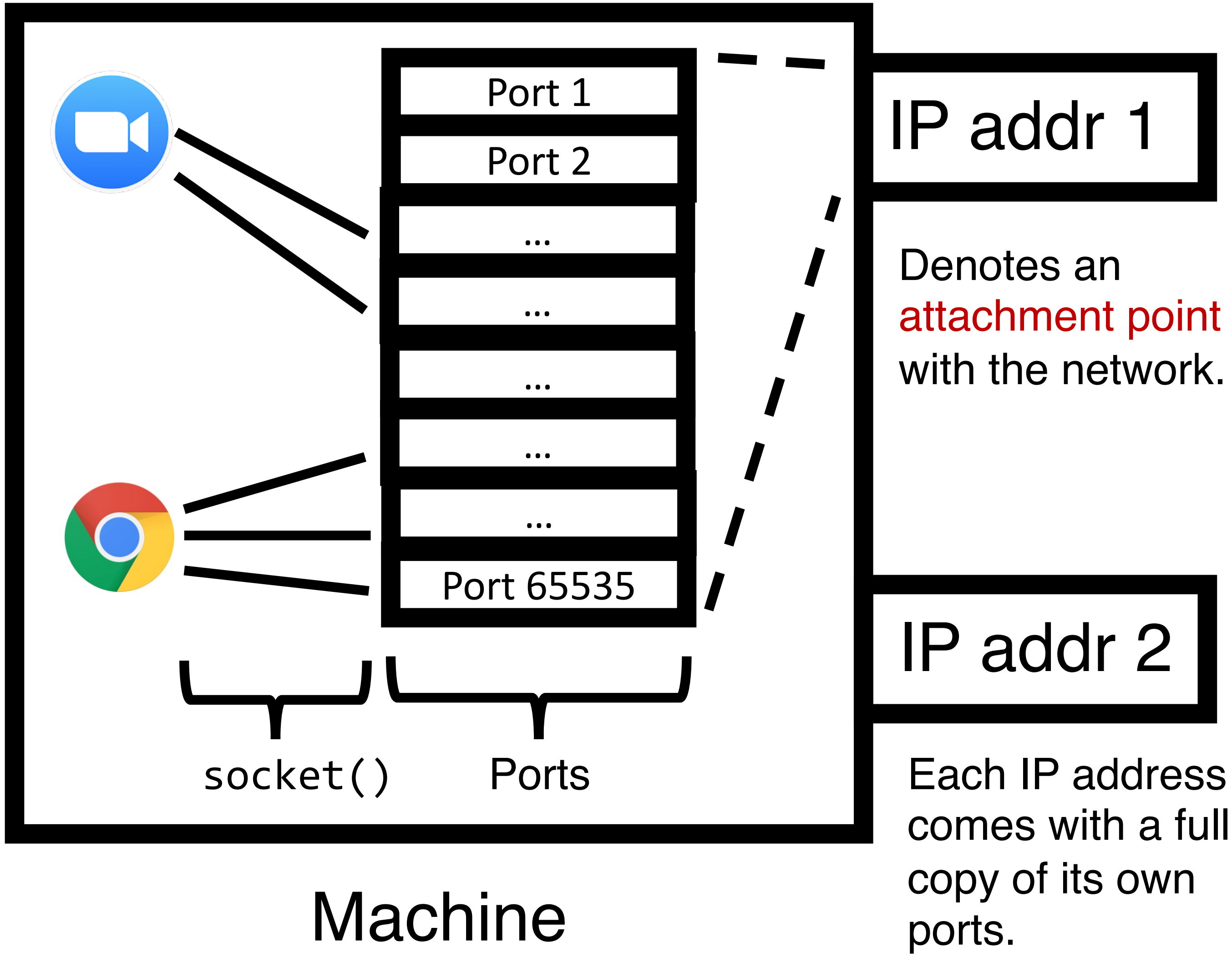


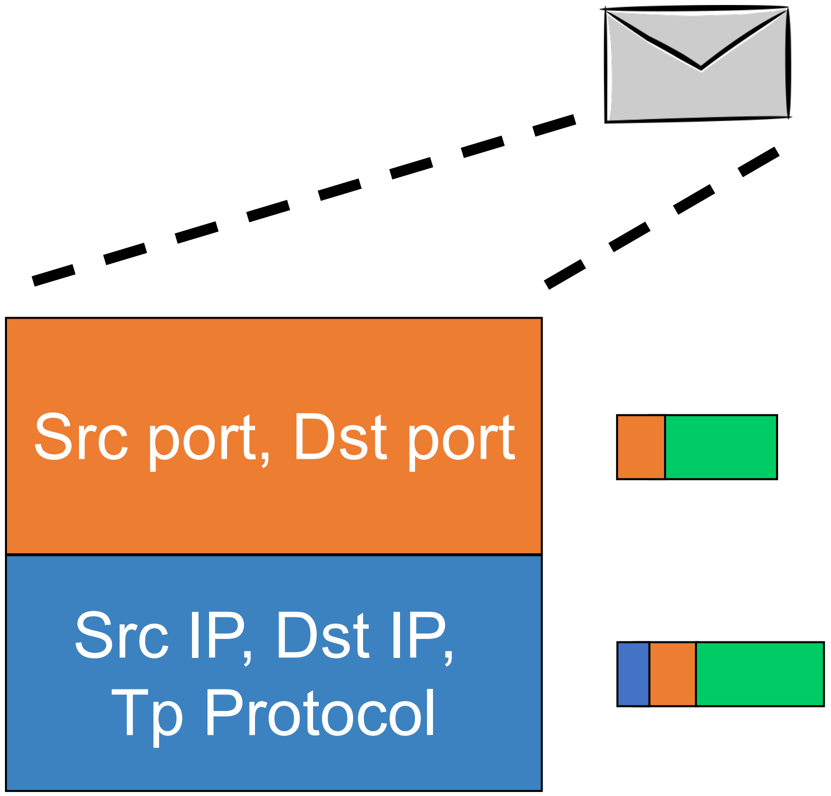
Denotes an attachment point with the network.

Each IP address comes with a full copy of its own ports.

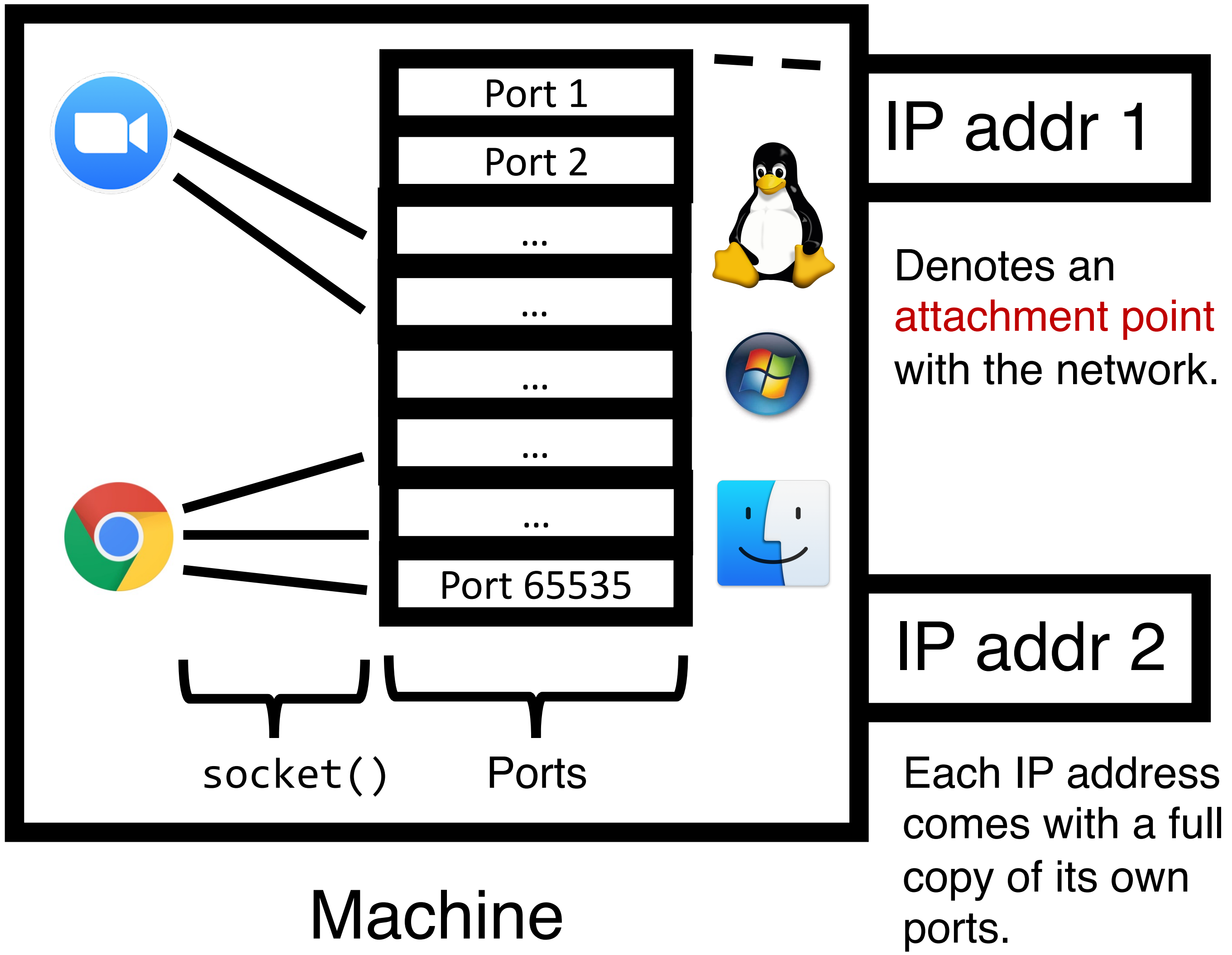


Demultiplexing

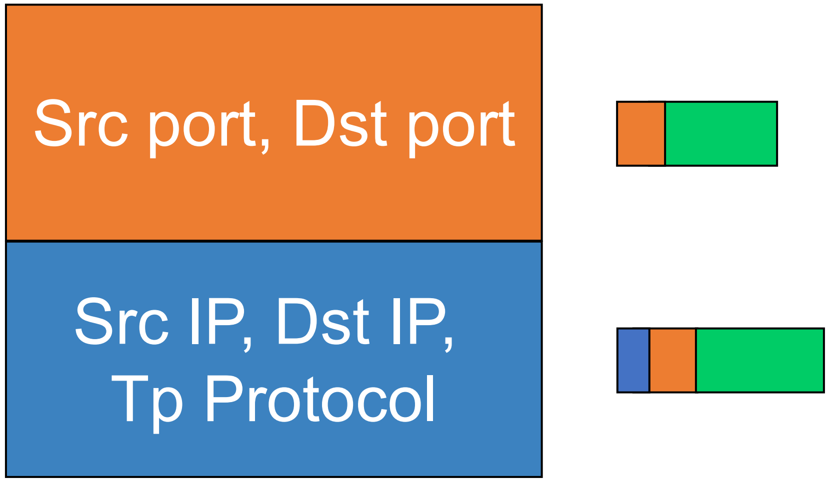




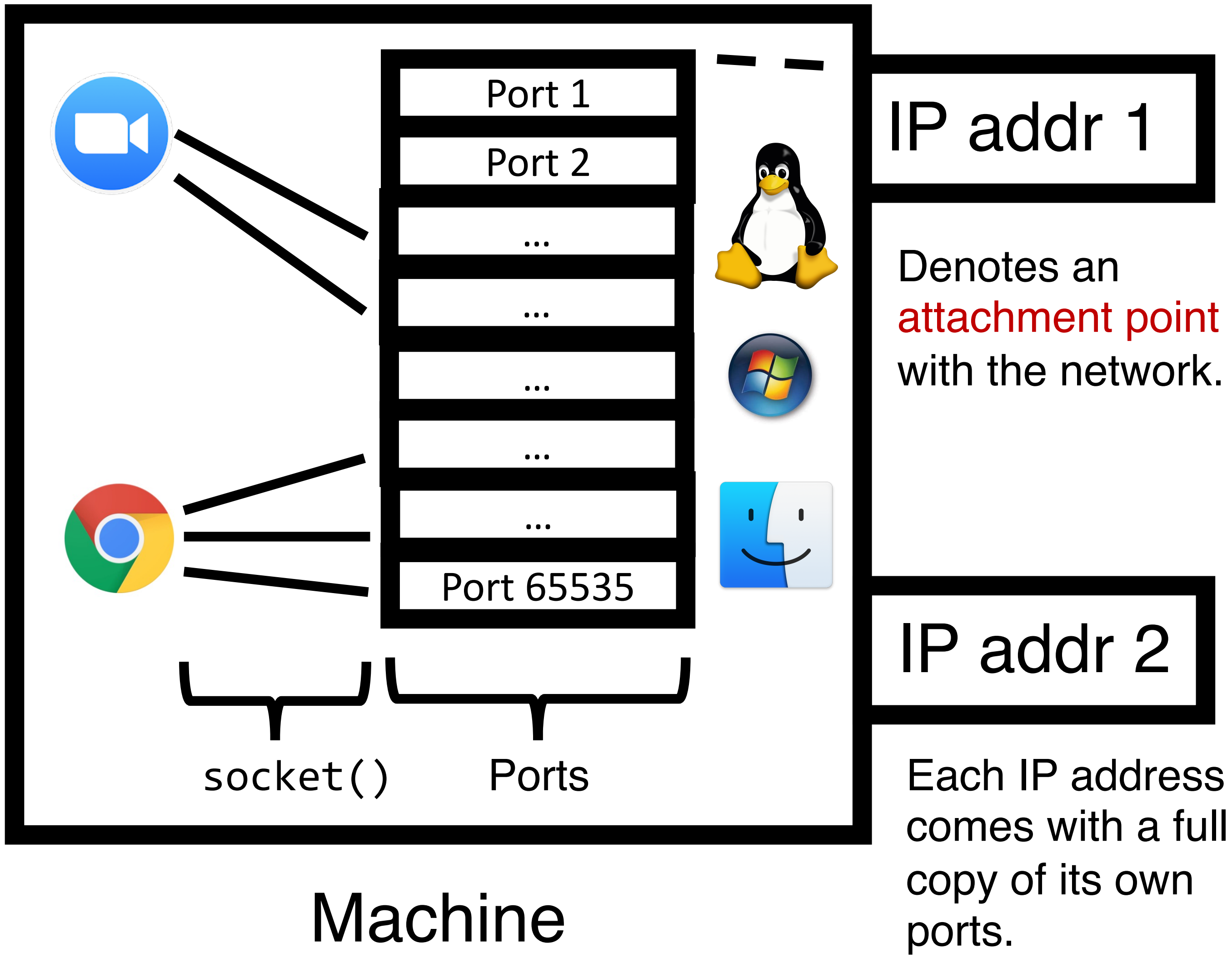
Demultiplexing



Connection lookup: The operating system does a lookup using these data to determine the right socket and app.



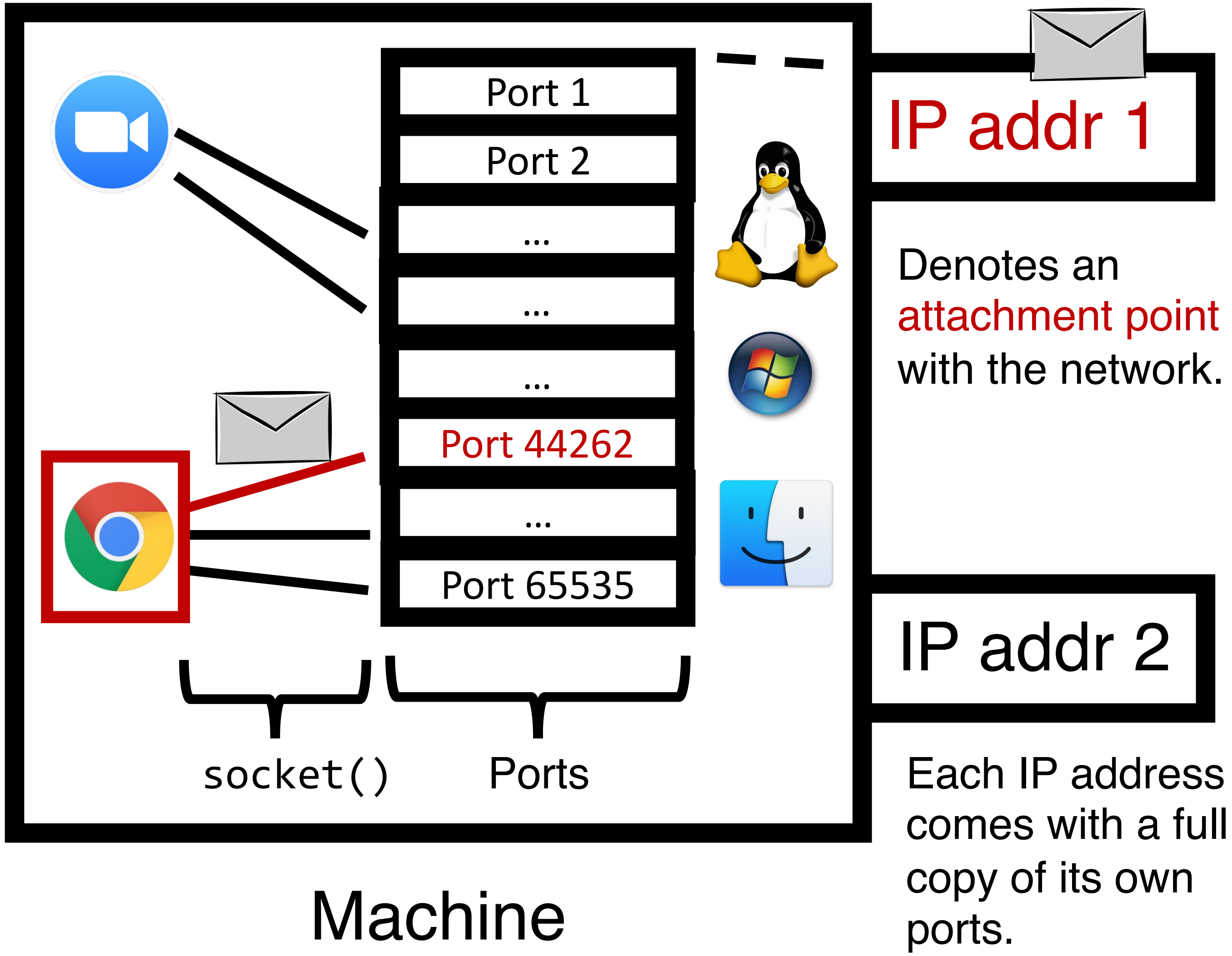
Demultiplexing



Connection lookup: The operating system does a lookup using these data to determine the right socket and app.

TCP sockets:
(src IP, dst IP, src port, dst port)
➔
Socket ID

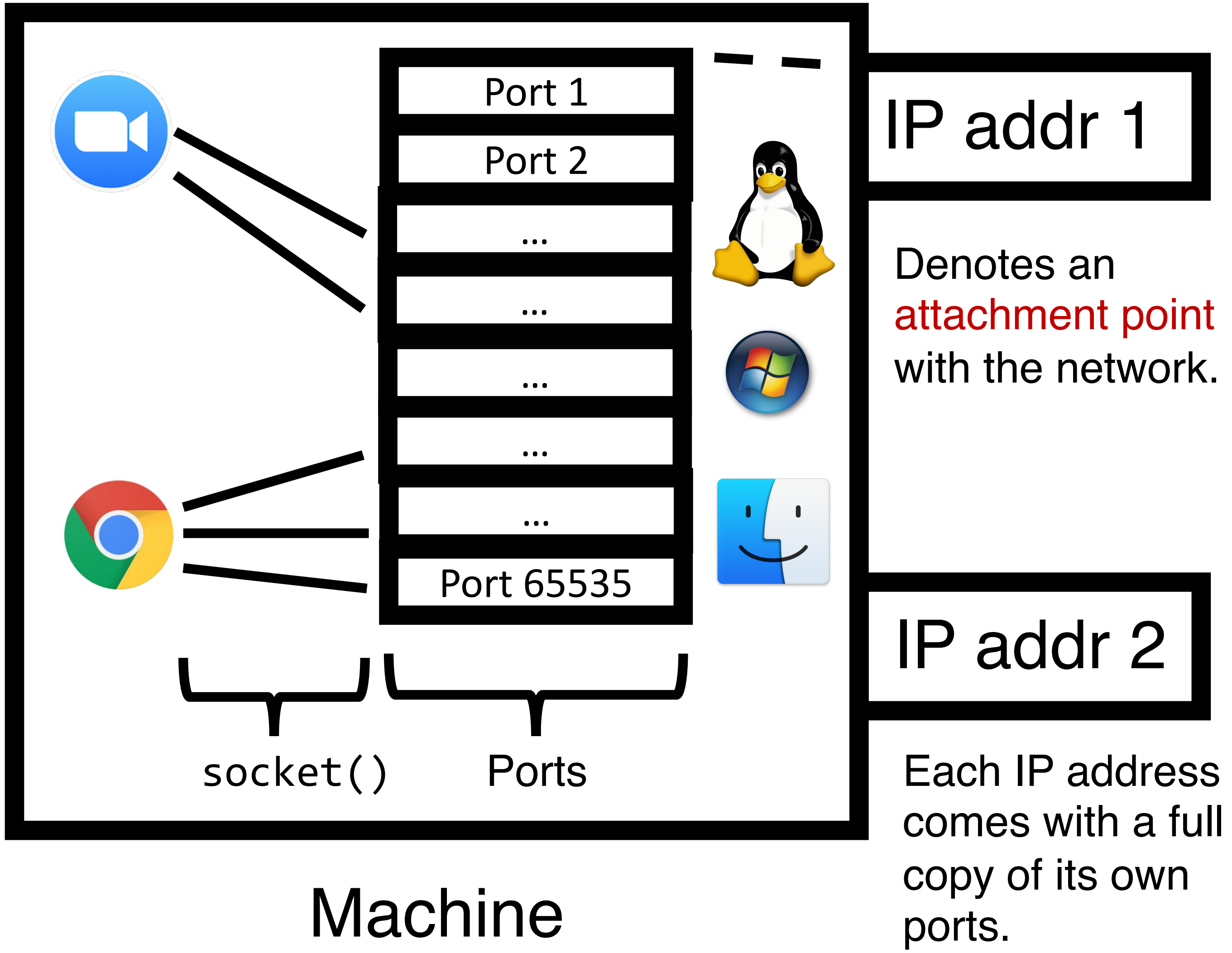
Demultiplexing



Connection lookup: The operating system does a lookup using these data to determine the right socket and app.

TCP sockets:
(src IP, dst IP, src port, dst port)
➔
Socket ID

Demultiplexing



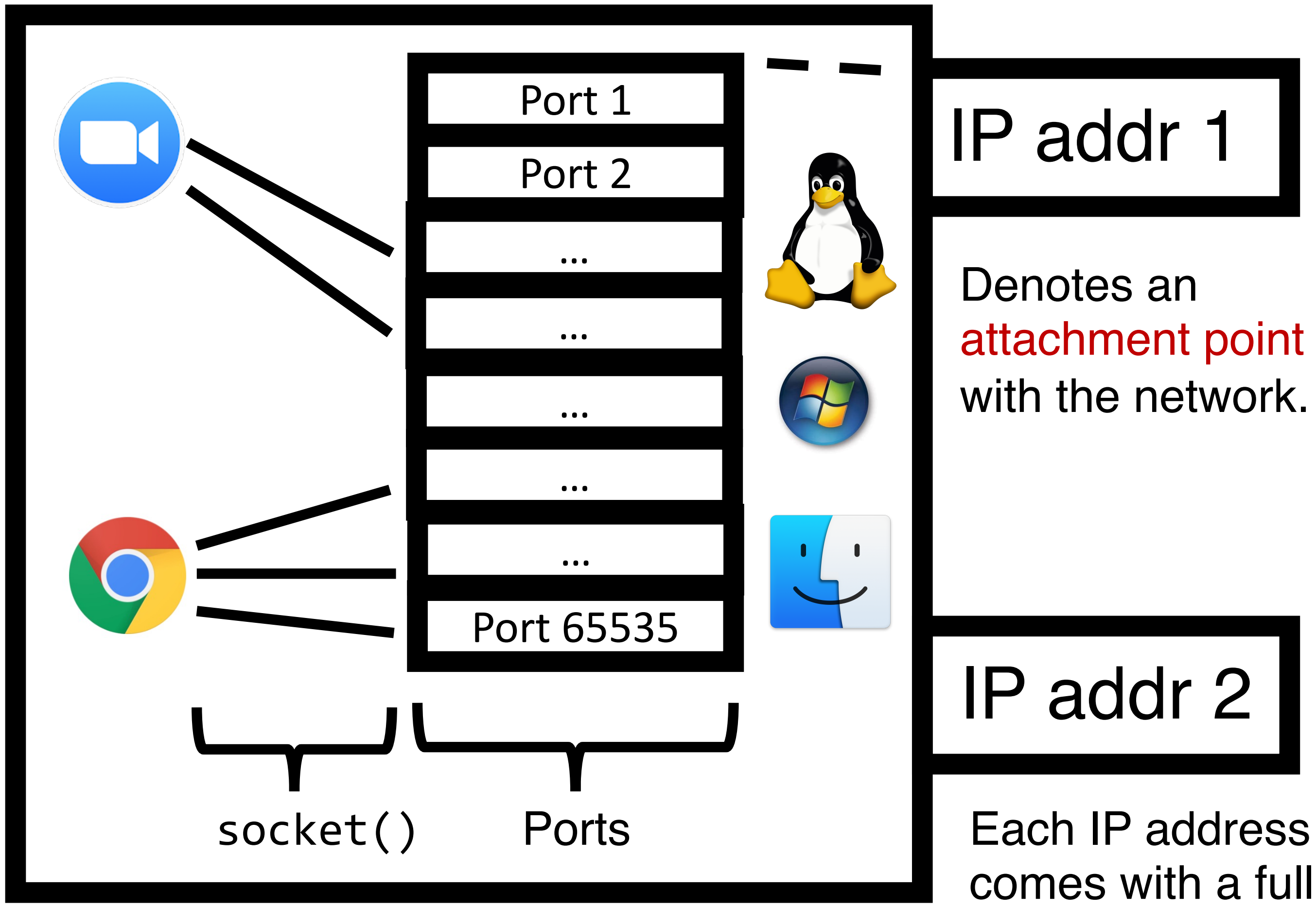
Connection lookup: The operating system does a lookup using these data to determine the right socket and app.

TCP sockets:
(src IP, dst IP, src port, dst port)
➔
Socket ID

UDP sockets:
(dst IP, dst port)
➔
Socket ID

Connectionless: the socket is shared across all sources!

Demultiplexing



Machine

Each IP address comes with a full copy of its own ports.

Connection lookup: The operating system does a lookup using these data to determine the right socket and app.

TCP sockets** Some caveats!
(src IP, dst IP, src port, dst port)
➔
Socket ID

UDP sockets:
(dst IP, dst port)
➔
Socket ID

Connectionless: the socket is shared across all sources!

TCP sockets of different types

Listening (bound but unconnected)

```
# On server side
ss = socket(AF_INET, SOCK_STREAM)
ss.bind(serv_ip, serv_port)
ss.listen() # no accept() yet
```

Connected (Established)

```
# On server side
csockid, addr = ss.accept()

# On client side
cs.connect(serv_ip, serv_port)
```

(src IP, dst IP, src port, dst port)

➔

Socket (`csockid NOT ss`)

TCP sockets of different types

Listening (bound but unconnected)

```
# On server side
ss = socket(AF_INET, SOCK_STREAM)
ss.bind(serv_ip, serv_port)
ss.listen() # no accept() yet
```

(dst IP, dst port)

➔

Socket `(ss)`

Enables new connections to be demultiplexed correctly

Connected (Established)

```
# On server side
csockid, addr = ss.accept()

# On client side
cs.connect(serv_ip, serv_port)
```

`accept()` creates a new socket with the 4-tuple (established) mapping

(src IP, dst IP, src port, dst port)

➔

Socket `(csockid NOT ss)`

Enables existing connections to be demultiplexed correctly

TCP demultiplexing

- When a TCP packet comes in, the operating system:

- Looks up table of existing connections using 4-tuple
  - If success, send to corresponding (established) socket

- If fail (no table entry), look up table of listening connections using just (dst IP, dst port)
  - If success, send to corresponding (listening) socket

- If fail again (no table entry), send error to client
  - Connection refused

UDP demultiplexing

- When a UDP packet comes in, the operating system:

- Looks up table of listening UDP sockets using (dst IP, dst port)
  - If success, send packet to corresponding socket
  - There are no established UDP sockets; they're all "unconnected"

- If fail (no table entry), send error to client
  - Port unreachable

Listing sockets and connections

- List all sockets with `ss`
- Create and observe UDP sockets with `iperf`
- Observe a TCP listening socket with `iperf` (or your own server!)

User Datagram Protocol

UDP: User Datagram Protocol [RFC 768]

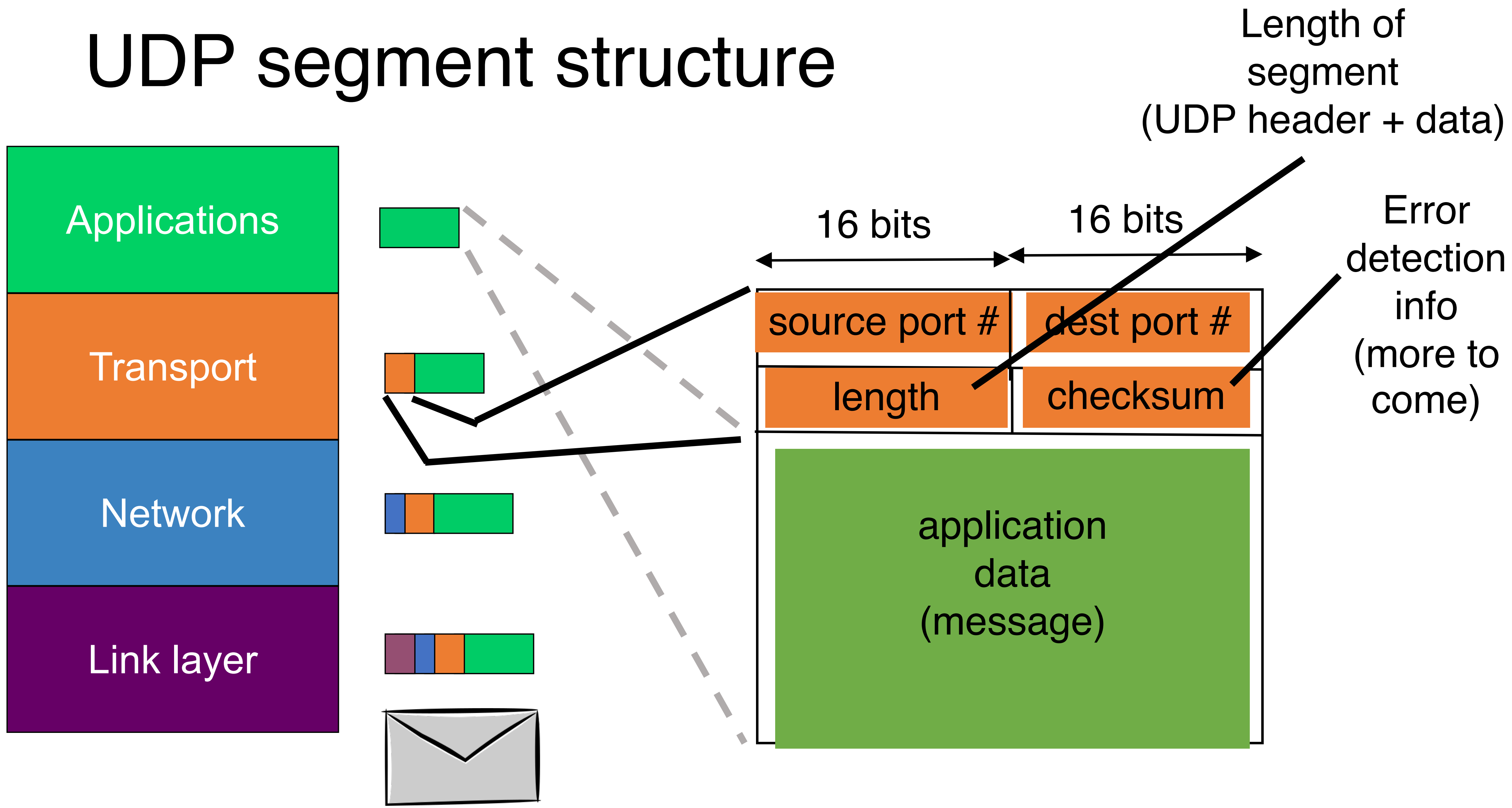
- Best effort service. UDP segments may be:
  - Lost
  - Delivered out of order to app
- UDP is connectionless
  - Each UDP segment handled independently of others (i.e. no “memory” across packets)
- Suitable for one-off req/resp
  - E.g., DNS uses UDP
- Also for loss-tolerant delay-sensitive apps, e.g., video calling

Why are UDP’s guarantees even okay?

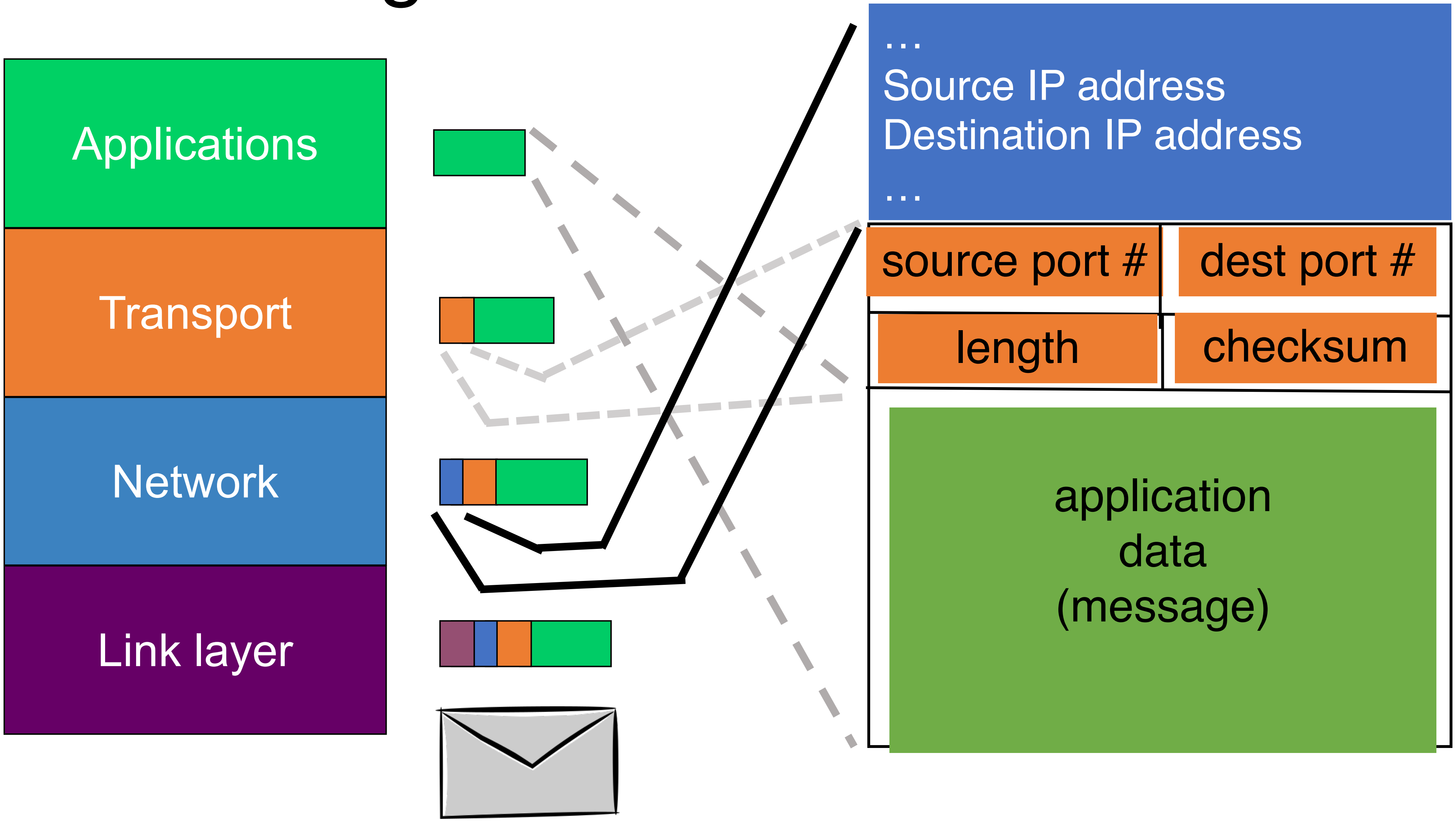
Simple & low overhead compared to TCP:

- No delays due to connection establishment
  - UDP can send data immediately
- No memory for connection state at sender & receiver
- Small segment header
- UDP can blast away data as fast as desired
  - UDP has no “congestion control”

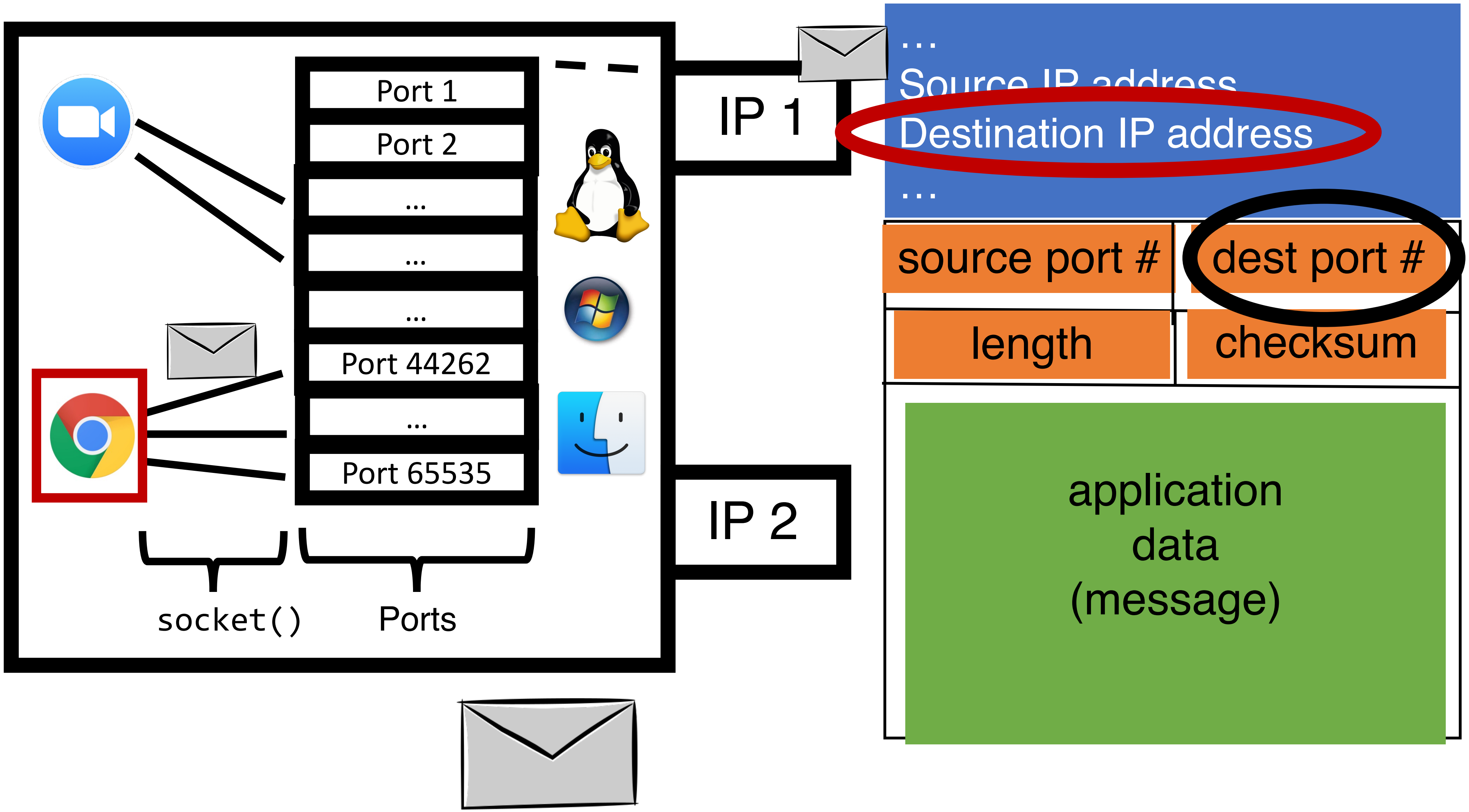
UDP segment structure



UDP segment structure



Review: UDP demultiplexing



Seeing UDP packets in action

- How to craft and send (UDP) packets?
  - It's simpler than you think!

- `sudo tcpdump -i lo udp –XAvvv # observe packets`
- `sudo scapy # tool used to send crafted packets`
- Example: `send(IP(dst="127.0.0.1")/UDP(sport=1024, dport=2048)/"hello world”, iface="lo")`
- See other fields of UDP using `UDP().fields_desc`
- Scapy can send and receive crafted packets!
  - However, it requires sudo (superuser privileges)

Error Detection

Why error detection?

- Network provides best effort service
- UDP is a simple and low overhead transport
  - Data may be lost
  - Data may be corrupted along the way (e.g., 1 -> 0)
  - Data may be reordered

- However, simple error detection is possible!
  - Was the data I received the same data the remote machine sent?

- Error detection is a useful feature for all transport protocols including TCP

Error Detection in UDP and TCP

- Key idea: have sender compute a function over the data
  - Store the result in the packet
  - Receiver can check the function's value in received packet

- An analogy: you're sending a package of goodies and want your recipient to know if goodies were leaked along the way

- Your idea: weigh the package; stamp the weight on the package
  - Have the recipient weigh the package and cross-check the weight with the stamped value

Requirements on error detection function

- Function must be easy to compute
- Function must capture the likely changes to the packet
  - If the packet was corrupted through these likely changes, the function value must change
- Function must be easy to verify

- UDP and TCP use a class of function called a checksum
  - Very common idea: used in multiple parts of networks and computer systems

UDP & TCP's Checksum function

Sender:

- treat segment contents as sequence of 16-bit integers
- checksum: addition (1's complement sum) of segment contents
- sender puts checksum value into UDP checksum field

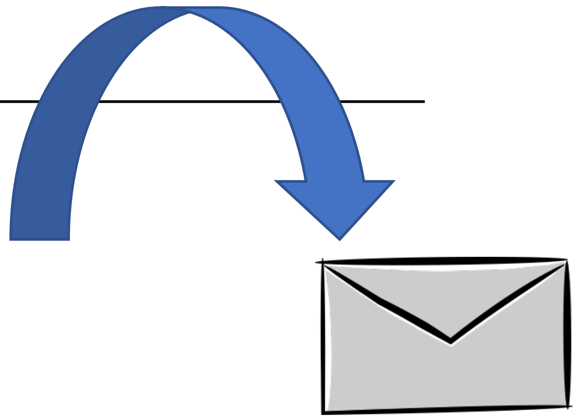
Receiver:

- compute a checksum of the received segment, including the checksum in packet itself
- check if the resulting (computed) checksum is 0
- NO – an error is detected
- YES – *assume* no error

Computing 1's complement sum

- Very similar to regular (unsigned) binary addition.
- However, when adding numbers, a carryout from the most significant bit needs to be added to the result
- Example: add two 16-bit integers

		1 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0
		1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
wraparound	1	1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1
sum		1 0 1 1 1 0 1 1 1 0 1 1 1 1 0 0
checksum		0 1 0 0 0 1 0 0 0 1 0 0 0 0 1 1



From the UDP specification (RFC 768)

- Checksum is the 16-bit one's complement of the one's complement sum of a pseudo header of information from the IP header, the UDP header, and the data, padded with zero octets at the end (if necessary) to make a multiple of two octets.

- The pseudo header conceptually prefixed to the UDP header contains the source address, the destination address, the protocol, and the UDP length.

Some observations on checksums

- Checksums don't detect all bit errors
  - Consider (x, y) vs. (x – 1, y + 1) as adjacent 16-bit values in packet
  - Analogy: you can't assume the package hasn't been meddled with if its weight matches the one on the stamp. More smarts needed for that. ☺
  - But it's a lightweight method that works well in many cases

- Checksums are part of the packet; they can get corrupted too
  - The receiver will just declare an error if it finds an error
  - However, checksums don't enable the receiver to detect where the error lies or correct the error(s)
  - Checksum is an error detection mechanism; not a correction mechanism.

Some observations on checksums

- Checksums are insufficient for reliable data delivery
  - If a packet is lost, so is its checksum
- UDP and TCP use the same checksum function
  - TCP also uses the lightweight error detection capability
  - However, TCP has more mature mechanisms for reliable data delivery (more to come on this)

Playing with checksums

Summary of UDP

- UDP is a thin shim around network layer's best-effort delivery
  - One-off request/response messages
  - Lightweight transport for loss-tolerant delay-sensitive applications

- Provides basic multiplexing/demultiplexing for application
- No reliability, performance, or ordering guarantees
- Can do basic error detection (bit flips) using checksums
  - Error detection is necessary to deliver data reliably, but it is insufficient