

Pertemuan 2 — Jaringan: IP, Socket, TCP, UDP, QUIC

Bagaimana dua program di dua mesin bertukar data

Hendri Karisma

Semester Ganjil 2026/2027

Program Studi Teknik Informatika, STMIK Tazkia

Alur pertemuan hari ini

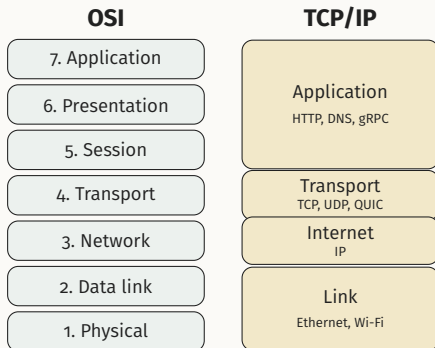
1. Lapisan jaringan: OSI dan TCP/IP
2. Alamat IP, *port*, dan DNS
3. *Socket API*: cara program memakai jaringan
4. TCP: koneksi, *three-way handshake*, keandalan
5. *Flow control* dan *congestion control*
6. *Head-of-line blocking*
7. UDP: *datagram* dan kapan dipakai
8. QUIC: transport baru di atas UDP
9. Mengamati paket: *tcpdump*, Wireshark, *tc netem*
10. Praktikum: *echo server* TCP dan UDP

Sasaran pertemuan ini

Anda dapat menjelaskan perjalanan satu pesan dari program ke program, menulis *client-server* sederhana dengan *socket*, dan membedakan TCP, UDP, dan QUIC berdasarkan apa yang **dijamin** dan apa yang **dikorbankan** masing-masing.

Lapisan jaringan: OSI dan TCP/IP

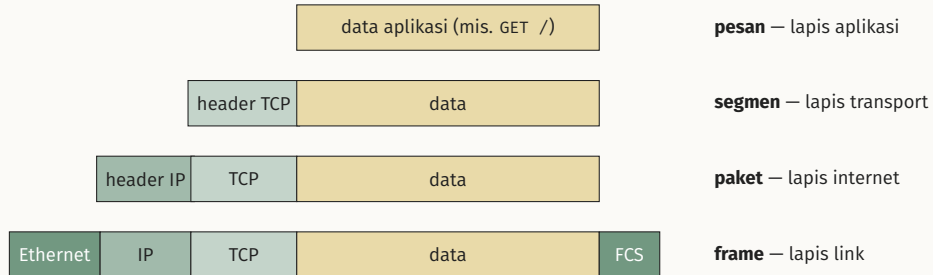
Dua model lapisan



- ▶ **OSI** adalah model rujukan tujuh lapis: bagus untuk berbicara, tetapi tidak persis diterapkan.
- ▶ **TCP/IP** adalah model yang benar-benar dipakai internet: empat lapis.
- ▶ Tiap lapis hanya berbicara dengan lapis di atas dan di bawahnya, lewat antarmuka yang jelas.
- ▶ Akibatnya, Wi-Fi bisa diganti fiber tanpa mengubah HTTP, dan HTTP bisa diganti gRPC tanpa mengubah kabel.

Istilah “L4 load balancer” dan “L7 load balancer” yang akan kita temui memakai penomoran OSI.

Enkapsulasi: setiap lapis menambah amplop



- ▶ Pengirim membungkus dari atas ke bawah; penerima membuka dari bawah ke atas.
- ▶ *Router* di tengah jalan hanya membaca sampai header IP. Ia tidak peduli isi HTTP Anda — kecuali perangkat yang sengaja memeriksa lebih dalam (*firewall, proxy, middlebox*).
- ▶ Setiap header adalah *overhead*. Untuk pesan kecil, header bisa lebih besar daripada datanya sendiri.

Alamat IP, *port*, dan DNS

Alamat IP dan *port*

Alamat IP — “mesin yang mana”

- ▶ **IPv4:** 32 bit, ditulis 192.168.1.10; sekitar 4,3 miliar alamat, sudah habis dibagikan
- ▶ **IPv6:** 128 bit, ditulis 2001:db8::1
- ▶ Alamat privat (10.0.0.0/8, 172.16.0.0/12, 192.168.0.0/16) tidak dirutekan di internet
- ▶ **NAT:** banyak mesin privat berbagi satu alamat publik

Port — “program yang mana”

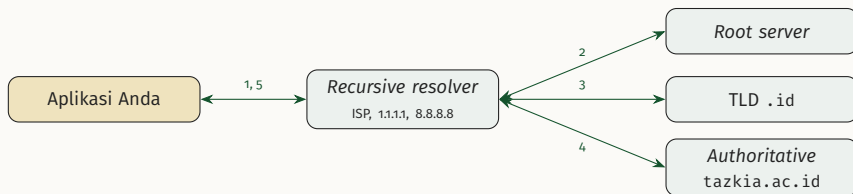
- ▶ Bilangan 16 bit: 0–65535
- ▶ *Well-known:* 22 SSH, 53 DNS, 80 HTTP, 443 HTTPS
- ▶ Klien biasanya memakai *ephemeral port* acak yang dipilih sistem operasi

Satu koneksi dikenali dari lima hal (*5-tuple*)

protokol, IP asal, *port* asal, IP tujuan, *port* tujuan.

Karena itu satu server di *port* 443 dapat melayani ribuan klien sekaligus: tiap pasangan klien membentuk *5-tuple* yang berbeda.

DNS: buku telepon internet



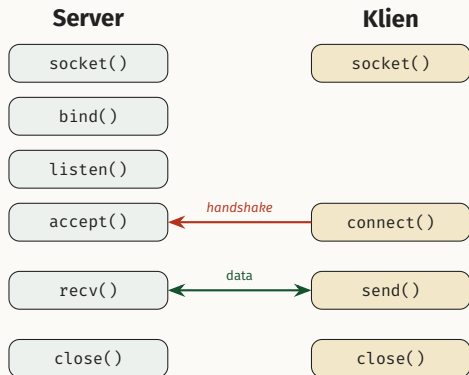
- ▶ Jawaban disimpan sementara (*cache*) selama TTL, sehingga langkah 2-4 jarang diulang.
- ▶ DNS umumnya memakai **UDP** port 53: satu pertanyaan, satu jawaban, tanpa perlu koneksi. Jawaban besar beralih ke TCP.
- ▶ DNS sendiri adalah sistem terdistribusi raksasa: hierarkis, tereplikasi, dan sangat bergantung pada *caching*.

Coba sendiri:

```
$ dig +short stmik.tazkia.ac.id
$ dig stmik.tazkia.ac.id +trace
$ dig @1.1.1.1 google.com AAAA
```

Socket API

Socket: pintu program ke jaringan



- ▶ *Socket API* lahir di BSD Unix pada awal 1980-an dan kini dipakai hampir semua sistem operasi.
- ▶ Bagi program, *socket* terlihat seperti berkas: dibaca dan ditulis.
- ▶ `accept()` mengembalikan *socket* **baru** untuk tiap klien; *socket* lama tetap mendengar.
- ▶ Semua pustaka HTTP, gRPC, dan basis data pada akhirnya memanggil fungsi-fungsi ini.

Echo server TCP dalam Go

```
package main

import ("bufio"; "log"; "net")

func main() {
    ln, err := net.Listen("tcp", ":9000")
    if err != nil { log.Fatal(err) }
    for {
        conn, err := ln.Accept()
        if err != nil { continue }
        go layani(conn) // satu goroutine per klien
    }
}

func layani(c net.Conn) {
    defer c.Close()
    s := bufio.NewScanner(c) // baca per baris
    for s.Scan() {
        c.Write(append(s.Bytes(), '\n'))
    }
}
```

- ▶ `net.Listen = socket() + bind() + listen()`.
- ▶ `Accept()` menunggu sampai ada klien yang selesai *handshake*.
- ▶ Setiap klien dilayani *goroutine* sendiri, sehingga klien lambat tidak menahan klien lain.
- ▶ Uji cepat tanpa menulis klien:
`nc localhost 9000`

Klien TCP dalam Python, sambil mengukur waktu

```
import socket, time

alamat = ("127.0.0.1", 9000)
with socket.create_connection(alamat) as s:
    f = s.makefile("rwb")      # bungkus jadi berkas
    for i in range(5):
        t0 = time.perf_counter()
        f.write(f"halo {i}\n".encode())
        f.flush()
        balas = f.readline()   # tunggu satu baris
        ms = (time.perf_counter() - t0) * 1000
        print(balas.decode().strip(), f"{ms:.2f} ms")
```

- ▶ Server ditulis Go, klien ditulis Python: keduanya hanya sepakat pada **protokol** (satu baris teks diakhiri `\n`).
- ▶ Waktu yang tercetak adalah *round-trip time* (RTT) tingkat aplikasi.
- ▶ Di *localhost*, angkanya puluhan mikrodetik. Coba lagi setelah kita menambahkan *delay* buatan dengan `tc netem`.

TCP: koneksi yang andal

Apa yang dijanjikan TCP

- ▶ **Berorientasi koneksi** — ada *handshake* sebelum data.
- ▶ **Andal** — data yang hilang dikirim ulang.
- ▶ **Berurutan** — byte tiba sesuai urutan kirim.
- ▶ **Tanpa duplikat** — data ganda dibuang.
- ▶ **Full-duplex** — dua arah sekaligus.
- ▶ **Flow control** dan **congestion control** — tidak membanjiri penerima maupun jaringan.

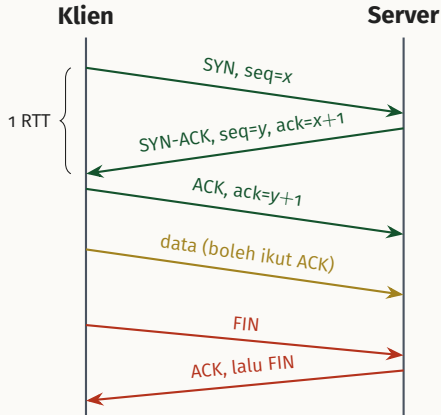
Intinya

Karena tidak ada batas pesan, setiap protokol di atas TCP harus membuat *framing* sendiri: pemisah baris (HTTP/1.1 header, *echo server* kita) atau awalan panjang (HTTP/2, gRPC).

Yang tidak dijanjikan TCP

- ▶ **Batas pesan.** TCP adalah aliran byte (*byte stream*). Dua kali `send()` bisa tiba sebagai satu `recv()`, atau satu `send()` terpecah jadi tiga.
- ▶ **Waktu tiba.** “Andal” berarti akhirnya sampai, bukan cepat sampai.
- ▶ **Kepastian diproses.** ACK hanya berarti byte sampai di *kernel* penerima, bukan sudah diproses aplikasinya.

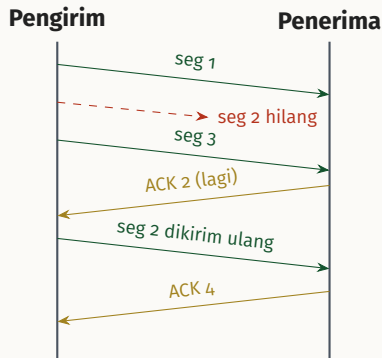
Three-way handshake dan penutupan koneksi



- ▶ Kedua pihak saling memberi tahu nomor urut awal (*initial sequence number*) dan saling mengakuinya.
- ▶ Data aplikasi baru bisa dikirim setelah **satu RTT** penuh.
- ▶ Bila RTT Jakarta–Eropa sekitar 200 ms, setiap koneksi baru “membayar” 200 ms sebelum satu byte pun terkirim — belum termasuk TLS.
- ▶ Inilah alasan koneksi dipakai ulang (*keep-alive, connection pool*) dan alasan QUIC lahir.

Bagaimana TCP menjadi andal

- ▶ **Sequence number** — setiap byte bernomor, sehingga penerima bisa mengurutkan dan membuang duplikat.
- ▶ **ACK kumulatif** — penerima memberi tahu “saya sudah terima semua sampai byte ke- n ”.
- ▶ **Retransmission timeout** (RTO) — bila ACK tidak datang dalam batas waktu yang diperkirakan dari RTT, data dikirim ulang.
- ▶ **Fast retransmit** — tiga ACK ganda untuk nomor yang sama menandakan ada segmen hilang; kirim ulang tanpa menunggu RTO.

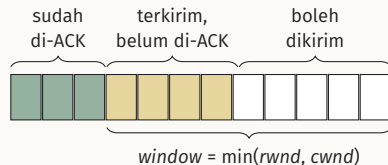


Seg 3 sudah tiba, tetapi aplikasi penerima belum boleh membacanya sebelum seg 2 datang. Ingat ini: inilah benih *head-of-line blocking*.

Flow control: jangan banjir penerima

- ▶ Penerima punya *buffer* terbatas. Bila aplikasinya lambat membaca, *buffer* penuh.
- ▶ Di setiap ACK, penerima mengiklankan **receive window** (*rwnd*): berapa byte lagi yang sanggup ia tampung.
- ▶ Pengirim tidak boleh mengirim data yang belum di-ACK melebihi *rwnd*.
- ▶ *rwnd* = 0 berarti “berhenti dulu”.

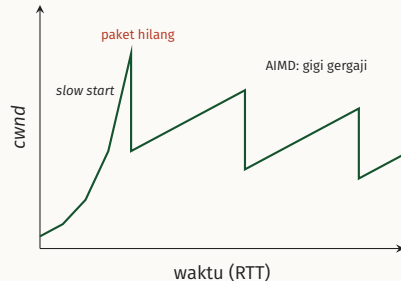
Bandingkan dengan *backpressure* pada *reactive streams* (pertemuan 6): gagasan yang sama, di lapis yang berbeda.



Sliding window: ketika ACK datang, jendela bergeser ke kanan.

Congestion control: jangan banjir jaringan

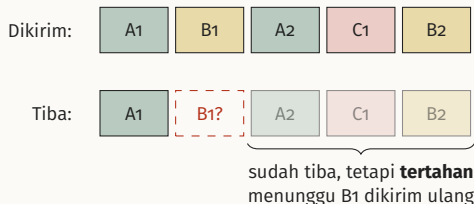
- ▶ Pengirim menjaga **congestion window** (*cwnd*) sendiri, tebakkan berapa yang sanggup dibawa jaringan.
- ▶ **Slow start** — *cwnd* berlipat tiap RTT sampai ambang tertentu.
- ▶ **AIMD** — setelah itu naik pelan (*additive increase*); bila ada paket hilang, dipotong (*multiplicative decrease*).
- ▶ Varian modern: **CUBIC** (bawaan Linux) dan **BBR** (Google), yang memperkirakan *bandwidth* dan RTT alih-alih hanya bereaksi pada kehilangan.



Intinya

Akibat praktis: koneksi baru tidak langsung secepat *bandwidth* jaringan. Ia “memanas” dulu. Koneksi yang dipakai ulang sudah panas; koneksi baru belum.

Head-of-line blocking pada TCP



- ▶ A, B, C adalah tiga permintaan berbeda yang berbagi **satu** koneksi TCP.
- ▶ Satu segmen milik B hilang. Karena TCP menjanjikan urutan byte, A2 dan C1 — yang tidak ada hubungannya dengan B — ikut menunggu.
- ▶ Pada jaringan seluler yang sering kehilangan paket, efek ini terasa sebagai “tersendat”.

Intinya

HTTP/2 memasukkan banyak permintaan ke satu koneksi TCP, sehingga terkena masalah ini. HTTP/3 di atas QUIC menyelesaikannya. Detailnya pertemuan 3.

UDP: kirim dan lupakan

UDP: sederhana mungkin

- ▶ Tidak ada koneksi, tidak ada *handshake*.
- ▶ Setiap *datagram* berdiri sendiri, dengan **batas pesan** yang terjaga.
- ▶ Header hanya 8 byte: *port* asal, *port* tujuan, panjang, *checksum*.
- ▶ Bisa hilang, tertukar urutan, atau ganda. UDP tidak peduli.
- ▶ Tidak ada *congestion control* bawaan: tanggung jawab aplikasi.

Intinya

UDP bukan “TCP yang rusak”. UDP adalah bahan mentah: aplikasi memilih sendiri jaminan apa yang ia butuhkan, dan tidak membayar jaminan yang tidak ia butuhkan.

Kapan UDP justru lebih baik

- ▶ **DNS**: satu tanya, satu jawab; ulangi saja bila hilang.
- ▶ **Video call dan VoIP**: suara yang terlambat tidak berguna; lebih baik dilewati daripada ditunggu.
- ▶ **Game online**: posisi pemain terbaru lebih penting daripada posisi yang hilang tadi.
- ▶ **QUIC**: membangun keandalannya sendiri di atas UDP.

Echo UDP dalam Python

Server

```
import socket

s = socket.socket(socket.AF_INET,
                  socket.SOCK_DGRAM)
s.bind(("0.0.0.0", 9001))
while True:
    data, alamat = s.recvfrom(2048)
    s.sendto(data, alamat)
```

Tidak ada `listen()` dan `accept()`. Satu *socket* melayani semua klien; alamat klien datang bersama setiap *datagram*.

Klien, dengan *timeout*

```
import socket

c = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
c.settimeout(0.5)           # wajib: bisa hilang
hilang = 0
for i in range(100):
    c.sendto(f"ping {i}".encode(), ("127.0.0.1", 9001))
    try:
        data, _ = c.recvfrom(2048)
    except socket.timeout:
        hilang += 1
print(f"hilang: {hilang} dari 100")
```

Di TCP, kehilangan paket disembunyikan oleh *kernel*. Di UDP, aplikasi Anda sendiri yang melihatnya — dan harus memutuskan apa yang dilakukan.

TCP vs UDP: memilih dengan sadar

	TCP	UDP
Satuan data	Aliran byte, tanpa batas pesan	<i>Datagram</i> , batas pesan terjaga
Biaya awal	<i>Handshake</i> 1 RTT	Nol
Keandalan dan urutan	Dijamin <i>kernel</i>	Urusan aplikasi
Ketika paket hilang	Semua data sesudahnya menunggu	Hanya pesan itu yang hilang
<i>Congestion control</i>	Bawaan	Harus dibuat sendiri bila perlu
Ukuran header	20 byte atau lebih	8 byte
Contoh	HTTP/1.1, HTTP/2, SSH, basis data, gRPC	DNS, VoIP, game, <i>streaming</i> langsung, QUIC

Pertanyaan yang tepat bukan “mana yang lebih cepat”, melainkan “apa yang harus terjadi pada aplikasi saya ketika satu paket hilang?”

QUIC: transport baru di atas UDP

Mengapa tidak memperbaiki TCP saja?

- ▶ **TCP tertanam di kernel.** Mengubahnya berarti menunggu miliaran perangkat memperbarui sistem operasi.
- ▶ **Ossification.** *Middlebox* di internet (*firewall*, NAT, *proxy*) memeriksa dan kadang membuang paket TCP yang “terlihat aneh”, sehingga ekstensi baru TCP sulit dipakai luas.
- ▶ **Head-of-line blocking** melekat pada janji urutan byte TCP.
- ▶ **Koneksi terikat alamat.** Ganti dari Wi-Fi ke seluler berarti alamat IP berubah, dan koneksi TCP putus.

Jalan keluar QUIC

Bangun transport baru di *user space*, di atas UDP yang dilewatkan hampir semua *middlebox*, dan enkripsi hampir seluruh header agar tidak bisa “dioprek” di tengah jalan. QUIC lahir di Google, lalu distandarkan IETF sebagai RFC 9000 (2021).

QUIC: empat gagasan utama

1. Handshake digabung TLS 1.3

Pengaturan koneksi dan enkripsi terjadi bersamaan: data bisa dikirim setelah 1 RTT. Pada koneksi ulang ke server yang sama, **0-RTT** memungkinkan data ikut di paket pertama.

2. Stream independen

Banyak *stream* dalam satu koneksi; paket hilang hanya menahan *stream* miliknya.

3. Connection migration

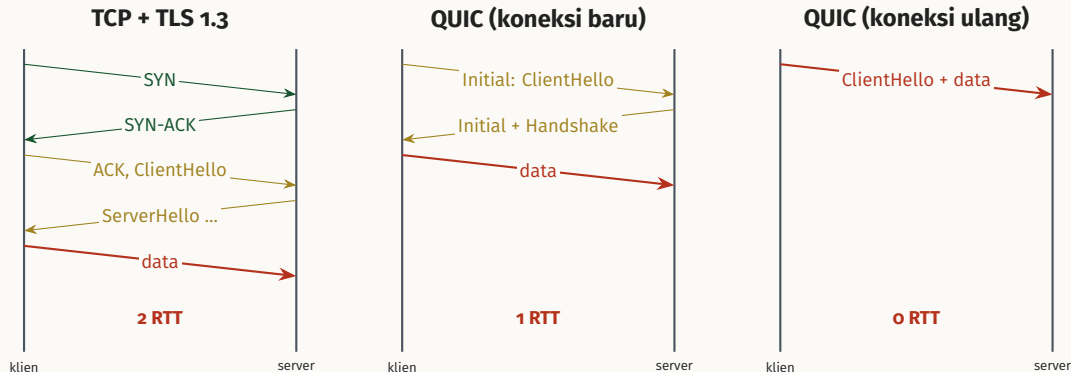
Koneksi dikenali lewat *connection ID*, bukan *5-tuple*. Ponsel berpindah jaringan, koneksi tetap hidup.

4. Enkripsi wajib, di *user space*

Tidak ada QUIC tanpa enkripsi. Karena berjalan di pustaka aplikasi, protokolnya dapat diperbarui secepat aplikasi diperbarui.

Catatan: 0-RTT punya risiko *replay*; hanya aman untuk permintaan yang *idempotent*. Kita bahas *idempotency* pada pertemuan 4.

Berapa RTT sebelum data pertama terkirim?



Dengan RTT 200 ms, selisih 2 RTT dan 1 RTT adalah 200 ms pada **setiap** koneksi baru. Untuk pengguna seluler yang sering membuka koneksi baru, ini terasa.

Mengamati paket

Alat untuk melihat jaringan

```
# siapa mendengar di port mana
$ ss -tulpn

# koneksi TCP yang sedang terbuka
$ ss -tan state established

# jalur ke tujuan, hop demi hop
$ traceroute stmik.tazkia.ac.id
$ mtr -rw google.com

# tangkap paket di port 9000
$ sudo tcpdump -i lo -nn port 9000
$ sudo tcpdump -i lo -w echo.pcap port 9000
```

Wireshark

- ▶ Buka `echo.pcap`, atau tangkap langsung di antarmuka `lo`
- ▶ Filter tampilan yang berguna:
 - `tcp.port == 9000`
 - `tcp.flags.syn == 1`
 - `tcp.analysis.retransmission`
 - `udp.port == 9001`
 - `quic` atau `dns`
- ▶ Menu *Statistics* → *Flow Graph* menggambar diagram panah seperti slide *handshake* tadi

tc netem: membuat jaringan “jelek” dengan sengaja

```
# tambah delay 100 ms dan loss 5% di loopback
$ sudo tc qdisc add dev lo root netem \
    delay 100ms loss 5%

# lihat aturan yang aktif
$ tc qdisc show dev lo

# ubah: delay bervariasi (jitter)
$ sudo tc qdisc change dev lo root netem \
    delay 100ms 20ms loss 5%

# WAJIB: hapus setelah selesai
$ sudo tc qdisc del dev lo root
```

- ▶ netem (*network emulator*) adalah fitur *kernel* Linux untuk meniru *delay*, *jitter*, *loss*, duplikasi, dan urutan tertukar.
- ▶ Pada `lo`, *delay* kena di kedua arah, sehingga RTT kira-kira dua kali nilainya.
- ▶ Lebih aman dijalankan di dalam *container* atau VM. Di WSL2 dan macOS, gunakan VM Linux.

Hati-hati

Lupa menghapus aturan membuat semua program lokal ikut lambat.

Praktikum

Praktikum: TCP vs UDP saat jaringan buruk

1. Jalankan *echo server* TCP (Go) di *port* 9000 dan *echo server* UDP (Python) di *port* 9001.
2. Jalankan `tcpdump -w` lalu klien TCP. Buka di Wireshark dan temukan SYN, SYN-ACK, ACK, data, dan FIN. Hitung RTT dari *timestamp*.
3. Ubah klien TCP agar mengirim 100 pesan; catat RTT rata-rata, median, dan maksimum.
4. Lakukan hal yang sama dengan klien UDP; catat jumlah pesan yang hilang.
5. Pasang `netem delay 50ms loss 5%`. Ulangi langkah 3 dan 4.
6. Hapus aturan `netem`.

Kondisi	TCP: median RTT	TCP: maks RTT	UDP: median RTT	UDP: hilang
Normal	...	...	...	...
delay 50ms loss 5%	...	...	...	...

Pertanyaan analisis praktikum

1. Saat *loss* 5%, mengapa TCP tetap mengembalikan 100 balasan, tetapi nilai maksimum RTT-nya melonjak jauh di atas median? Temukan bukti *retransmission* di Wireshark.
2. Mengapa UDP tidak pernah “tersendat”, tetapi sebagian pesannya tidak kembali? Untuk aplikasi apa perilaku ini lebih baik?
3. Bila 5% paket hilang di tiap arah, berapa kira-kira persen *ping* UDP yang tidak kembali? Bandingkan dengan hasil Anda.
4. Kirim dua pesan dengan dua kali `send()` berturut-turut tanpa jeda ke server TCP yang membaca dengan `recv(1024)` biasa. Apakah keduanya tiba sebagai satu potong? Apa artinya bagi desain protokol?

Intinya

Laporan yang baik tidak hanya berisi angka, tetapi menjelaskan **mengapa** angkanya demikian, dengan bukti dari tangkapan paket.

- ▶ Jaringan berlapis; setiap lapis menambah header dan menyembunyikan lapis di bawahnya.
- ▶ Satu koneksi dikenali dari *5-tuple*; DNS menerjemahkan nama menjadi alamat.
- ▶ *Socket* adalah antarmuka program ke jaringan; semua protokol tingkat tinggi berdiri di atasnya.
- ▶ TCP memberi aliran byte yang andal dan berurutan, dengan harga *handshake*, *head-of-line blocking*, dan tanpa batas pesan.
- ▶ *Flow control* melindungi penerima; *congestion control* melindungi jaringan.
- ▶ UDP memberi *datagram* tanpa jaminan; aplikasi memilih jaminannya sendiri.
- ▶ QUIC membangun keandalan, *stream* independen, dan enkripsi di atas UDP, dan menjadi dasar HTTP/3.

Intinya

Hubungkan dengan pertemuan 1: *fallacy* “jaringan itu andal” dan “*latency* itu nol” sekarang punya wujud nyata — *retransmission* dan RTT.

Tugas untuk pertemuan berikutnya

1. Selesaikan praktikum dan kumpulkan laporan: kode, tabel hasil, potongan tangkapan layar Wireshark, dan jawaban empat pertanyaan analisis.
2. Baca Kurose & Ross Bab 3 (transport) dan van Steen & Tanenbaum Bab 4.1–4.2.
3. Perluas *echo server* TCP: buat protokol sederhana dengan *framing* awalan panjang (4 byte panjang pesan, lalu isi pesan), dan uji dengan pesan berukuran 1 MB.
4. Pasang `curl` versi terbaru dan periksa apakah ia mendukung HTTP/3 (`curl -version`, cari kata HTTP3).

Pertemuan depan

Kita naik satu lapis: HTTP/1.1, HTTP/2, dan HTTP/3, serta TLS yang membuat huruf “S” pada HTTPS — dan cara mengukur pengaruh semuanya pada waktu muat halaman.

Terima kasih

Pertanyaan?