

Pertemuan 3 – HTTP/1.1, HTTP/2, HTTP/3, TLS dan HTTPS

Protokol yang membawa hampir semua lalu lintas sistem terdistribusi modern

Hendri Karisma

Semester Ganjil 2026/2027

Program Studi Teknik Informatika, STMIK Tazkia

Alur pertemuan hari ini

1. Anatomi HTTP: metode, status, header
2. *Cookie* dan *caching*
3. HTTP/1.1: *keep-alive*, *pipelining*, dan batasnya
4. HTTP/2: *framing* biner, *multiplexing*, HPACK
5. HTTP/3 di atas QUIC
6. Mengukur waktu muat dengan `curl`
7. TLS 1.3: *handshake*, sertifikat X.509, CA
8. mTLS antar layanan
9. *Reverse proxy*, *load balancer* L4 vs L7, CDN
10. Praktikum: HTTP/1.1 vs /2 vs /3 dan HTTPS lokal

Sasaran pertemuan ini

Anda dapat membaca *request* dan *response* HTTP, menjelaskan masalah apa yang diselesaikan tiap versi HTTP, menjelaskan bagaimana TLS membuktikan identitas server, dan mengukur sendiri pengaruh semua itu pada waktu muat.

Anatomi HTTP

Satu request, satu response

Request

```
GET /api/produk/42 HTTP/1.1
Host: toko.contoh.id
Accept: application/json
User-Agent: curl/8.5.0
Cookie: sesi=a8f3c1
```

Baris pertama: metode, *path*, versi. Lalu header, satu baris kosong, lalu *body* (kosong untuk GET).

Response

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 41
Cache-Control: max-age=60
ETag: "v7"
```

```
{"id": 42, "nama": "Kopi", "harga": 25000}
```

Baris pertama: versi dan kode status. Content-Length adalah *framing*: penerima tahu kapan *body* selesai.

Intinya

HTTP/1.1 adalah **teks** yang bisa dibaca manusia — Anda bisa mengetiknya langsung lewat `nc toko.contoh.id 80`. HTTP/2 dan HTTP/3 membawa makna yang sama, tetapi dalam bentuk biner.

Metode dan kode status

Metode	Makna	Safe	Idempotent
GET	ambil	ya	ya
HEAD	ambil header saja	ya	ya
POST	buat / proses	tidak	tidak
PUT	ganti seluruhnya	tidak	ya
PATCH	ubah sebagian	tidak	tidak*
DELETE	hapus	tidak	ya

Safe: tidak mengubah apa pun. *Idempotent*: diulang berkali-kali hasil akhirnya sama. *PATCH bisa dirancang *idempotent*, tetapi tidak dijamin.

Kelas	Contoh
1xx informasi	101 <i>Switching Protocols</i>
2xx berhasil	200 OK, 201 <i>Created</i> , 204
3xx alihkan	301, 302, 304 <i>Not Modified</i>
4xx salah klien	400, 401, 403, 404, 429
5xx salah server	500, 502, 503, 504

502/503/504 sering berasal dari *proxy* atau *load balancer*, bukan dari aplikasi Anda: tanda khas sistem terdistribusi.

Intinya

Mengapa *idempotent* penting? Karena di jaringan, *retry* itu wajar. Mengulang GET aman; mengulang POST “bayar” bisa menagih dua kali.

HTTP itu *stateless*

Setiap *request* berdiri sendiri; server tidak wajib ingat *request* sebelumnya. Ini yang membuat server HTTP mudah diskalakan: *request* mana pun bisa dilayani server mana pun.

Cookie menambahkan “ingatan” di sisi klien: server mengirim Set-Cookie, peramban mengembalikannya di setiap *request* berikutnya. Atribut penting: HttpOnly, Secure, SameSite.

Caching: request tercepat adalah yang tidak dikirim

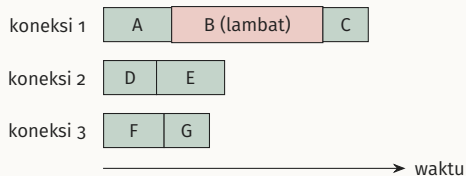
- ▶ Cache-Control: max-age=60 — boleh dipakai ulang 60 detik tanpa bertanya
- ▶ Cache-Control: no-store — jangan simpan (data sensitif)
- ▶ ETag + If-None-Match — tanya “masih versi v7?”; bila ya, server menjawab **304** tanpa *body*
- ▶ Cache bisa ada di peramban, *proxy*, dan CDN

Ingat pepatah lama: dua hal tersulit dalam ilmu komputer adalah *cache invalidation* dan memberi nama.

HTTP/1.1: satu permintaan pada satu waktu

Keep-alive dan pipelining

- ▶ **HTTP/1.0:** secara bawaan satu koneksi TCP per *request*. Setiap gambar membayar *handshake* dan *slow start* lagi.
- ▶ **HTTP/1.1 keep-alive:** koneksi dipakai ulang untuk *request* berikutnya. Jauh lebih hemat.
- ▶ Tetapi dalam satu koneksi, *request* berikutnya harus menunggu *response* sebelumnya selesai.
- ▶ **Pipelining:** kirim beberapa *request* tanpa menunggu. Namun *response* wajib kembali **berurutan**; satu *response* lambat menahan yang lain. Peramban praktis tidak pernah mengaktifkannya.



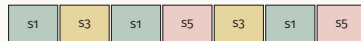
Peramban menyiasatinya dengan membuka sekitar 6 koneksi paralel per *host*, bahkan memecah situs ke banyak *subdomain* (*domain sharding*) dan menggabung berkas (*sprite*, *bundling*). Semua itu adalah tambalan.

HTTP/2: banyak *stream*, satu koneksi

HTTP/2: *framing* biner dan *multiplexing*

- ▶ Distanstandarkan 2015 (kini RFC 9113). Makna HTTP sama; cara membawanya berbeda.
- ▶ Pesan dipecah menjadi **frame** biner (HEADERS, DATA, SETTINGS, ...), tiap *frame* membawa nomor **stream**.
- ▶ Banyak *stream* berjalan selang-seling dalam **satu** koneksi TCP: tidak perlu 6 koneksi lagi.
- ▶ **HPACK** mengompresi header; header yang sama (Cookie, User-Agent) tidak dikirim berulang-ulang.
- ▶ **Server push**: server mengirim berkas sebelum diminta. Dalam praktik jarang menguntungkan, dan peramban besar seperti Chrome sudah menghapusnya.

satu koneksi TCP



stream 1, 3, 5 = tiga request yang frame-nya diselang-seling

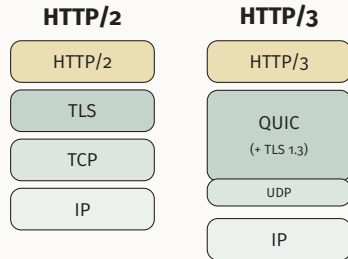
Tetapi...

Semua *stream* masih berbagi satu aliran byte TCP. Satu paket TCP hilang menahan **semua** *stream* — *head-of-line blocking* pindah dari lapis HTTP ke lapis TCP. Pada jaringan yang sering kehilangan paket, HTTP/2 bisa lebih buruk daripada HTTP/1.1 dengan 6 koneksi.

HTTP/3: HTTP di atas QUIC

HTTP/3: menyelesaikan masalah di lapis transport

- ▶ Distandarkan sebagai RFC 9114 (2022), berjalan di atas QUIC (pertemuan 2).
- ▶ Tiap *request* = satu *stream* QUIC. Paket hilang hanya menahan *stream* miliknya.
- ▶ **QPACK** menggantikan HPACK, karena HPACK mengandalkan urutan yang tidak lagi dijamin lintas *stream*.
- ▶ *Handshake* transport dan TLS 1.3 menyatu: 1 RTT, atau 0 RTT untuk koneksi ulang.
- ▶ Peramban biasanya **menemukan** HTTP/3 lewat header *Alt-Svc* dari *response* HTTP/2 pertama, lalu mencoba UDP 443 berikutnya.



Catatan lapangan

Sebagian jaringan kantor memblokir UDP 443. Klien lalu kembali ke HTTP/2 di atas TCP. HTTP/3 adalah peningkatan, bukan pengganti total.

Tiga versi dalam satu tabel

	HTTP/1.1	HTTP/2	HTTP/3
Format	Teks	Biner (<i>frame</i>)	Biner (<i>frame</i>)
Transport	TCP	TCP	QUIC di atas UDP
Enkripsi	Opsional (HTTPS)	Praktis wajib di peramban	Wajib
<i>Request</i> paralel	Banyak koneksi	Banyak <i>stream</i> , 1 koneksi	Banyak <i>stream</i> , 1 koneksi
<i>Head-of-line blocking</i>	Di lapis HTTP	Di lapis TCP	Hanya per <i>stream</i>
Kompresi header	Tidak ada	HPACK	QPACK
<i>Handshake</i> + TLS 1.3	2 RTT	2 RTT	1 RTT (0 RTT ulang)
Ganti jaringan	Koneksi putus	Koneksi putus	<i>Connection migration</i>

Perhatikan polanya: setiap versi memindahkan satu hambatan. Tidak ada versi yang “gratis”; HTTP/3 misalnya memakan CPU lebih banyak karena enkripsi dan pengolahan paket dilakukan di *user space*.

Mengukur sendiri dengan curl

```
$ cat > fmt.txt <<'EOF'
dns=%{time_namelookup} tcp=%{time_connect} tls=%{time_appconnect}
ttfb=%{time_starttransfer} total=%{time_total} versi=%{http_version}\n
EOF
$ curl -so /dev/null -w @fmt.txt --http1.1 https://www.cloudflare.com/
$ curl -so /dev/null -w @fmt.txt --http2 https://www.cloudflare.com/
$ curl -so /dev/null -w @fmt.txt --http3 https://www.cloudflare.com/
# --http3 hanya ada bila curl dibangun dengan dukungan HTTP/3

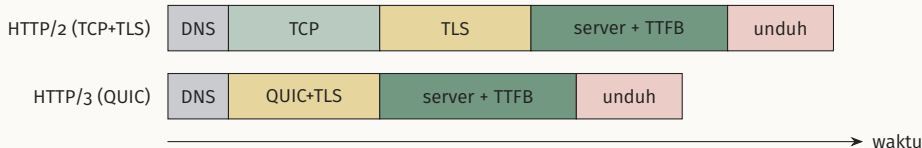
$ curl -sI --http2 https://www.cloudflare.com/ | grep -i alt-svc # iklan HTTP/3
```

time_namelookup	selesai DNS
time_connect	selesai <i>handshake</i> TCP
time_appconnect	selesai <i>handshake</i> TLS
time_starttransfer	byte pertama tiba (TTFB)
time_total	selesai semua

Intinya

Semua angka bersifat kumulatif sejak awal. Jalankan tiap varian minimal 10 kali dan laporkan median, bukan satu kali saja.

Membaca hasilnya: ke mana waktu habis?



Ilustrasi proporsi, bukan hasil pengukuran.

- ▶ Bila server jauh (RTT besar), porsi *handshake* mendominasi: HTTP/3 dan *connection reuse* paling terasa manfaatnya.
- ▶ Bila server lambat berpikir (*query* basis data berat), yang membesar adalah TTFB, dan versi HTTP tidak banyak menolong.
- ▶ Bila berkas besar, *bandwidth* dan *congestion control* yang menentukan.

Intinya

Ukur dulu, baru optimasi. Memilih HTTP/3 untuk masalah *query* lambat adalah memperbaiki bagian yang salah.

TLS dan HTTPS

Tiga janji TLS

Kerahasiaan

Isi pesan dienkripsi. Penyadap di Wi-Fi kafe hanya melihat byte acak.

Integritas

Pesan yang diubah di tengah jalan akan ketahuan dan ditolak.

Autentikasi

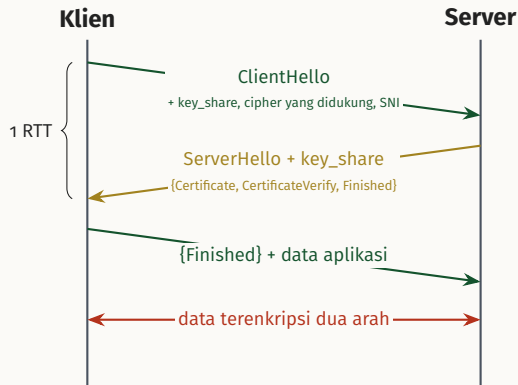
Server membuktikan ia benar pemilik nama domain itu, lewat sertifikat.

- ▶ **HTTPS** = HTTP di dalam TLS, biasanya *port* 443.
- ▶ Versi yang dipakai kini TLS 1.2 dan **TLS 1.3** (RFC 8446, 2018). SSL dan TLS versi lama dianggap tidak aman.
- ▶ Kriptografi asimetris (kunci publik) dipakai saat *handshake* untuk sepakat kunci; setelah itu data dienkripsi dengan kunci simetris yang jauh lebih cepat.

Yang tidak disembunyikan TLS

Alamat IP tujuan, ukuran dan waktu paket, dan — pada kebanyakan koneksi — nama domain lewat SNI. Isi pesan aman; metadata tidak sepenuhnya.

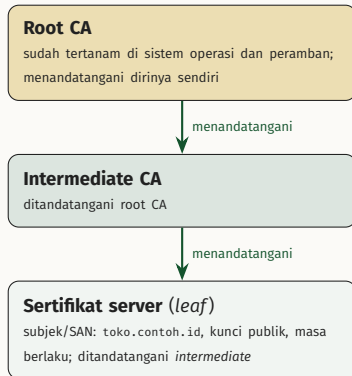
Handshake TLS 1.3 (sederhana)



Kurung kurawal = sudah terenkripsi.

- ▶ Klien langsung mengirim bagian kunci Diffie–Hellman-nya (`key_share`), menebak algoritma yang akan disetujui server.
- ▶ Setelah `ServerHello`, kedua pihak sudah bisa menghitung kunci bersama; sisanya terenkripsi, termasuk sertifikat.
- ▶ `CertificateVerify`: server menandatangani transkrip *handshake* dengan kunci privatnya, membuktikan ia memegang kunci sertifikat.
- ▶ TLS 1.2 butuh 2 RTT; TLS 1.3 cukup 1 RTT.

Sertifikat X.509 dan rantai kepercayaan



Klien memeriksa:

1. Nama domain cocok dengan SAN di sertifikat.
2. Sertifikat belum kedaluwarsa.
3. Setiap tanda tangan sah, terus naik sampai ke root CA yang dipercaya.
4. Sertifikat belum dicabut (*revocation*).

Let's Encrypt menerbitkan sertifikat gratis secara otomatis lewat protokol ACME, sehingga HTTPS kini menjadi bawaan, bukan kemewahan.

Intinya

Kepercayaan tidak dibangun dari nol di setiap koneksi; ia **didelegasikan** ke segelintir CA. Itu kekuatan sekaligus titik lemahnya.

Melihat sertifikat dengan openssl

```
# jalankan handshake TLS dan tampilkan rantai sertifikat
$ openssl s_client -connect www.google.com:443 -servername www.google.com </dev/null

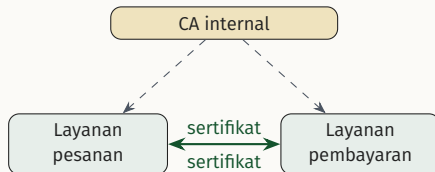
# simpan sertifikat server, lalu baca isinya
$ openssl s_client -connect www.google.com:443 -servername www.google.com </dev/null \
  | openssl x509 -noout -subject -issuer -dates -ext subjectAltName

# versi TLS dan cipher yang disepakati
$ curl -vI https://www.google.com 2>&1 | grep -E "SSL connection|subject|issuer"
```

- ▶ Perhatikan bagian Certificate chain: biasanya dua sampai tiga tingkat.
- ▶ -servername mengirim SNI. Tanpanya, server yang menampung banyak domain bisa mengembalikan sertifikat yang salah.

mTLS: saling membuktikan identitas antar layanan

- ▶ Pada HTTPS biasa, hanya **server** yang membuktikan identitas. Klien dibuktikan belakangan, lewat kata sandi atau token.
- ▶ Pada **mutual TLS** (mTLS), **klien juga** menyerahkan sertifikat.
- ▶ Di *microservices*, “klien” adalah layanan lain. Dengan mTLS, layanan pembayaran dapat memastikan yang memanggilnya benar layanan pesanan, bukan penyusup yang berhasil masuk jaringan internal.
- ▶ Inilah salah satu pilar **zero trust**: jaringan internal tidak otomatis dipercaya.



Mengelola ribuan sertifikat yang cepat kedaluwarsa itu berat. **Service mesh** seperti Istio dan Linkerd menerbitkan, memasang, dan memperbarui sertifikat mTLS secara otomatis lewat *sidecar proxy*.

HTTPS lokal: mkcert dan server Go

```
# buat CA lokal dan pasang ke
# trust store sistem/peramban
$ mkcert -install

# terbitkan sertifikat untuk localhost
$ mkcert localhost 127.0.0.1
# -> localhost+1.pem
# -> localhost+1-key.pem

$ go run server.go
$ curl -v https://localhost:8443/
```

```
package main

import ("fmt"; "log"; "net/http")

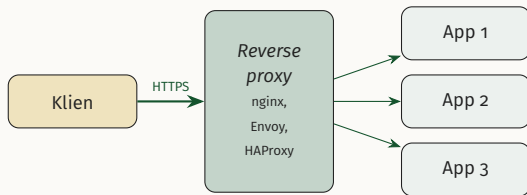
func main() {
    http.HandleFunc("/", func(w http.ResponseWriter,
        r *http.Request) {
        fmt.Fprintf(w, "halo, protokol: %s\n", r.Proto)
    })
    // dengan TLS, Go otomatis menawarkan HTTP/2 (ALPN)
    log.Fatal(http.ListenAndServeTLS(":8443",
        "localhost+1.pem", "localhost+1-key.pem", nil))
}
```

Coba `curl -http1.1` dan `curl -http2`: jawaban `r.Proto` berubah antara `HTTP/1.1` dan `HTTP/2.0`.

Tanpa `mkcert`: `openssl req -x509 -newkey rsa:2048 -nodes -keyout key.pem -out cert.pem -days 30 -subj "/CN=localhost"`, tetapi klien akan menolak sertifikat itu kecuali diberi `-k` atau CA-nya dipasang.

Di depan server: *proxy, load balancer*, CDN

Reverse proxy



Klien hanya melihat satu alamat; di belakangnya bisa ada banyak server.

Tugas yang biasa diambil alih *reverse proxy*:

- ▶ **TLS termination:** mengurus HTTPS sekali di depan, ke belakang boleh HTTP biasa (atau mTLS lagi)
- ▶ Membagi beban ke banyak server aplikasi
- ▶ *Caching* dan kompresi
- ▶ Membatasi laju (*rate limiting*) dan menyaring permintaan jahat
- ▶ Menerjemahkan protokol: HTTP/3 di depan, HTTP/1.1 ke belakang

Load balancer: L4 vs L7

	L4 (transport)	L7 (aplikasi)
Yang dilihat	IP dan <i>port</i> (5-tuple)	Isi HTTP: URL, header, <i>cookie</i>
Keputusan	Per koneksi	Per <i>request</i>
TLS	Diteruskan apa adanya	Biasanya dibuka (<i>termination</i>)
Kemampuan	Sangat cepat, murah, protokol apa saja	Rute per <i>path</i> , <i>canary</i> , <i>retry</i> , <i>header</i>
Contoh	IPVS, AWS NLB, Maglev	nginx, Envoy, HAProxy (mode http), AWS ALB

Algoritma pembagian beban:

- ▶ *Round robin* dan *weighted round robin*
- ▶ *Least connections*
- ▶ *Hash* dari IP atau kunci (*consistent hashing*, pertemuan 7)

Intinya

Health check adalah separuh nilai *load balancer*: server yang mati dikeluarkan dari daftar secara otomatis.

Contoh: nginx sebagai *load balancer* L7

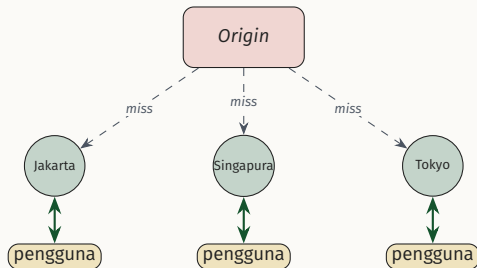
```
upstream katalog {
    least_conn;
    server 10.0.0.11:8080 max_fails=3 fail_timeout=10s;
    server 10.0.0.12:8080 max_fails=3 fail_timeout=10s;
    server 10.0.0.13:8080 backup;
}

server {
    listen 443 ssl;
    server_name toko.contoh.id;
    ssl_certificate      /etc/ssl/toko.pem;
    ssl_certificate_key  /etc/ssl/toko-key.pem;

    location /api/ {
        proxy_pass http://katalog;
        proxy_set_header X-Forwarded-For $remote_addr;
        proxy_read_timeout 5s;
    }
}
```

- ▶ `least_conn`: kirim ke server dengan koneksi aktif paling sedikit.
- ▶ `max_fails` dan `fail_timeout`: *health check* pasif; server yang gagal beruntun diistirahatkan sementara.
- ▶ `backup`: hanya dipakai bila yang lain tidak tersedia.
- ▶ `proxy_read_timeout`: jangan menunggu selamanya — *fallacy* “*latency itu nol*”.

CDN: bawa konten mendekati pengguna



Lingkaran = *edge* / PoP (*point of presence*).

- ▶ CDN menaruh salinan konten di ratusan *edge* di seluruh dunia.
- ▶ *Cache hit*: dijawab dari *edge* terdekat, RTT kecil. *Cache miss*: *edge* mengambil dari *origin*, lalu menyimpannya.
- ▶ Pengguna diarahkan ke *edge* terdekat lewat DNS dan/atau **anycast** (satu alamat IP diumumkan dari banyak lokasi).
- ▶ Bonus: menyerap serangan DDoS, TLS ditangani di *edge*, sering mendukung HTTP/3 lebih dulu.

Intinya

CDN adalah jawaban atas batas yang tidak bisa dibeli: kecepatan cahaya. Bila jarak tidak bisa diperpendek, salinannya yang didekatkan.

Praktikum

Praktikum 1: HTTP/1.1 vs HTTP/2 vs HTTP/3

1. Pilih tiga situs: satu di Indonesia, satu di Asia, satu di Eropa/Amerika. Pastikan minimal satu mendukung HTTP/3 (periksa header `alt-svc`).
2. Untuk tiap situs dan tiap versi, jalankan `curl -w 10` kali; catat median DNS, TCP, TLS, TTFB, dan total.
3. Buka situs yang sama di peramban, *DevTools* → *Network*, aktifkan kolom *Protocol*. Catat versi tiap berkas (`http/1.1`, `h2`, `h3`) dan waktu muat total dengan *cache* dimatikan.
4. Ulangi di bawah *throttling* jaringan (*DevTools*, “Slow 4G”) atau `tc netem` dengan *loss* 2%.

Situs	Versi	DNS	TCP	TLS	TTFB	Total
...	h1.1 / h2 / h3	...	...	...	...	...

Praktikum 2: memasang HTTPS dan menengok ke dalamnya

1. Pasang mkcert, buat CA lokal, terbitkan sertifikat untuk localhost.
2. Jalankan server Go HTTPS dari slide sebelumnya. Uji dengan `curl -http1.1` dan `curl -http2`; catat nilai `r.Proto`.
3. Tangkap lalu lintasnya dengan Wireshark di `lo` port 8443. Temukan ClientHello dan ServerHello. Bisakah Anda membaca isi *response*? Mengapa tidak?
4. Jalankan `openssl s_client -connect localhost:8443` dan baca rantai sertifikatnya. Siapa *issuer*-nya?
5. (Tantangan) Taruh dua instans server di belakang nginx dengan `least_conn`, matikan satu instans, dan amati apa yang terjadi pada *request* berikutnya.

Intinya

Laporan memuat tabel pengukuran, tangkapan layar Wireshark dan *DevTools*, serta penjelasan: **di bagian mana** waktu paling banyak habis, dan mengapa.

- ▶ HTTP adalah protokol *request-response* yang *stateless*; *cookie* dan *caching* melengkapinya.
- ▶ Metode *safe* dan *idempotent* menentukan apa yang aman di-*retry*.
- ▶ HTTP/1.1 terhambat satu *request* per koneksi; HTTP/2 me-*multiplex* di atas satu koneksi TCP tetapi mewarisi *head-of-line blocking* TCP; HTTP/3 memindahkannya ke QUIC.
- ▶ TLS 1.3 memberi kerahasiaan, integritas, dan autentikasi dalam 1 RTT; kepercayaan bersandar pada rantai CA. mTLS menerapkannya ke dua arah antar layanan.
- ▶ *Reverse proxy*, *load balancer* L4/L7, dan CDN adalah lapis sistem terdistribusi yang berdiri di depan aplikasi Anda.

Intinya

Setiap peningkatan protokol menyerang satu sumber *latency*. Tugas insinyur adalah mengukur sumber mana yang dominan sebelum memilih senjatanya.

Tugas untuk pertemuan berikutnya

1. Selesaikan Praktikum 1 dan 2; kumpulkan laporan dengan tabel, tangkapan layar, dan analisis.
2. Baca Kurose & Ross Bab 2.2 (HTTP) dan Bab 8 bagian TLS; baca sekilas daftar isi RFC 9110 untuk melihat bagaimana sebuah protokol dispesifikasikan.
3. Pasang protoc (Protocol Buffers) dan plugin gRPC untuk Go atau Python.
4. Renungkan: bila Anda harus membuat API untuk aplikasi *mobile* dan untuk komunikasi antar layanan internal, apakah keduanya harus memakai gaya yang sama?

Pertemuan depan

Di atas HTTP kita memilih gaya bicara: RPC, Java RMI, REST, gRPC, atau GraphQL. Kita akan membangun layanan yang sama dengan REST dan gRPC, lalu mengukur mana yang lebih cepat.

Terima kasih

Pertanyaan?