

Pertemuan 4 — RPC, Java RMI, REST, gRPC, GraphQL

Memilih cara layanan saling memanggil, dan membuktikannya dengan pengukuran

Hendri Karisma

Semester Ganjil 2026/2027

Program Studi Teknik Informatika, STMIK Tazkia

Alur pertemuan hari ini

1. Dari panggilan lokal ke panggilan jarak jauh: RPC
2. Java RMI dan CORBA: RPC berorientasi objek
3. REST: sumber daya di atas HTTP
4. gRPC: kontrak Protobuf di atas HTTP/2
5. GraphQL: klien menyebut data yang ia butuhkan
6. *Serialization*: JSON vs Protobuf
7. Panggilan yang andal: *timeout*, *retry*, *idempotency*
8. *Versioning* dan evolusi kontrak API
9. Mengukur: *latency* P_{50}/P_{99} dan *throughput*
10. Praktikum: satu layanan, dua protokol

Sasaran pertemuan ini

Anda dapat menjelaskan konsep RPC, *stub*, dan *serialization*; membandingkan Java RMI, REST, gRPC, dan GraphQL; membangun layanan yang sama dengan REST dan gRPC; serta memilih di antara keduanya berdasarkan **angka hasil pengukuran**, bukan selera.

Dari panggilan lokal ke panggilan jarak jauh

Terlihat sama di kode, berbeda di dunia nyata

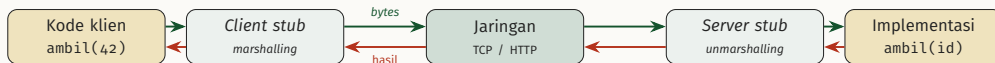
	Panggilan fungsi lokal	Panggilan jarak jauh
Waktu	Nanodetik	Ratusan mikrodetik sampai ratusan milidetik
Kegagalan	Berhasil atau <i>exception</i> yang jelas	Bisa hilang, lambat, atau terjadi tanpa Anda tahu
Parameter	Dilewatkan lewat memori (<i>pointer</i>)	Harus diubah jadi deretan <i>byte</i> lalu dikirim
Dijalankan berapa kali	Tepat sekali	Nol, sekali, atau lebih dari sekali
Versi kode	Selalu sama, satu <i>binary</i>	Klien dan server bisa beda versi

Intinya

Semua gaya komunikasi hari ini adalah upaya menyembunyikan baris-baris kanan tabel ini. Penyembunyian itu nyaman, tetapi tidak pernah sempurna.

RPC: Birrell dan Nelson (1984)

Gagasannya: membuat panggilan ke mesin lain **terlihat** seperti memanggil prosedur biasa. Kuncinya adalah **stub** yang dibangkitkan otomatis.



- ▶ **Marshalling** — mengubah parameter (angka, teks, objek) menjadi deretan *byte* dengan format yang disepakati. **Unmarshalling** adalah kebalikannya.
- ▶ **IDL** (*Interface Definition Language*) — kontrak yang ditulis sekali, lalu dipakai untuk membangkitkan *stub* di berbagai bahasa.
- ▶ Keturunannya: Sun RPC, CORBA, Java RMI, XML-RPC, SOAP, Thrift, gRPC.

Bahaya ilusi “seolah lokal”: semantik pemanggilan

Klien mengirim permintaan dan **tidak mendapat jawaban**. Tiga kemungkinan:

1. Permintaan hilang sebelum sampai
2. Server mati saat memproses
3. Server selesai, tetapi jawabannya hilang

Dari sisi klien, ketiganya **tidak dapat dibedakan**.

Mengulang aman untuk kasus 1, tetapi pada kasus 3 berarti pekerjaannya dikerjakan dua kali.

Tiga semantik

- ▶ **At-most-once** — tidak pernah diulang; bisa tidak terjadi sama sekali
- ▶ **At-least-once** — diulang sampai berhasil; bisa terjadi lebih dari sekali
- ▶ **Exactly-once** — yang benar-benar terwujud adalah *at-least-once* **ditambah** pemrosesan *idempotent* di server

Intinya

Transfer uang yang terkirim dua kali adalah bug nyata, bukan soal teori. Kita kembali ke sini di bagian *idempotency*.

Java RMI dan CORBA

Java RMI (*Remote Method Invocation*)

- ▶ Memanggil *method* pada objek yang hidup di JVM lain
- ▶ Kontraknya *interface* Java yang meng-extend *Remote*
- ▶ Objek dicari lewat *RMI registry* (*port* bawaan 1099)
- ▶ Parameter dikirim dengan *Java serialization*
- ▶ Klien dan server **harus** sama-sama Java

CORBA

- ▶ Standar OMG untuk objek terdistribusi lintas bahasa
- ▶ Kontrak ditulis dalam IDL, lalu dibangkitkan ke C++, Java, dan lainnya
- ▶ Perantara bernama ORB (*Object Request Broker*)
- ▶ Spesifikasinya besar dan rumit; implementasi dari vendor berbeda sering sulit saling bicara

Keduanya populer pada 1990-an sampai awal 2000-an, terutama di perbankan dan telekomunikasi. Gagasan *stub* dan IDL-nya hidup terus di gRPC.

Java RMI dalam tiga potong kode

Kontrak dan implementasi

```
import java.rmi.*;

public interface Katalog extends Remote {
    Produk ambilProduk(long id) throws RemoteException;
}

// dikirim lewat jaringan: wajib Serializable
public record Produk(long id, String nama, long harga)
    implements java.io.Serializable {}

public class KatalogImpl implements Katalog {
    public Produk ambilProduk(long id) {
        return new Produk(id, "Kopi", 25000);
    }
}
```

Server

```
Katalog stub = (Katalog) UnicastRemoteObject
    .exportObject(new KatalogImpl(), 0);
Registry reg = LocateRegistry.createRegistry(1099);
reg.rebind("Katalog", stub);
```

Klien

```
Registry reg =
    LocateRegistry.getRegistry("localhost", 1099);
Katalog k = (Katalog) reg.lookup("Katalog");
Produk p = k.ambilProduk(42); // terlihat lokal!
```

Perhatikan throws RemoteException: Java memaksa pemanggil mengakui bahwa panggilan ini **bisa gagal karena jaringan**. Itu kejujuran yang baik.

Mengapa RMI jarang dipilih untuk sistem baru

- ▶ **Terikat satu bahasa** — layanan Go, Python, atau aplikasi *mobile* tidak bisa ikut bicara.
- ▶ **Java serialization berbahaya** — membaca objek kiriman pihak lain dapat menjalankan kode yang tidak diinginkan (*deserialization vulnerability*). Banyak celah keamanan Java berasal dari sini.
- ▶ **Tidak ramah jaringan modern** — *port* dinamis sulit dilewatkan *firewall* dan *load balancer* HTTP.
- ▶ **Objek jarak jauh menggoda desain “cerewet”** — banyak panggilan kecil yang masing-masing membayar *latency* jaringan.

Intinya

Pelajaran yang terbawa: kontrak harus **netral bahasa**, format kiriman harus **aman dan sederhana**, dan protokolnya sebaiknya **menumpang HTTP**. Ketiganya menjadi ciri REST dan gRPC.

REST: sumber daya di atas HTTP

REST menurut Roy Fielding (2000)

REST (*Representational State Transfer*) adalah **gaya arsitektur**, bukan protokol. Ia dirumuskan dalam disertasi Fielding sebagai penjelasan mengapa web dapat diskalakan.

- ▶ **Resource** — semua hal diberi alamat: */produk/42*
- ▶ **Representation** — *resource* dikirim dalam suatu format, biasanya JSON
- ▶ **Uniform interface** — operasi memakai *verb* HTTP yang sudah baku
- ▶ **Stateless** — tiap permintaan membawa semua informasi yang dibutuhkan; server tidak mengingat sesi
- ▶ **Cacheable** — jawaban menyatakan boleh tidaknya disimpan (Cache-Control, ETag)
- ▶ **Layered system** — *proxy*, CDN, dan *load balancer* dapat disisipkan tanpa klien tahu
- ▶ **HATEOAS** — jawaban memuat tautan ke langkah berikutnya; jarang diterapkan penuh dalam praktik

Intinya

Stateless adalah kunci skala: karena server tidak mengingat apa pun tentang klien, permintaan berikutnya boleh dilayani server mana saja di belakang *load balancer*.

Verb HTTP dan sifatnya

Verb	Makna	Safe	Idempotent	Contoh
GET	Ambil representasi	ya	ya	GET /v1/produk/42
POST	Buat baru / proses	tidak	tidak	POST /v1/pesanan
PUT	Ganti seluruhnya	tidak	ya	PUT /v1/produk/42
PATCH	Ubah sebagian	tidak	tidak dijamin	PATCH /v1/produk/42
DELETE	Hapus	tidak	ya	DELETE /v1/produk/42

- ▶ **Safe:** tidak mengubah apa pun di server.
- ▶ **Idempotent:** dikirim sekali atau sepuluh kali, **akibatnya** sama. DELETE kedua mungkin menjawab 404, tetapi keadaan server tetap sama.
- ▶ Karena itu *proxy* dan klien HTTP boleh mengulang GET, PUT, dan DELETE secara otomatis, tetapi **tidak boleh** mengulang POST begitu saja.

REST dengan Go: layanan katalog

```
type Produk struct {
    ID      int64  `json:"id"`
    Nama    string `json:"nama"`
    Harga   int64  `json:"harga"`
}

var gudang = map[int64]Produk{42: {42, "Kopi", 25000}}

func ambil(w http.ResponseWriter, r *http.Request) {
    id, err := strconv.ParseInt(r.PathValue("id"), 10, 64)
    p, ok := gudang[id]
    if err != nil || !ok {
        http.Error(w, "tidak ditemukan", http.StatusNotFound)
        return
    }
    w.Header().Set("Content-Type", "application/json")
    json.NewEncoder(w).Encode(p)
}

func main() {
    mux := http.NewServeMux()
    mux.HandleFunc("GET /v1/produk/{id}", ambil) // Go 1.22+
    log.Fatal(http.ListenAndServe(":8080", mux))
}
```

Mencoba dari terminal:

```
$ curl -i localhost:8080/v1/produk/42
HTTP/1.1 200 OK
Content-Type: application/json

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

Hal yang membuat REST mudah diadopsi:

- ▶ Bisa diuji dengan curl dan peramban
- ▶ Dibaca manusia
- ▶ Didukung semua bahasa, proxy, dan CDN
- ▶ Kontrak bisa dibakukan dengan OpenAPI

REST yang baik: kebiasaan yang perlu dibangun

- ▶ **Status code jujur** — 200, 201 *Created*, 400 salah masukan, 401/403 soal izin, 404, 409 konflik, 429 terlalu banyak, 5xx kesalahan server
- ▶ **Format galat seragam** — misalnya `application/problem+json` (RFC 9457) berisi `type`, `title`, `detail`
- ▶ **Pagination** — jangan kembalikan sejuta baris; pakai `cursor` atau `limit/offset`
- ▶ **Versioning** — `/v1/...` di URL atau lewat *header*; perubahan yang merusak klien lama wajib naik versi
- ▶ **Kontrak tertulis** — OpenAPI (dulu Swagger) untuk dokumentasi, validasi, dan pembangkitan klien
- ▶ **Caching** — manfaatkan ETag dan Cache-Control untuk data yang jarang berubah

Intinya

Kelemahan REST yang akan dijawab gRPC dan GraphQL: JSON cukup boros, tidak ada kontrak bertipe yang wajib, dan satu layar aplikasi sering butuh banyak panggilan.

gRPC: kontrak Protobuf di atas HTTP/2

- ▶ Dibuka sebagai *open source* oleh Google pada 2015, berakar dari sistem RPC internal mereka.
- ▶ Kontrak ditulis dalam **Protocol Buffers** (.proto); *stub* klien dan server dibangun untuk Go, Java, Python, C++, dan banyak lainnya.
- ▶ Berjalan di atas **HTTP/2**: biner, *multiplexing*, satu koneksi untuk banyak panggilan.
- ▶ Mendukung **deadline**, pembatalan, dan *metadata* sebagai bagian inti.

Empat jenis panggilan

Unary	satu permintaan, satu jawaban
Server streaming	satu permintaan, aliran jawaban
Client streaming	aliran permintaan, satu jawaban
Bidirectional	dua aliran berjalan bersamaan

Kekurangannya: tidak bisa langsung dipanggil peramban tanpa perantara (gRPC-Web), dan isi kiriman tidak terbaca manusia tanpa alat seperti `grpcurl`.

Kontrak katalog.proto

```
syntax = "proto3";
package katalog;
option go_package = "contoh/katalog/gen";

service Katalog {
  rpc AmbilProduk(AmbilProdukRequest) returns (Produk);
  rpc PantauHarga(AmbilProdukRequest)
    returns (stream Produk); // server streaming
}

message AmbilProdukRequest {
  int64 id = 1;
}

message Produk {
  int64 id = 1;
  string nama = 2;
  int64 harga = 3;
}
```

Membangkitkan kode Go:

```
protoc --go_out=. --go_opt=paths=source_relative \
  --go-grpc_out=. \
  --go-grpc_opt=paths=source_relative \
  katalog.proto
```

- ▶ Angka = 1, = 2, = 3 adalah **field number**: yang dikirim lewat jaringan adalah angka ini, bukan nama *field*.
- ▶ Nama *field* boleh diganti; *field number* **tidak boleh** diubah atau dipakai ulang.
- ▶ Kontrak ini menjadi satu-satunya sumber kebenaran untuk klien dan server di bahasa apa pun.

Server gRPC dengan Go

```
type server struct {  
    pb.UnimplementedKatalogServer // menjaga kompatibilitas saat kontrak bertambah  
}  
  
func (s *server) AmbilProduk(ctx context.Context,  
    req *pb.AmbilProdukRequest) (*pb.Produk, error) {  
    p, ok := gudang[req.GetId()]  
    if !ok {  
        return nil, status.Errorf(codes.NotFound, "produk %d tidak ada", req.GetId())  
    }  
    return p, nil  
}  
  
func main() {  
    lis, err := net.Listen("tcp", ":50051")  
    if err != nil { log.Fatal(err) }  
    s := grpc.NewServer()  
    pb.RegisterKatalogServer(s, &server{})  
    log.Fatal(s.Serve(lis))  
}
```

Bandingkan dengan versi REST: tidak ada *parsing* URL, tidak ada *encode* JSON manual. Galat memakai kode baku gRPC (NotFound, DeadlineExceeded, Unavailable, ...), bukan status HTTP.

Klien gRPC: selalu pasang *deadline*

```
conn, err := grpc.NewClient("localhost:50051",
    grpc.WithTransportCredentials(insecure.NewCredentials()))
if err != nil {
    log.Fatal(err)
}
defer conn.Close()
c := pb.NewKatalogClient(conn)

ctx, cancel := context.WithTimeout(
    context.Background(), 300*time.Millisecond)
defer cancel()

p, err := c.AmbilProduk(ctx, &pb.AmbilProdukRequest{Id: 42})
switch status.Code(err) {
case codes.OK:
    fmt.Println(p.GetNama(), p.GetHarga())
case codes.DeadlineExceeded:
    log.Println("terlalu lambat, pakai cadangan")
default:
    log.Println("gagal:", err)
}
```

- ▶ **Deadline** ikut terkirim ke server. Bila server memanggil layanan lain dengan `ctx` yang sama, sisa waktunya ikut berkurang (*deadline propagation*).
- ▶ Tanpa *deadline*, satu layanan lambat membuat seluruh rantai menunggu, dan *thread* serta koneksi menumpuk.
- ▶ `insecure` hanya untuk praktikum. Di produksi pakai TLS atau mTLS.

**GraphQL: klien menyebut data
yang ia butuhkan**

GraphQL: satu *endpoint*, *query* yang fleksibel

- ▶ Dikembangkan Facebook untuk aplikasi *mobile*-nya, lalu dibuka pada 2015.
- ▶ Server menerbitkan **schema** bertipe. Klien mengirim **query** yang menyebut persis *field* yang ia butuhkan.
- ▶ Satu *endpoint* (biasanya POST /graphql) untuk semua kebutuhan.
- ▶ Tiga operasi: **query** (baca), **mutation** (ubah), **subscription** (aliran perubahan).
- ▶ Tiap *field* dihitung oleh fungsi **resolver**.

Masalah REST yang dijawab

- ▶ **Over-fetching** — REST mengirim 30 *field* padahal layar hanya butuh 3
- ▶ **Under-fetching** — satu layar butuh produk, penjual, dan ulasan: tiga kali bolak-balik jaringan
- ▶ Di jaringan seluler, setiap bolak-balik itu mahal

Skema dan query

Skema (server)

```
type Penjual {  
  id: ID!  
  nama: String!  
  kota: String  
}  
  
type Produk {  
  id: ID!  
  nama: String!  
  harga: Int!  
  penjual: Penjual!  
}  
  
type Query {  
  produk(kategori: String): [Produk!]!  
}
```

Query (klien)

```
query {  
  produk(kategori: "kopi") {  
    nama  
    harga  
    penjual { nama }  
  }  
}
```

Jawaban: bentuknya mengikuti query

```
{ "data": { "produk": [  
  { "nama": "Kopi Gayo", "harga": 25000,  
    "penjual": { "nama": "Toko A" } },  
  { "nama": "Kopi Toraja", "harga": 27000,  
    "penjual": { "nama": "Toko B" } } ] } }
```

Jebakan khas GraphQL: masalah N+1

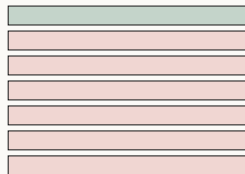
Query tadi tampak satu, tetapi *resolver* yang naif menjalankan:

- 1 *query* basis data untuk daftar produk (N baris)
2. lalu, untuk **tiap** produk, 1 *query* untuk penjualnya

Total $1 + N$ *query*. Untuk 100 produk: 101 kali ke basis data.

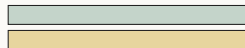
Solusi: batching — pola *DataLoader* mengumpulkan semua id penjual dalam satu putaran, lalu mengambilnya sekaligus: `SELECT ... WHERE id IN (...)`. Total menjadi 2 *query*.

Naif: $1 + N$



... dan seterusnya

DataLoader: 2



Intinya

Kebebasan klien menyusun *query* berarti server harus menjaga diri: batasi kedalaman dan kerumitan *query*, dan ukur biayanya. Fleksibilitas selalu ada harganya.

Serialization: JSON vs Protobuf

Data yang sama, ukuran berbeda

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

JSON — 37 byte

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

Nama *field* ikut dikirim di setiap pesan. Angka ditulis sebagai teks. Mudah dibaca manusia, tetapi harus di-*parse* karakter demi karakter.

Protobuf — 12 byte

```
08 2A ← field 1, varint 42  
12 04 4B 6F 70 69 ← field 2, panjang 4, "Kopi"  
18 A8 C3 01 ← field 3, varint 25000
```

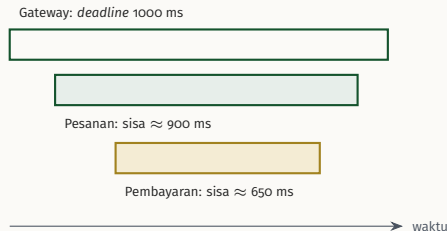
Yang dikirim hanya nomor *field* dan nilainya. Bilangan kecil memakai lebih sedikit *byte* (*varint*).

	JSON	Protobuf
Terbaca manusia	ya	tidak, perlu alat
Skema	opsional (JSON Schema)	wajib (.proto)
Ukuran dan biaya <i>parse</i>	lebih besar dan lambat	lebih kecil dan cepat
Evolusi	bebas, tetapi tanpa penjaga	aturan jelas lewat <i>field number</i>

**Panggilan yang andal: *timeout*,
retry, *idempotency***

Timeout: batas sabar yang harus selalu ada

- ▶ Tanpa *timeout*, klien bisa menunggu selamanya pada server yang macet.
- ▶ Terlalu pendek: permintaan sehat ikut dibatalkan. Terlalu panjang: sumber daya tertahan.
- ▶ Pedoman awal: sedikit di atas P_{99} *latency* normal layanan itu, lalu disetel berdasarkan pengukuran.
- ▶ **Deadline propagation:** layanan di hilir tidak boleh diberi waktu lebih lama daripada sisa waktu pemanggilnya.



Intinya

Setiap panggilan jaringan wajib punya batas waktu. Pertanyaannya bukan “perlukah”, melainkan “berapa”, dan jawabannya datang dari data pengukuran.

Retry dengan *exponential backoff* dan *jitter*

```
func denganRetry(ctx context.Context,
    f func(context.Context) error) error {
    base, maks := 100*time.Millisecond, 2*time.Second
    var err error
    for coba := 0; coba < 5; coba++ {
        if err = f(ctx); err == nil || !bisaDiulang(err) {
            return err
        }
        tunda := min(maks, base<<coba) // 100,200,400,...
        tunda = time.Duration(rand.Int64N(int64(tunda)))
        select {
            // full jitter
        case <-time.After(tunda):
        case <-ctx.Done():
            return ctx.Err()
        }
    }
    return err
}
```

- ▶ **Exponential backoff** — jeda digandakan tiap gagal, supaya server yang kewalahan diberi napas.
- ▶ **Jitter** — jeda diacak. Tanpa acak, ribuan klien yang gagal bersamaan akan mencoba ulang **bersamaan** pula, menghantam server serentak.
- ▶ Ulangi hanya galat sementara (Unavailable, 503, 429), dan hanya untuk operasi *idempotent*.
- ▶ Batasi jumlah percobaan. *Retry* di setiap lapis bisa berlipat: 3 lapis $\times$ 3 percobaan = 27 permintaan.

Idempotency key: membuat POST aman diulang



- ▶ Klien mengirim kunci yang **sama** pada setiap percobaan ulang untuk operasi yang sama. Penyedia layanan pembayaran banyak memakai pola ini.
- ▶ Penyimpanan kunci dan hasil harus atomik dengan prosesnya, misalnya dalam satu transaksi basis data dengan *unique constraint* pada kunci.
- ▶ Kunci diberi masa berlaku (misalnya 24 jam) supaya tabelnya tidak tumbuh tanpa batas.

Intinya

At-least-once + idempotent = efek exactly-once. Inilah resep praktis yang dipakai hampir semua sistem pembayaran.

Versioning dan evolusi kontrak

Klien dan server tidak pernah diperbarui pada detik yang sama. Kontrak harus bisa berubah tanpa mematikan klien lama.

	Perubahan aman	Perubahan yang merusak
REST/JSON	Menambah <i>field</i> opsional; menambah <i>endpoint</i> baru	Menghapus atau mengganti nama <i>field</i> ; mengubah tipe; naikan ke /v2
gRPC/Protobuf	Menambah <i>field</i> dengan nomor baru; klien lama mengabaikannya	Mengubah atau memakai ulang <i>field number</i> ; tandai yang dihapus dengan <i>reserved</i>
GraphQL	Menambah <i>field</i> dan tipe baru	Menghapus <i>field</i> ; jalur biasanya: tandai <i>@deprecated</i> , pantau pemakaian, baru hapus

Intinya

Prinsip *robustness*: ketat pada apa yang Anda kirim, longgar pada apa yang Anda terima. Klien yang menolak *field* tak dikenal akan patah setiap kali server berkembang.

Membandingkan semuanya

	Java RMI	REST	gRPC	GraphQL
Kontrak	<i>Interface Java</i>	Opsional (OpenAPI)	Wajib (.proto)	Wajib (<i>schema</i>)
Format	<i>Java serialization</i>	Umumnya JSON	Protobuf biner	JSON
Transport	JRMP di atas TCP	HTTP/1.1, /2, /3	HTTP/2	HTTP, WebSocket untuk <i>subscription</i>
Lintas bahasa	Tidak	Ya	Ya, lewat <i>codegen</i>	Ya
Peramban	Tidak	Ya	Lewat gRPC-Web	Ya
<i>Streaming</i>	Tidak	Terbatas (SSE)	Ya, empat jenis	<i>Subscription</i>
<i>Caching</i> HTTP	Tidak	Sangat baik	Sulit	Sulit (satu <i>endpoint</i> POST)
Tempat terbaik	Sistem Java lama	API publik, integrasi	Antar <i>microservices</i>	<i>Frontend</i> dengan kebutuhan data beragam

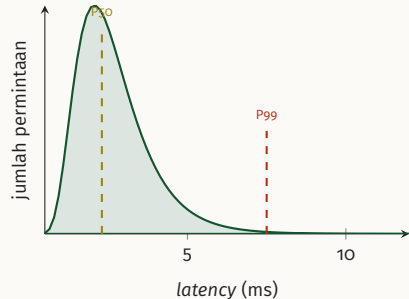
Pola umum di perusahaan: **REST atau GraphQL di tepi** (menghadap klien), **gRPC di dalam** (antar layanan), **messaging** untuk yang tidak perlu dijawab segera. *Messaging* kita bahas pada pertemuan depan.

**Mengukur: *latency* dan
*throughput***

Membaca *latency*: mengapa rata-rata menipu

- ▶ **P50** (median) — separuh permintaan lebih cepat dari angka ini. Wajah “biasa” layanan.
- ▶ **P99** — 99% permintaan lebih cepat; 1% sisanya lebih lambat. Inilah *tail latency*.
- ▶ **Throughput** — berapa permintaan selesai per detik (RPS).

Mengapa P99 penting: bila satu halaman memanggil 100 layanan, peluang semuanya lebih cepat dari P99 masing-masing hanya $0,99^{100} \approx 37\%$. Ekor yang “langka” menjadi pengalaman sebagian besar pengguna.



Ilustrasi bentuk sebaran: ekornya panjang ke kanan.

Alat ukur: ghz untuk gRPC, k6 untuk REST

gRPC dengan ghz

```
ghz --insecure \  
  --proto katalog.proto \  
  --call katalog.Katalog.AmbilProduk \  
  -d '{"id": 42}' \  
  -c 50 -n 20000 \  
  localhost:50051
```

-c = jumlah pemanggil serentak, -n = total permintaan.
Laporannya memuat sebaran *latency* (termasuk persentil) dan RPS.

REST dengan k6 (uji.js)

```
import http from 'k6/http';  
import { check } from 'k6';  
  
export const options = {  
  vus: 50, duration: '30s',  
  summaryTrendStats: ['avg', 'p(50)', 'p(99)', 'max'],  
};  
  
export default function () {  
  const r = http.get('http://localhost:8080/v1/produk/42');  
  check(r, { 'status 200': (x) => x.status === 200 });  
}
```

Jalankan: k6 run uji.js

Pengukuran yang jujur

1. **Samakan kondisi** — logika layanan, data, mesin, dan jumlah pemanggil serentak harus sama untuk REST dan gRPC. Yang dibandingkan hanya protokolnya.
2. **Pemanasan** — buang beberapa detik pertama (koneksi baru, JIT, *cache* dingin).
3. **Ulangi** — jalankan minimal tiga kali; laporkan median dan sebarannya.
4. **Pisahkan pembangkit beban** — bila mungkin, alat ukur jangan berebut CPU dengan server yang diukur.
5. **Laporkan persentil**, bukan hanya rata-rata, beserta RPS dan tingkat galat.
6. **Waspada coordinated omission** — alat yang menunggu jawaban sebelum mengirim permintaan berikutnya cenderung menyembunyikan ekor *latency* saat server tersendat.

Intinya

Kesimpulan “gRPC lebih cepat” hanya sah bila Anda menunjukkan angkanya, kondisinya, dan seberapa besar selisihnya. Pada *payload* kecil di jaringan lokal, selisihnya bisa tipis.

Praktikum

Praktikum: satu layanan katalog, dua protokol

1. Siapkan proyek Go: `go mod init contoh/katalog`; pasang `protoc`, `protoc-gen-go`, dan `protoc-gen-go-grpc`.
2. Tulis `katalog.proto`, bangkitkan kodenya, lalu buat server gRPC di *port* 50051.
3. Buat server REST di *port* 8080 yang memakai **data dan logika yang sama** (GET `/v1/produk/{id}`).
4. Uji fungsional: `curl` untuk REST, `grpcurl -plaintext` untuk gRPC.
5. Ukur: `ghz` dan `k6` dengan 10, 50, dan 200 pemanggil serentak, tiap konfigurasi tiga kali.
6. Isi tabel: P50, P99, RPS, ukuran *payload*, dan tingkat galat.
7. Bonus: jalankan contoh Java RMI dan catat apa yang terjadi bila server dimatikan saat klien memanggil.

Simpan kode ini: layanan katalog akan menjadi salah satu bagian tugas besar UTS.

Tugas untuk pertemuan berikutnya

1. Kumpulkan laporan praktikum: kode (repositori), tabel hasil pengukuran, grafik P50/P99 terhadap jumlah pemanggil serentak, dan analisis satu halaman.
2. Dalam analisis, jawab: pada kondisi apa gRPC unggul jelas, dan pada kondisi apa selisihnya tidak berarti? Mengapa?
3. Rancang *endpoint* POST `/v1/pesanan` yang aman diulang: tuliskan *header idempotency key*, skema tabel penyimpanannya, dan perilakunya bila kunci yang sama datang dua kali.
4. Baca van Steen & Tanenbaum Bab 2 (arsitektur) dan dokumentasi singkat RabbitMQ “Hello World”.

Pertemuan depan

Dari satu layanan ke banyak layanan: 1-tier sampai *microservices*, *API gateway*, *message queue*, Kafka, *saga*, *circuit breaker*, dan *observability*.

Terima kasih

Pertanyaan?