Pertemuan 4 — Komunikasi Antar Layanan Modern

gRPC, REST, GraphQL, *WebSocket*: memilih cara bicara dan membuktikannya dengan pengukuran

Hendri Karisma

Semester Ganjil 2026/2027

Program Studi Teknik Informatika, STMIK Tazkia

Alur pertemuan hari ini

1. Akar: RPC, *stub*, dan ilusi panggilan lokal
2. Sejarah singkat: Sun RPC, CORBA, Java RMI, SOAP
3. REST modern: OpenAPI, ETag, *problem details*
4. gRPC mendalam: empat mode, *deadline*, *interceptor*
5. GraphQL: *DataLoader*, *federation*
6. *Real-time*: SSE, *WebSocket*, *WebTransport*
7. *API gateway*, *service mesh*, dan *tracing*
8. *Timeout*, *retry*, *idempotency*, *versioning*
9. Memilih: pakai yang mana kapan
10. Mengukur P_{50} / P_{99}; praktikum

Sasaran pertemuan ini

Membangun layanan gRPC dan REST yang benar (*deadline*, *interceptor*, kontrak yang bisa berevolusi) dan memilih cara komunikasi berdasarkan **angka pengukuran**.

Akar: RPC dan sejarahnya

Terlihat sama di kode, berbeda di dunia nyata

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

Ringkasan

Semua cara komunikasi hari ini adalah upaya menyembunyikan kolom 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. gRPC hari ini masih memakai gagasan yang persis sama.



- ***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. Di gRPC, IDL-nya adalah *file* `.proto`.

Bahaya ilusi “seolah lokal”: semantik pemanggilan

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

1. *Request* hilang sebelum sampai
2. Server mati saat memproses
3. Server selesai, tetapi *response*-nya hilang

Dari sisi klien, ketiganya **tidak bisa dibedakan**. Mengulang aman untuk kasus 1, tetapi pada kasus 3 berarti pekerjaannya dikerjakan dua kali.

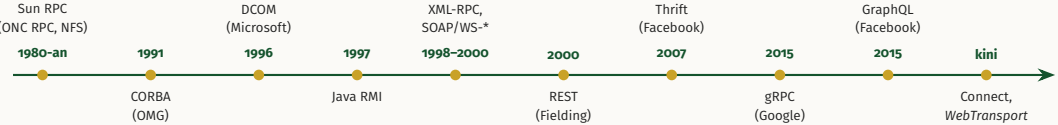
Tiga semantik

- ***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*

Ringkasan

Transfer uang yang terkirim dua kali adalah *bug* yang sering terjadi. Kita kembali ke sini di bagian *idempotency*.

Sejarah singkat: dari Sun RPC ke gRPC



RPC berorientasi objek

CORBA, DCOM, dan Java RMI memanggil *method* pada objek di mesin lain. Ambisius, tetapi berat, sering terikat satu bahasa atau vendor, dan sulit melewati *firewall*.

Era XML dan web

SOAP membungkus panggilan dalam XML di atas HTTP, lengkap dengan WSDL dan *stack* WS-*. Lintas bahasa, tetapi *verbose*. REST memilih jalan sebaliknya: sederhana, memakai HTTP apa adanya.

Era biner dan kontrak

Thrift dan gRPC menghidupkan lagi *stub* dan IDL, kini dengan format biner ringkas, netral bahasa, di atas HTTP/2.

Yang lama masih hidup, tetapi jarang dipilih untuk sistem baru

Teknologi	Masih ditemui di	Mengapa ditinggalkan
CORBA	Sistem warisan telekomunikasi, sebagian *core banking* lama	Spesifikasi besar, implementasi antar vendor sulit saling bicara
Java RMI	Aplikasi Java *enterprise* lama (EJB)	Terikat Java; *Java serialization* rawan *deserialization vulnerability*; *port* dinamis
SOAP/WS-*	Integrasi bank, asuransi, dan instansi pemerintah	*Verbose* (XML), alat berat, tetapi kontraknya (WSDL) masih diwajibkan sebagian mitra
Thrift	Sebagian sistem besar yang lahir sebelum gRPC	Ekosistem lebih kecil daripada gRPC

Ringkasan

Pelajaran yang terbawa ke era modern: kontrak harus **netral bahasa**, format kiriman harus **aman dan sederhana**, dan protokolnya sebaiknya **menumpang HTTP**. Bila harus berintegrasi dengan SOAP milik mitra, bungkus di balik satu *adapter* atau *gateway*, jangan biarkan menyebar ke seluruh sistem.

REST modern

REST menurut Roy Fielding (2000)

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

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

Ringkasan

Stateless adalah kunci skala: karena *server* tidak mengingat apa pun tentang klien, *request* 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 *state 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

```go
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}
```

Mengapa REST tetap populer:

- Bisa diuji dengan `curl` dan *browser*
- Terbaca manusia
- Didukung semua bahasa, *proxy*, dan CDN
- Kontrak bisa dibakukan dengan OpenAPI

OpenAPI: kontrak dulu, kode belakangan (*contract-first*)

```yaml
openapi: 3.1.0
info: { title: Katalog, version: 1.0.0 }
paths:
  /v1/produk/{id}:
    get:
      operationId: ambilProduk
      parameters:
        - name: id
          in: path
          required: true
          schema: { type: integer, format: int64 }
      responses:
        "200":
          description: produk ditemukan
          content:
            application/json:
              schema: { $ref: "#/components/schemas/Produk" }
        "404":
          description: tidak ditemukan
          content:
            application/problem+json:
              schema: { $ref: "#/components/schemas/Problem" }
```

- ***Contract-first***: tim menyepakati *file* ini dulu, baru menulis kode. *Frontend* dan *backend* bisa bekerja paralel.
- ***Codegen***: *server stub* dan klien dibangkitkan dari kontrak, misalnya `oapi-codegen` (Go) atau `openapi-generator` (banyak bahasa).
- *File* yang sama menghasilkan dokumentasi interaktif, validasi *request*, dan *mock server* untuk pengujian.
- Simpan di Git dan periksa di CI: perubahan yang merusak klien lama ketahuan sebelum *deploy*.

REST yang baik: *pagination*, *problem details*, ETag, *rate limiting*

Error seragam: RFC 9457 (*Problem Details*)

```
HTTP/1.1 404 Not Found
Content-Type: application/problem+json

{ "type": "https://contoh.id/err/produk-tidak-ada",
  "title": "Produk tidak ditemukan",
  "status": 404,
  "detail": "Produk 99 tidak ada di katalog",
  "instance": "/v1/produk/99" }
```

Cursor pagination

```
GET /v1/transaksi?limit=50&after=eyJpZCI6MTIzNH0
```

Offset besar lambat dan bergeser saat ada data baru; *cursor* menunjuk posisi terakhir yang sudah dibaca.

ETag dan *conditional request*

```
GET /v1/produk/42            -> 200, ETag: "v7"
GET  + If-None-Match: "v7"   -> 304 Not Modified
PUT  + If-Match: "v7"        -> 412 bila sudah
                                diubah orang lain
```

- `If-None-Match` menghemat *bandwidth*: data tidak dikirim ulang bila belum berubah.
- `If-Match` mencegah *lost update* (*optimistic concurrency*): dua admin mengubah produk yang sama, yang kedua ditolak.
- ***Rate limiting***: jawab 429 *Too Many Requests* beserta `Retry-After`; algoritma umum *token bucket*.
- *Status code* jujur: 201, 400, 401/403, 404, 409, 429, 5xx.

gRPC mendalam

gRPC: RPC modern 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* dibangkitkan untuk Go, Java, Python, C++, Rust, dan lainnya.
- Berjalan di atas **HTTP/2**: biner, *multiplexing*, satu koneksi untuk banyak panggilan.
- *Deadline*, pembatalan, *metadata*, *status code*, dan *streaming* adalah bagian inti, bukan tambahan.

Empat mode panggilan

Unary	satu *request*, satu *response*
Server streaming	satu *request*, aliran *response*
Client streaming	aliran *request*, satu *response*
Bidirectional	dua aliran berjalan bersamaan

Kekurangannya: *browser* tidak bisa memanggil gRPC asli secara langsung (perlu gRPC-Web atau Connect), dan isinya tidak terbaca manusia tanpa alat seperti `grpcurl`.

Kontrak `katalog.proto`: empat mode sekaligus

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

service Katalog {
  // unary: satu tanya, satu jawab
  rpc AmbilProduk(AmbilProdukRequest) returns (Produk);
  // server streaming: harga dikirim terus
  rpc PantauHarga(AmbilProdukRequest) returns (stream Produk);
  // client streaming: unggah banyak, satu ringkasan
  rpc ImporProduk(stream Produk) returns (RingkasanImpor);
  // bidirectional: tawar-menawar dua arah
  rpc Tawar(stream Tawaran) returns (stream Tawaran);
}

message AmbilProdukRequest { int64 id = 1; }
message Produk {
  int64  id    = 1;
  string nama  = 2;
  int64  harga = 3;
}
message RingkasanImpor { int32 jumlah = 1; }
message Tawaran { int64 id = 1; int64 harga = 2; }
```

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*.
- Kontrak ini menjadi *single source of truth* untuk klien dan *server* di bahasa apa pun.
- Alternatif `protoc`: `buf generate` dengan *configuration* `buf.gen.yaml`.

Server gRPC dengan Go: *unary* dan *server streaming*

```go
type server struct {
    pb.UnimplementedKatalogServer // aman saat kontrak bertambah
}

func (s *server) AmbilProduk(ctx context.Context,
    req *pb.AmbilProdukRequest) (*pb.Produk, error) {
    p, ok := gudang[req.GetId()] // map[int64]*pb.Produk
    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))
}
```

```go
func (s *server) PantauHarga(req *pb.AmbilProdukRequest,
    stream pb.Katalog_PantauHargaServer) error {
    t := time.NewTicker(time.Second)
    defer t.Stop()
    for {
        select {
        case <-stream.Context().Done():
            // klien batal atau deadline habis
            return stream.Context().Err()
        case <-t.C:
            if err := stream.Send(gudang[req.GetId()]);
                err != nil {
                return err
            }
        }
    }
}
```

`stream.Context().Done()` membuat *server* **berhenti bekerja** begitu klien pergi. Tanpa ini, *server* terus mengirim ke klien yang sudah tidak ada. Catatan: `gudang` di sini berisi `*pb.Produk`, berbeda dari *struct* `Produk` versi REST, sehingga data dari peta REST perlu dikonversi dulu.

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* (*header* `grpc-timeout`). Bila *server* memanggil layanan lain dengan `ctx` yang sama, sisa waktunya ikut berkurang: *deadline propagation*.
- ***Cancellation propagation***: bila klien membatalkan, pembatalan itu mengalir ke seluruh rantai panggilan di bawahnya.
- Tanpa *deadline*, satu layanan lambat membuat seluruh rantai menunggu, dan *goroutine* serta koneksi menumpuk.
- `insecure` hanya untuk praktikum. Di produksi pakai TLS atau mTLS.

Metadata dan *interceptor*: *middleware* versi gRPC

```go
// server: satu interceptor untuk auth dan log semua method
func authLog(ctx context.Context, req any,
    info *grpc.UnaryServerInfo,
    h grpc.UnaryHandler) (any, error) {
    md, _ := metadata.FromIncomingContext(ctx)
    if len(md.Get("authorization")) == 0 {
        return nil, status.Error(codes.Unauthenticated,
            "token kosong")
    }
    mulai := time.Now()
    resp, err := h(ctx, req) // panggil handler asli
    log.Printf("%s %v %s", info.FullMethod,
        status.Code(err), time.Since(mulai))
    return resp, err
}
s := grpc.NewServer(grpc.ChainUnaryInterceptor(authLog))

// klien: menitipkan metadata
ctx = metadata.AppendToOutgoingContext(ctx,
    "authorization", "Bearer eyJhbGci...")
```

- ***Metadata***: pasangan kunci-nilai di luar isi *message*, setara *header* HTTP. Dipakai untuk *token*, ID *request*, dan konteks *tracing*.
- ***Interceptor***: fungsi yang membungkus setiap panggilan, di klien maupun *server*. Tempat yang tepat untuk *auth*, *logging*, *metrics*, dan *tracing*, supaya tidak ditulis ulang di tiap *method*.
- Versi *streaming* memakai `StreamServerInterceptor`.
- OpenTelemetry menyediakan *instrumentation* siap pakai untuk gRPC.

Status code gRPC: *error* yang bisa ditindaklanjuti

Kode	Kira-kira setara HTTP	Kapan dipakai	Aman diulang?
`INVALID_ARGUMENT`	400	*Input* salah dari klien	tidak
`UNAUTHENTICATED`	401	*Token* tidak ada atau tidak sah	tidak
`PERMISSION_DENIED`	403	Sah, tetapi tidak berhak	tidak
`NOT_FOUND`	404	Data tidak ada	tidak
`ALREADY_EXISTS`	409	Data sudah ada	tidak
`FAILED_PRECONDITION`	400/412	*State* sistem belum memungkinkan	tidak
`RESOURCE_EXHAUSTED`	429	Kuota habis, *rate limit*	ya, dengan jeda
`DEADLINE_EXCEEDED`	504	Waktu habis	tergantung *idempotency*
`UNAVAILABLE`	503	Server sementara tidak bisa melayani	ya
`INTERNAL`	500	*Bug* atau *error* tak terduga di *server*	umumnya tidak

Pilih kode yang tepat, bukan selalu `INTERNAL`: klien, *proxy*, dan *retry policy* membuat keputusan berdasarkan kode ini.

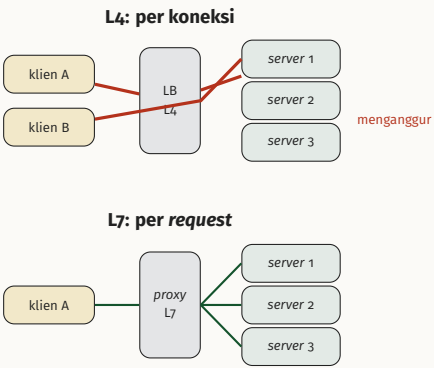
Retry policy dan *hedging* lewat *service config*

```
{ "methodConfig": [{
    "name": [{ "service": "katalog.Katalog" }],
    "timeout": "0.3s",
    "retryPolicy": {
      "maxAttempts": 4,
      "initialBackoff": "0.1s",
      "maxBackoff": "1s",
      "backoffMultiplier": 2,
      "retryableStatusCodes": ["UNAVAILABLE"]
    }
}]}
```

```
conn, _ := grpc.NewClient(alamat,
    grpc.WithDefaultServiceConfig(konfigJSON), ...)
```

- ***Retry policy***: *library* gRPC mengulang otomatis untuk kode tertentu, dengan *exponential backoff*. Kode aplikasi tidak perlu *loop* sendiri.
- ***Hedging***: kirim salinan *request* ke *server* lain bila jawaban pertama belum datang setelah jeda tertentu; ambil yang tercepat. Ampuh menekan *tail latency*, tetapi hanya untuk operasi *idempotent* dan menambah beban.
- Satu *method* memakai *retry* **atau** *hedging*, tidak keduanya. Dukungan *hedging* berbeda antar bahasa; cek dokumentasi *library* yang dipakai.

Load balancing gRPC: mengapa L4 saja tidak cukup



- gRPC memakai **satu koneksi HTTP/2 berumur panjang** untuk ribuan panggilan. *Load balancer* L4 hanya membagi **koneksi**, jadi semua panggilan dari satu klien menumpuk di satu *server*.
- ***Proxy* L7** (Envoy, Linkerd, NGINX, APISIX) membuka HTTP/2 dan membagi per *request*.
- ***Client-side load balancing***: klien tahu daftar *server* (DNS, xDS) dan membagi sendiri, misalnya kebijakan `round_robin`.
- Penambal di sisi *server*: `MaxConnectionAge` memaksa koneksi diperbarui berkala supaya klien tersebar ulang.
- Di Kubernetes, *Service* biasa bekerja di L4; gRPC biasanya memakai *headless service* + *client-side* LB atau *service mesh*.

Health checking dan *reflection*

Health checking protocol (`grpc.health.v1`)

```
hs := health.NewServer()
healthpb.RegisterHealthServer(s, hs)
hs.SetServingStatus("katalog.Katalog",
    healthpb.HealthCheckResponse_SERVING)
```

```
# Kubernetes: probe gRPC bawaan
readinessProbe:
  grpc: { port: 50051, service: katalog.Katalog }
```

Layanan bisa melaporkan `NOT_SERVING` saat sedang *warm-up* atau kehilangan *database*, sehingga *load balancer* berhenti mengirim panggilan.

Server reflection

```
reflection.Register(s)
```

```
$ grpcurl -plaintext localhost:50051 list
katalog.Katalog
grpc.health.v1.Health
$ grpcurl -plaintext -d '{"id": 42}' \
    localhost:50051 katalog.Katalog/AmbilProduk
```

Reflection membuat *server* "menjelaskan dirinya", sehingga `grpcurl` dan alat GUI bisa memanggilnya tanpa *file* `.proto`. Aktifkan di *development*; pertimbangkan mematikannya di layanan publik.

gRPC di *browser*: gRPC-Web dan Connect

Masalahnya

gRPC asli butuh kendali penuh atas *frame* HTTP/2 dan *trailers*. API `fetch` di *browser* tidak memberinya, jadi *browser* tidak bisa memanggil gRPC secara langsung.

gRPC-Web

Varian protokol yang cocok untuk *browser*. Biasanya butuh *proxy* penerjemah (misalnya Envoy) di depan *server* gRPC. *Client streaming* dan *bidirectional* tidak didukung.

Connect (Buf)

Satu *server* melayani tiga protokol sekaligus: gRPC, gRPC-Web, dan protokol Connect sendiri. Protokol Connect berjalan di HTTP/1.1 atau HTTP/2 dan bisa memakai JSON, sehingga bisa diuji dengan `curl` biasa:

```
curl -H 'Content-Type: application/json' -d '{"id":42}' \
  https://api.contoh.id/katalog.Katalog/AmbilProduk
```

Alternatif lain: `grpc-gateway` membangkitkan REST/JSON dari anotasi di `.proto`, sehingga satu kontrak melayani klien gRPC dan klien REST.

Schema evolution Protobuf tanpa merusak klien

```
message Produk {
  int64  id      = 1;
  string nama    = 2;
  int64  harga   = 3;
  // field 4 (diskon) dihapus di v2:
  reserved 4;
  reserved "diskon";
  string kategori = 5;   // baru, aman
}
```

```
$ buf lint
$ buf breaking --against '.git#branch=main'
```

- **Aman**: menambah *field* dengan nomor baru (klien lama mengabaikannya), menambah *method* atau *service* baru.
- **Merusak**: mengubah tipe atau nomor *field*, memakai ulang nomor yang pernah dipakai, mengganti nama *method*.
- **`reserved`**: mengunci nomor dan nama *field* yang sudah dihapus supaya tidak dipakai ulang tanpa sengaja.
- **`buf lint`**: memeriksa gaya penamaan dan struktur. **`buf breaking`**: membandingkan dengan versi di *branch* utama dan menggagalkan CI bila ada perubahan yang merusak.

GraphQL

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

Schema dan *query*

***Schema* (*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 }
  }
}
```

Response: bentuknya mengikuti *query*

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

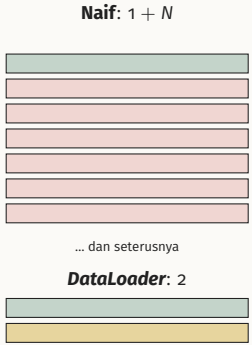
Jebakan khas GraphQL: masalah N+1 dan *DataLoader*

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

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

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

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



Ringkasan

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

GraphQL di organisasi besar: *federation* dan *persisted queries*

Federation

Tiap tim memiliki ***subgraph*** sendiri (produk, penjual, ulasan). Sebuah *router* menggabungkannya menjadi satu ***supergraph***, sehingga klien tetap melihat satu *endpoint*.

```
# subgraph ulasan menambah field ke Produk
# milik tim lain
type Produk @key(fields: "id") {
  id: ID!
  ulasan: [Ulasan!]!
}
```

Contoh implementasi: Apollo Federation dan *router* sejenisnya.

Persisted queries

Klien hanya mengirim *hash* dari teks *query*. Server hanya mau menjalankan *query* yang sudah terdaftar.

- **Keamanan**: penyerang tidak bisa mengirim *query* raksasa sembarangan
- **Ukuran**: *request* jauh lebih kecil
- ***Caching***: *query* terdaftar bisa dikirim lewat `GET`, sehingga CDN bisa menyimpannya

Real-time ke klien

Mengirim data ke klien tanpa ditanya

Cara	Cara kerja	Arah	Cocok untuk
Polling	Klien bertanya tiap *N* detik	klien → *server*	Prototipe; data jarang berubah
Long polling	Server menahan *request* sampai ada data	seolah dua arah	Jaringan yang memblokir cara lain
SSE	*Server-Sent Events*: satu *response* HTTP panjang, `text/event-stream`, *reconnect* otomatis	*server* → klien	Notifikasi, *feed* harga, progres
WebSocket	*Upgrade* dari HTTP menjadi kanal *full-duplex* di atas TCP	dua arah	Chat, *game*, kolaborasi
WebTransport	Di atas HTTP/3 (QUIC): banyak *stream* dan *datagram*	dua arah	Masih berkembang; dukungan *browser* belum merata
gRPC *streaming*	*Stream* HTTP/2 antar layanan	satu atau dua arah	Antar *backend*

Pertanyaan pertama: apakah klien juga perlu mengirim terus-menerus? Bila tidak, SSE lebih sederhana daripada *WebSocket*: tetap HTTP biasa, lewat *proxy* tanpa *configuration* khusus, dan *reconnect* sudah ditangani *browser*.

SSE dan *WebSocket* dalam praktik

Server SSE dengan Go

```go
func harga(w http.ResponseWriter, r *http.Request) {
    w.Header().Set("Content-Type", "text/event-stream")
    w.Header().Set("Cache-Control", "no-cache")
    f := w.(http.Flusher)
    for {
        select {
        case <-r.Context().Done(): // klien pergi
            return
        case h := <-perubahanHarga:
            fmt.Fprintf(w, "data: %d\n\n", h)
            f.Flush()
        }
    }
}
```

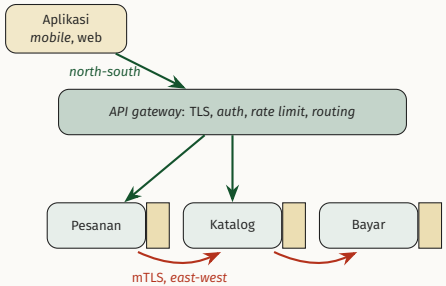
```javascript
const es = new EventSource('/v1/harga/42');
es.onmessage = (e) => tampilkan(e.data);
```

Hal yang sering terlupa

- ***Stateful connection***: koneksi *WebSocket* menempel pada satu *server*. *Scale-out* butuh *pub-sub* di belakang (Redis, NATS, Kafka) agar *message* sampai ke *server* tempat klien terhubung.
- ***Heartbeat***: kirim *ping* berkala; *proxy* dan *load balancer* memutus koneksi yang diam terlalu lama.
- ***Backpressure***: klien lambat jangan sampai membuat *queue* di *server* tumbuh tanpa batas.
- *Auth* dilakukan saat koneksi dibuka, lalu *token* kedaluwarsa tetap harus ditangani.

API gateway, *service mesh*, dan *tracing*

North-south dan *east-west*



kotak emas kecil = *sidecar proxy service mesh*

API gateway (*north-south*)

Pintu masuk dari luar: terminasi TLS, *auth*, *rate limiting*, *routing*, penerjemahan protokol (misalnya REST ke gRPC). Contoh: Kong, APISIX, Envoy Gateway, layanan *cloud*.

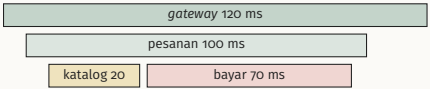
Service mesh (*east-west*)

Lalu lintas antar layanan di dalam: mTLS otomatis, *retry* dan *timeout* sebagai kebijakan, *load balancing* L7, dan *telemetry* tanpa mengubah kode. Contoh: Istio (*sidecar* atau mode *ambient* tanpa *sidecar*), Linkerd (*sidecar*).

Hati-hati: *retry* di kode, di *mesh*, dan di *gateway* bisa berlipat. Tentukan **satu** *layer* yang bertanggung jawab.

Distributed tracing: mengikuti satu *request* melintasi layanan

```
traceparent:
 00-4bf92f3577b34da6a3ce929d0e0e4736-00f067aa0ba902b7-01
 ^  ^ trace-id (32 hex)              ^ span-id (16)   ^
 versi                                               flags
```



Satu *trace* = pohon *span*. Langsung terlihat layanan mana yang memakan waktu (ilustrasi).

- **W3C Trace Context**: *header* `traceparent` membawa *trace-id* yang sama dari layanan ke layanan. Di gRPC, ia dibawa sebagai *metadata*.
- **OpenTelemetry**: standar terbuka untuk *traces*, *metrics*, dan *logs*, dengan *instrumentation* siap pakai untuk HTTP dan gRPC. Data dikirim ke Jaeger, Grafana Tempo, atau layanan sejenis.
- Cantumkan *trace-id* di setiap baris *log*, supaya *log* dari banyak layanan bisa dirangkai menjadi satu cerita.

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 *message*. 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
```

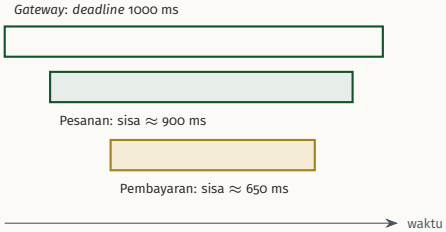
Yang dikirim hanya nomor *field* dan nilainya. Bilangan kecil memakai lebih sedikit *byte* (*varint*).

	JSON	**Protobuf**
Terbaca manusia	ya	tidak, perlu alat
Schema	opsional (JSON Schema)	wajib (`.proto`)
Ukuran dan *parse cost*	lebih besar dan lambat	lebih kecil dan cepat
Evolusi	bebas, tetapi tanpa penjaga	aturan jelas lewat *field number*

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: *request* yang sehat ikut dibatalkan. Terlalu panjang: *resource* tertahan.
- Pedoman awal: sedikit di atas P_{99} *latency* normal layanan itu, lalu disetel berdasarkan pengukuran.
- ***Deadline propagation***: layanan *downstream* tidak boleh diberi waktu lebih lama daripada sisa waktu pemanggilnya.



Ringkasan

Setiap panggilan jaringan wajib punya batas waktu. Yang perlu diputuskan adalah “berapa lama”, 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 *error* sementara (`UNAVAILABLE`, 503, 429), dan hanya untuk operasi *idempotent*.
- Batasi jumlah percobaan. *Retry* di setiap *layer* bisa berlipat: 3 *layer* $\times$ 3 percobaan = 27 *request*.
- Di gRPC, pola ini bisa diganti *retry policy* pada *service config*.

Idempotency key: membuat POST aman diulang



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

Ringkasan

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 *field* opsional; menambah *endpoint* baru	Menghapus atau mengganti nama *field*; mengubah tipe; naikkan ke `/v2`
gRPC/Protobuf	Menambah *field* dengan nomor baru; klien lama mengabaikannya	Mengubah atau memakai ulang *field number*; tandai yang dihapus dengan `reserved`
GraphQL	Menambah *field* dan tipe baru	Menghapus *field*; jalur biasanya: tandai `@deprecated`, pantau pemakaian, baru hapus

Ringkasan

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

Memilih: pakai yang mana kapan

Tabel keputusan

Kebutuhan	Pilihan utama	Alasan
API publik untuk pihak ketiga	REST + OpenAPI	Mudah dipakai siapa saja, *caching* HTTP, alat dan dokumentasi melimpah
Antar layanan internal, banyak bahasa, *latency* rendah	gRPC	Kontrak ketat, biner, *streaming*, *deadline*, *codegen*
Frontend dengan banyak layar dan kebutuhan data beragam	GraphQL atau *BFF* (*Backend for Frontend*)	Mengurangi *over-* dan *under-fetching*
Push real-time ke *browser* atau aplikasi	SSE (satu arah) atau *WebSocket* (dua arah)	Server mengirim tanpa ditanya
Pekerjaan yang tidak perlu dijawab segera	*Messaging* (pertemuan 5)	*Decoupling*, menahan lonjakan, *retry* di *broker*
Integrasi dengan sistem mitra lama	SOAP atau format mitra, dibungkus *adapter*	Mengikuti kontrak mitra tanpa menyebarkannya ke dalam

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

Membandingkan semuanya

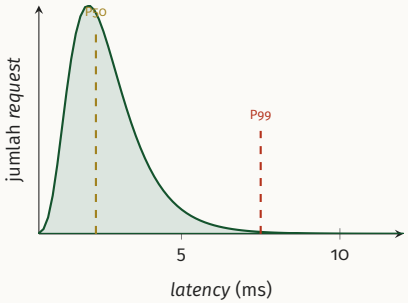
	REST	**gRPC**	**GraphQL**	***WebSocket***
Kontrak	Opsional (OpenAPI)	Wajib (`.proto`)	Wajib (*schema*)	Tidak ada bawaan
Format	Umumnya JSON	Protobuf biner	JSON	Bebas (teks/biner)
Transport	HTTP/1.1, /2, /3	HTTP/2	HTTP; *WebSocket* untuk *subscription*	TCP setelah *upgrade* HTTP
Browser	Ya	Lewat gRPC-Web atau Connect	Ya	Ya
Streaming	Terbatas (SSE)	Ya, empat mode	*Subscription*	Ya, dua arah
Caching HTTP	Sangat baik	Sulit	Sulit (kecuali *persisted query* via GET)	Tidak
Tempat terbaik	API publik, integrasi	Antar *microservices*	*Frontend* dengan kebutuhan data beragam	*Real-time* dua arah

Mengukur: *latency* dan *throughput*

Membaca *latency*: mengapa rata-rata menipu

- **P50** (median) — separuh *request* lebih cepat dari angka ini. Wajah "biasa" layanan.
- **P99** — 99% *request* lebih cepat; 1% sisanya lebih lambat dan disebut *tail latency*.
- ***Throughput*** — berapa *request* 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 *request*. 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. ***Warm-up*** — buang beberapa detik pertama (koneksi baru, JIT, *cache* dingin).
3. **Ulangi** — jalankan minimal tiga kali; laporkan median dan sebarannya.
4. **Pisahkan *load generator*** — bila mungkin, alat ukur jangan berebut CPU dengan *server* yang diukur.
5. **Laporkan persentil** di samping rata-rata, beserta RPS dan *error rate*.
6. **Waspadai *coordinated omission*** — alat yang menunggu jawaban sebelum mengirim *request* berikutnya cenderung menyembunyikan ekor *latency* saat *server* tersendat.

Ringkasan

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` (atau `buf`), `protoc-gen-go`, dan `protoc-gen-go-grpc`.
2. Tulis `katalog.proto`, bangkitkan kodenya, lalu buat *server* gRPC di *port* 50051 dengan `AmbilProduk` (*unary*) dan `PantauHarga` (*server streaming*).
3. Pasang *interceptor auth* + *log*, *health check*, dan *reflection*; uji dengan `grpcurl`.
4. Buat *server* REST di *port* 8080 dengan **data dan logika yang sama**, beserta *file* OpenAPI dan *error* RFC 9457.
5. Di klien, pasang *deadline* 300 ms; buktikan `DEADLINE_EXCEEDED` dengan menambahkan jeda buatan di *server*.
6. Ukur: `ghz` dan `k6` dengan 10, 50, dan 200 pemanggil serentak, tiap *configuration* tiga kali. Isi tabel: P50, P99, RPS, ukuran *payload*, *error rate*.
7. Bonus: layani `.proto` yang sama lewat Connect, lalu panggil dengan `curl` berisi JSON.

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*, *schema* tabel penyimpannya, 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?