Pertemuan 1 — Sistem Terdistribusi 101

Pengenalan sistem terdistribusi dan *high performance computing*

Hendri Karisma

Semester Ganjil 2026/2027

Program Studi Teknik Informatika, STMIK Tazkia

Perkenalan



Hendri Karisma

VP Engineering, jejakin.com
Produk yang ditangani:

- Emission and ESG analytics and calculation
- MRV (*Measurement, Reporting, Verification*) untuk ESG dan proyek karbon
- *Marketplace* proyek ESG, karbon, air, dan REC

Lecturer, STMIK Tazkia
Mengajar: Machine Learning, Artificial Intelligence, Data Engineering, Distributed System, Research Methodology.

Research area
Secure and Sustainable Computation, Artificial Intelligence, Quantum Computing.

Situs hendrikarisma.my.id
GitHub github.com/situkangsayur
LinkedIn linkedin.com/in/hendri-karisma-736a0a39
X @infoHendri Instagram @karism4_

Alur pertemuan hari ini

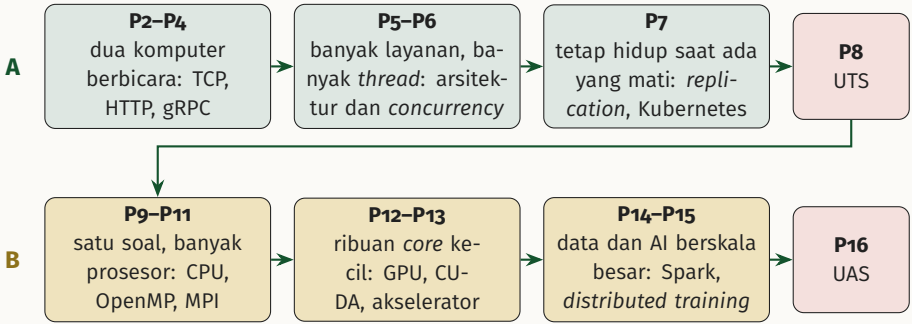
1. Dua capaian semester ini: Sister dan HPC
2. Apa itu sistem terdistribusi
3. Janji: lebih murah, andal, aman, cepat, beserta harganya
4. Bagaimana komputer berkomunikasi
5. Evolusi arsitektur: *monolith* sampai *microservices*
6. *Concurrency* dan dampak bahasa pemrograman
7. Data besar: Hadoop, Spark, Kubernetes
8. CPU, GPU, dan *heterogeneous computing*
9. Dua jalur: *scale & resilience* vs *high performance*
10. Rencana semester dan tugas besar

Sasaran pertemuan ini

Anda dapat menjelaskan mengapa sistem modern dibangun dari banyak komputer, menyebut apa yang didapat dan apa yang dikorbankan, serta menempatkan setiap topik semester ini (protokol, arsitektur, *concurrency*, *big data*, HPC) di dalam satu peta besar.

Dua capaian semester ini: Sistem Terdistribusi dan HPC

Arah mata kuliah: dari dua komputer bicara sampai ribuan GPU bekerja sama



Paruh pertama menjawab “bagaimana sistem tetap **benar dan hidup** walau terpecah di banyak mesin”. Paruh kedua menjawab “bagaimana memakai banyak prosesor agar **secepat mungkin**”. Pertanyaan dasarnya sama: bagaimana banyak komputer berkomunikasi.

Capaian A — Sistem Terdistribusi (pertemuan 1–7, UTS)

Di akhir paruh pertama, Anda mampu:

1. **Menimbang** kapan sistem perlu terdistribusi: *scale-up* vs *scale-out*, *availability*, *cost*, *latency*, dan delapan *fallacies*.
2. **Menjelaskan dan memakai** protokol komunikasi: *socket*, TCP, UDP, QUIC, HTTP/1.1–3, TLS/HTTPS.
3. **Memilih dan membangun** gaya API yang tepat (REST, gRPC, GraphQL, RPC/RMI, *messaging*) berdasarkan hasil **pengukuran**.
4. **Merancang** arsitektur 1/2/3/*n-tier*, *monolith*, dan *microservices*, lengkap dengan alasan dan *cost*-nya.
5. **Menulis** program *concurrent* dengan *thread*, *goroutine*, *actor*, dan *reactive streams*.
6. **Membuat sistem tahan gagal**: *replication*, *consistency*, CAP, gagasan *consensus* (Raft), *container* dan Kubernetes yang memulihkan diri.

Capaian B — *High Performance Computing* (pertemuan 9–15, UAS)

Di akhir paruh kedua, Anda mampu:

1. **Mengukur performa dengan benar**: *speedup*, efisiensi, *strong/weak scaling*, Amdahl dan Gustafson, FLOPS, *roofline*.
2. **Menjelaskan** mengapa kode cepat atau lambat di CPU: *cache*, *memory wall*, NUMA, SIMD; lalu memparalelkannya dengan OpenMP.
3. **Menjalankan** komputasi di *cluster* dengan MPI dan memahami interkoneksi cepat (InfiniBand, RDMA).
4. **Menulis** *kernel* GPU sederhana dengan CUDA dan menjelaskan *heterogeneous computing* (CPU, GPU, TPU, FPGA).
5. **Menerapkan** pola paralel data besar (MapReduce, Spark) dan menjelaskan cara melatih model AI di banyak GPU (*data, tensor, pipeline parallelism*).
6. **Menimbang** performa terhadap *cost* dan energi.

Berjalan bersama Data Engineering (semester ini)

Di Sistem Terdistribusi: “mengapa ia cepat dan tahan”

- *Partition*, *replication*, *consistent hashing*, *consistency*
- MapReduce dan Spark sebagai **pola paralel**: *data locality*, *shuffle*, DAG, toleransi kegagalan
- *Cache*, SIMD, GPU: alasan *hardware* di balik *columnar* dan *vectorized execution*
- Pengukuran *speedup* dan *scaling*

Di Data Engineering: “bagaimana memakainya”

- *Pipeline* ETL/ELT, *data lake* dan *lakehouse*
- Hadoop dan Spark **sepenuhnya di DE** (P9–P10), termasuk Spark SQL dan *tuning*; di sini hanya disebut
- Kafka diperkenalkan di sini (P5), dipakai mendalam di DE (P11)
- *Data quality*, *data modeling*, dan *data governance*

Ringkasan

Kedua mata kuliah berjalan bersamaan: Data Engineering **memakai dan memilih** alat, Sister **membedah mesinnya**. Bila Anda sudah memakai sesuatu di DE, di sini Anda akan melihat mengapa ia bekerja.

Apa itu sistem terdistribusi

Dua definisi, dua sudut pandang

Tanenbaum dan van Steen

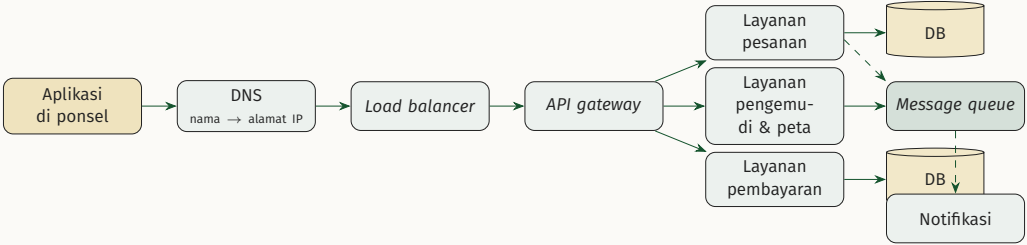
Sistem terdistribusi adalah kumpulan elemen komputasi otonom yang tampak bagi penggunanya sebagai **satu sistem yang koheren**.

Leslie Lamport (1987), setengah bercanda

"*A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.*"

Definisi pertama adalah **janjinya**: banyak mesin, terlihat seperti satu. Definisi kedua adalah **kenyataannya**: banyak mesin berarti banyak titik yang bisa gagal, dan kegagalan itu sering tidak terlihat sampai terasa.

Anda memakainya setiap hari: memesan ojek online



Satu ketukan “Pesan” melewati puluhan mesin, mungkin di beberapa kota, dalam waktu kurang dari satu detik. Pengguna hanya melihat satu aplikasi. Itulah “tampak sebagai satu sistem yang koheren”.

Ringkasan

Sepanjang semester kita akan membedah setiap panah di gambar ini: lewat protokol apa ia berjalan, bagaimana kalau ia putus, dan berapa *cost*-nya.

Mengapa tidak satu komputer saja?

- **Skala** — jutaan pengguna dan *petabyte* data tidak muat di satu mesin, berapa pun harganya.
- ***Availability*** — satu mesin pasti suatu hari mati. Kalau hanya ada satu, layanan ikut mati.
- **Jarak** — cahaya pun butuh waktu. Pengguna di Jakarta dan Amsterdam tidak bisa sama-sama dekat ke satu *server*.
- ***Cost*** — seribu mesin biasa sering lebih murah daripada satu mesin raksasa dengan kemampuan setara.
- **Kecepatan** — pekerjaan besar yang dipecah ke banyak prosesor selesai lebih cepat.
- **Organisasi** — tim yang berbeda dapat memiliki, merilis, dan memperbaiki bagiannya sendiri.

Ringkasan

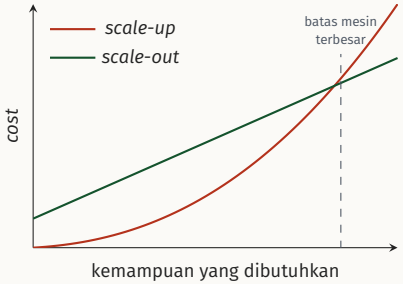
Kita tidak memilih sistem terdistribusi karena keren. Kita terpaksa, ketika salah satu batas di atas tercapai. Bila belum, satu mesin yang dirawat baik sering adalah jawaban terbaik.

Tiga janji: lebih murah, lebih andal, lebih cepat

Janji 1 — Lebih murah: *scale-up* vs *scale-out*

- ***Scale-up*** (*vertical*) — ganti dengan mesin yang lebih besar: lebih banyak *core*, RAM, dan disk dalam satu kotak. Harga naik jauh lebih cepat daripada kemampuannya, dan ada batas fisik.
- ***Scale-out*** (*horizontal*) — tambah mesin biasa (*commodity*). Harga naik kira-kira sebanding dengan jumlah mesin.

Google pada akhir 1990-an memilih ribuan PC biasa yang murah dan sering rusak, lalu menulis *software* yang tahan terhadap kerusakan itu. Pilihan ini menjadi pola *warehouse-scale computing*.



Ilustrasi bentuk kurva, bukan harga riil.

Tapi scale-out tidak gratis

- Mesin murah lebih sering rusak. Pada ribuan mesin, “kerusakan langka” menjadi **kejadian harian**.
- *Software*-nya jadi jauh lebih rumit: data harus dibagi, disalin, disinkronkan.
- Sebagian *cost* berpindah: dari membeli *hardware* ke membayar *engineer* dan *operations*.

Ringkasan

Scale-out memindahkan masalah dari “beli mesin yang tidak pernah rusak” (mustahil dan mahal) menjadi “tulis *software* yang tetap jalan walau mesinnya rusak” (sulit, tetapi bisa). Mata kuliah ini sebagian besar tentang kesulitan itu.

Janji 2 — Lebih andal: matematika *redundancy*

Misalkan satu *server* punya *availability* 99%. Bila dua *server* identik bekerja **saling bebas** dan cukup satu yang hidup:

$$A = 1 - (1 - 0{,}99)^2 = 99{,}99\%$$

Secara umum, untuk *n replica*: $A = 1 - (1 - a)^n$ (A: *availability* gabungan, a: *availability* satu *replica*).

Ukuran lain yang akan kita pakai:

- **MTBF** — rata-rata waktu antar kegagalan
- **MTTR** — rata-rata waktu untuk pulih
- $A = \dfrac{\text{MTBF}}{\text{MTBF} + \text{MTTR}}$: pulih cepat sama berharganya dengan jarang rusak

Availability	***Downtime* per tahun**
99% (“two nines”)	3,65 hari
99,9%	8,76 jam
99,99%	52,6 menit
99,999% (“five nines”)	5,26 menit

Jebakan kata “saling bebas”

Dua *server* di rak yang sama, listrik yang sama, dan versi *software* yang sama akan mati **bersama**. Karena itu ada konsep *fault domain*: rak, *availability zone*, *region*.

Janji 3 — Lebih aman? Tergantung desain

Yang membuat lebih aman

- **Isolasi** — layanan pembayaran terpisah dari layanan katalog; jebolnya satu tidak otomatis membuka yang lain (*blast radius* kecil)
- ***Least privilege*** — tiap layanan hanya diberi akses yang ia butuhkan
- ***Defense in depth*** — TLS, mTLS antar layanan, *zero trust*
- ***Replication*** — tahan terhadap rusaknya data dan serangan pada satu titik

Yang membuat lebih rawan

- *Attack surface* membesar: setiap panah adalah jalur jaringan yang bisa disadap atau dipalsukan
- Rahasia (*secret*, kunci, token) tersebar di banyak tempat
- *Configuration* yang banyak berarti banyak peluang *misconfiguration*
- Log dan jejak audit tersebar, sehingga serangan lebih sulit dilacak

Ringkasan

Sistem terdistribusi tidak otomatis lebih aman. Ia memberi **alat** untuk menjadi lebih aman, dan sekaligus lebih banyak **celah** bila alat itu tidak dipakai.

Janji 4 — Lebih cepat: hukum Amdahl

Bila bagian p dari pekerjaan dapat diparalelkan ke N prosesor:

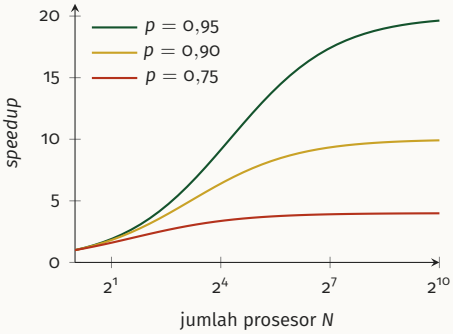
$$S(N) = \frac{1}{(1-p) + \frac{p}{N}}$$

Dengan $p = 0{,}95$:

- $N = 8 \quad \Rightarrow S \approx 5{,}9\times$
- $N = 64 \quad \Rightarrow S \approx 15{,}4\times$
- $N \to \infty \Rightarrow S \to 20\times$, tidak lebih

$S(N)$: *speedup*, berapa kali lebih cepat dengan N prosesor; $\approx$: kira-kira sama dengan; $\Rightarrow$: sehingga; $N \to \infty$: N dibuat makin besar tanpa batas (∞ dibaca *infinity*, tak hingga). Bagian serial 5% itulah yang akhirnya menentukan.

Gustafson menambahkan sisi yang lebih optimis: dengan lebih banyak prosesor, kita biasanya mengerjakan **masalah yang lebih besar** dalam waktu yang sama.



Harga yang dibayar: jaringan tidak pernah sempurna

Delapan *fallacies of distributed computing*

Peter Deutsch dan rekan di Sun Microsystems (1994–1997) mendaftar asumsi keliru yang hampir selalu dibuat pemrogram pemula:

1. Jaringan itu andal
2. *Latency* itu nol
3. *Bandwidth* tidak terbatas
4. Jaringan itu aman
5. Topologi tidak berubah
6. Hanya ada satu administrator
7. *Transport cost* itu nol
8. Jaringan itu homogen

Ringkasan

Hampir setiap teknik yang akan kita pelajari (*timeout*, *retry*, *idempotency*, *replication*, TLS, *service discovery*, *serialization* yang ringkas) ada untuk menambal salah satu dari delapan asumsi ini.

Seberapa jauh “jauh” itu? *Latency numbers*

Operasi	Waktu (kira-kira)	Bila 1 ns = 1 detik
Akses *L1 cache*	1 ns	1 detik
Akses *main memory* (DRAM)	100 ns	1,7 menit
Baca acak dari SSD NVMe	20–100 µs	6 jam – 1 hari
Round trip dalam satu *data center*	500 µs	5,8 hari
Seek pada *hard disk*	10 ms	4 bulan
Round trip Jakarta – Eropa – Jakarta	∼150–200 ms	5–6 tahun

Sumber: diadaptasi dari “Latency Numbers Every Programmer Should Know” (Jeff Dean, Peter Norvig); angka berubah mengikuti generasi *hardware*.

Ringkasan

Memanggil fungsi di memori dan memanggil layanan lewat jaringan **terlihat** sama di kode, tetapi berbeda ribuan sampai jutaan kali lipat.

Ini alasan mengapa “pecah saja jadi *microservices*” tanpa berpikir bisa membuat sistem jauh lebih lambat.

Dua hal yang membuat sistem terdistribusi sulit

Partial failure

Di satu komputer, bila ada yang rusak, biasanya semuanya berhenti. Di sistem terdistribusi, **sebagian** rusak dan sebagian tetap jalan.

Anda mengirim permintaan dan tidak mendapat jawaban. Apakah permintaannya hilang? Servernya mati? Servernya sudah memproses tetapi jawabannya yang hilang? **Anda tidak bisa tahu.**

Tidak ada jam bersama

Tiap mesin punya jam sendiri yang bergeser. Urutan "mana yang terjadi duluan" tidak bisa sekadar dibaca dari *timestamp*.

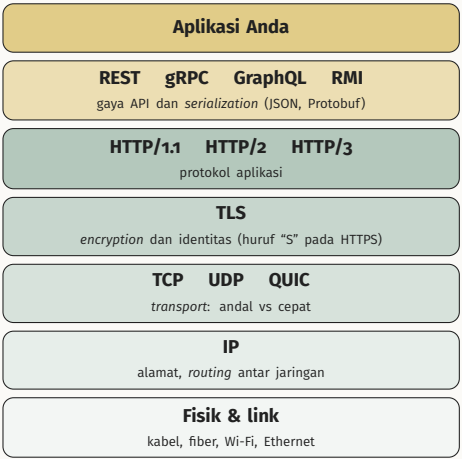
Dua orang menarik saldo yang sama pada saat "bersamaan" dari dua *server*. Siapa yang menang? Inilah pintu masuk ke *logical clock*, *replication*, dan *consensus* (Paxos, Raft).

Ringkasan

Inti mata kuliah ini: bagaimana membangun sistem yang **benar** di atas komponen yang **tidak dapat diandalkan**, dengan informasi yang **selalu terlambat**.

Bagaimana komputer berkomunikasi

Stack komunikasi yang akan kita bongkar



Setiap *layer* menyembunyikan kerumitan *layer* di bawahnya, dan masing-masing menambah sedikit *latency* serta *overhead*.

- Pertemuan 2: IP, *socket*, TCP, UDP, QUIC
- Pertemuan 3: HTTP/1.1, /2, /3, TLS dan HTTPS
- Pertemuan 4: RPC, Java RMI, REST, gRPC, GraphQL

Ringkasan

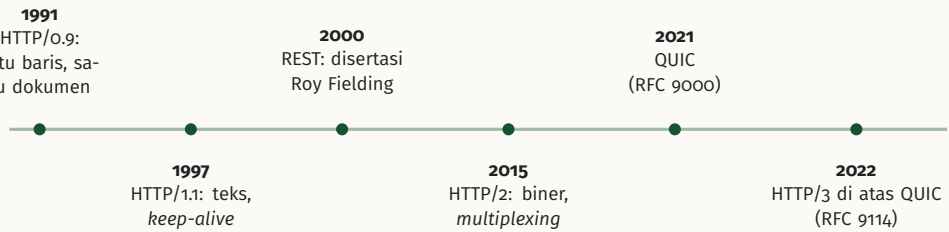
Insinyur yang baik tahu **di *layer* mana** masalahnya terjadi sebelum mencoba memperbaikinya.

TCP, UDP, dan QUIC dalam satu tabel

	TCP	UDP	QUIC
Koneksi	Ada *handshake* sebelum kirim	Tanpa koneksi, langsung kirim	Ada, digabung dengan TLS 1.3
Reliability	Terjamin sampai, urut, tanpa duplikat	Bisa hilang, tertukar, ganda	Terjamin, per *stream*
Head-of-line blocking	Ya: satu *packet* hilang menahan semua	Tidak ada	Hanya pada *stream* yang terdampak
Contoh pemakaian	Web (HTTP/1.1, /2), *database*, SSH	DNS, *video call*, game, *streaming* langsung	HTTP/3, aplikasi *mobile* modern

QUIC berjalan **di atas UDP**, tetapi menambahkan sendiri *reliability* dan *encryption* di *user space*. Ini contoh bagus bahwa protokol pun berevolusi ketika asumsinya berubah: jaringan seluler yang sering berganti alamat IP tidak cocok dengan koneksi TCP yang terikat pada alamat.

HTTP dari 1.1 ke 3: menambal *latency*



- **HTTP/1.1** — satu koneksi melayani satu permintaan pada satu waktu. *Browser* membuka 6 koneksi sekaligus untuk menyiasatinya.
- **HTTP/2** — banyak *stream* dalam satu koneksi TCP, header dikompresi. Tetapi satu *packet* TCP hilang masih menahan semua *stream*.
- **HTTP/3** — pindah ke QUIC/UDP: *handshake* lebih singkat, *packet loss* hanya menahan *stream* yang bersangkutan, koneksi bertahan saat ganti jaringan.
- **HTTPS** — HTTP yang dibungkus TLS: isi terenkripsi dan identitas *server* dibuktikan dengan *certificate*.

Cara memanggil komputer lain: dari RPC sampai GraphQL

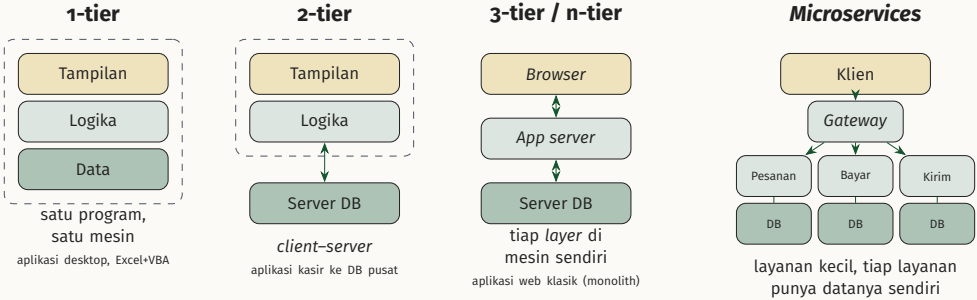
Gaya	Lahir	Gagasan utama	Cocok untuk
RPC (Birrell & Nelson)	1984	Memanggil prosedur di mesin lain seolah fungsi lokal	Fondasi semua yang di bawah
Java RMI, CORBA	1990-an	RPC berorientasi objek: memanggil *method* pada objek jarak jauh	Sistem Java internal (kini jarang di sistem baru)
REST	2000	Sumber daya beralamat URL, operasi lewat *verb* HTTP, umumnya JSON	API publik, web, integrasi antar perusahaan
gRPC	2015	RPC modern: kontrak Protobuf, biner, di atas HTTP/2, mendukung *streaming*	Komunikasi antar *microservices* yang butuh cepat dan bertipe ketat
GraphQL	2015	Klien menyebut persis data apa yang dibutuhkan dalam satu *query*	Aplikasi *mobile*/web dengan kebutuhan data beragam
Messaging (Kafka, RabbitMQ)	2000-an	Kirim *message* ke *queue*, penerima memproses kapan pun ia siap	Komunikasi *asynchronous*, peredam lonjakan beban

Ringkasan

Ilusi “seolah fungsi lokal” pada RPC itu berbahaya: fungsi lokal tidak pernah *timeout* dan tidak pernah dijalankan dua kali. Panggilan jaringan bisa keduanya.

Evolusi arsitektur: *monolith* sampai *microservices*

Dari *1-tier* sampai *microservices*



Setiap langkah ke kanan menambah **kebebasan** (skala per bagian, rilis per tim, teknologi per layanan) dan sekaligus menambah **panah jaringan** yang bisa gagal.

Monolith atau *microservices*?

Monolith

- Satu kode, satu *deploy*, satu *database*
- Panggilan antar modul = panggilan fungsi, cepat dan pasti
- Transaksi *database* sederhana
- Sulit diskalakan per bagian; satu *bug* bisa menjatuhkan semuanya
- Tim besar saling menunggu saat rilis

Microservices

- Tiap layanan di-*deploy* dan diskalakan sendiri
- Kegagalan bisa diisolasi
- Tim kecil memiliki layanannya sendiri
- Panggilan antar layanan = jaringan: lambat, bisa gagal
- Butuh *observability*, *service discovery*, *orchestration*

Ringkasan

Conway's law: struktur sistem cenderung meniru struktur organisasi yang membuatnya. *Microservices* adalah jawaban untuk masalah **organisasi** dan **skala**, bukan gaya yang otomatis lebih baik. Banyak tim memulai dengan *monolith* yang modular, lalu memecahnya ketika memang perlu.

Concurrency: mengerjakan banyak hal sekaligus

Concurrency bukan *parallelism*

Concurrency

Menangani banyak hal **dalam rentang waktu yang sama**: satu koki mengurus tiga panci bergantian.

Soal **struktur** program: bagaimana memecah pekerjaan menjadi bagian yang bisa berjalan sendiri-sendiri.

Parallelism

Mengerjakan banyak hal **pada detik yang sama**: tiga koki memasak tiga panci bersamaan.

Soal **eksekusi**: butuh lebih dari satu *core*, mesin, atau GPU.

Rob Pike: "*Concurrency is about dealing with lots of things at once. Parallelism is about doing lots of things at once.*" Sistem terdistribusi selalu *concurrent*; ia menjadi paralel ketika pekerjaannya benar-benar dibagi ke banyak prosesor.

Lima model *concurrency* yang akan kita coba

Model	Contoh	Cara berbagi kerja	Risiko khas
Thread + lock	Java, C++, Python	Banyak *thread* berbagi memori, dijaga *mutex*	*Race condition*, *deadlock*
CSP: *goroutine + channel*	Go	"Jangan berkomunikasi dengan berbagi memori; berbagilah memori dengan berkomunikasi"	*Goroutine leak*, *channel* macet
Actor model	Erlang/Elixir, Akka (Scala/Java)	Tiap *actor* punya *state* sendiri, hanya berkomunikasi lewat *message*	*Mailbox* menumpuk, urutan *message*
Reactive streams	RxJava, RxJS, Project Reactor	Data sebagai aliran, diolah dengan operator; ada *backpressure*	Sulit di-*debug*, rantai operator panjang
Event loop / *async*	Node.js, Python `asyncio`	Satu *thread*, tidak pernah menunggu I/O secara diam	Satu tugas berat membekukan semuanya

Ringkasan

Actor model menarik karena ia sama di satu mesin maupun di banyak mesin: *message* ke *actor* lokal dan *message* ke *actor* di *server* lain ditulis dengan cara yang sama. Itu jembatan alami dari *concurrency* ke sistem terdistribusi.

Rasanya di kode: Go dan Scala (Akka)

Go — satu *goroutine* per URL, hasil lewat *channel*

```go
func main() {
    hasil := make(chan string)
    for _, url := range daftarURL {
        // ribuan goroutine itu murah
        go func(u string) {
            hasil <- ukurLatency(u)
        }(url)
    }
    for range daftarURL {
        fmt.Println(<-hasil)
    }
}
```

Scala + Akka — *actor* penghitung, tanpa *lock*

```scala
object Penghitung {
  sealed trait Pesan
  case object Tambah extends Pesan
  case class Ambil(balas: ActorRef[Int])
    extends Pesan

  def apply(n: Int = 0): Behavior[Pesan] =
    Behaviors.receiveMessage {
      case Tambah       => apply(n + 1)
      case Ambil(balas) => balas ! n
                           Behaviors.same
    }
}
```

Keduanya menghindari berbagi variabel antar *thread*. Go memakai *channel*; Akka memakai *mailbox*. Pada pertemuan 6 kita bandingkan keduanya dengan versi *thread + lock* di Java.

Bahasa pemrograman ikut menentukan

Pilihan bahasa berdampak pada skala dan performa

Bahasa	Cara dijalankan	Model *concurrency* bawaan	Kekuatan	Harga yang dibayar
C, C++, Rust	*Native*, tanpa *garbage collector*	*Thread* OS; Rust menjamin bebas *data race* saat kompilasi	Performa puncak, kontrol memori, SIMD; bahasa utama HPC	Lebih sulit ditulis; C/C++ rawan bug memori
Go	*Native*, dengan GC	*Goroutine* + *channel*, ringan sekali	Layanan jaringan, Docker dan Kubernetes ditulis dengan Go	GC bisa memicu lonjakan *latency*
Java, Scala, Kotlin	JVM, JIT, dengan GC	*Thread*, *virtual thread*, Akka	Ekosistem besar: Hadoop, Spark, Kafka	*Warm-up* JIT, memori besar, jeda GC
Erlang, Elixir	VM BEAM	Jutaan proses ringan (*actor*), supervisi	*High availability*: telekomunikasi, WhatsApp	Lemah untuk komputasi angka berat
JavaScript (Node.js)	JIT, satu *thread*	*Event loop*, *async*	Banyak koneksi I/O dengan kode sederhana	Satu tugas CPU berat membekukan semua
Python	*Interpreter*, dengan GIL	*Thread* terhalang GIL; *multiprocessing*, *asyncio*	Cepat ditulis; perekat untuk NumPy, PyTorch, PySpark	Loop Python murni lambat
CUDA, Fortran	*Native* di GPU/CPU	Ribuan *thread* GPU; MPI, OpenMP	Simulasi ilmiah dan AI	Terikat *hardware* tertentu

Ringkasan

Bahasa menentukan tiga hal: seberapa dekat kode ke *hardware*, bagaimana memori dikelola, dan model *concurrency* apa yang murah dipakai.

Contoh nyata: bahasa mengubah arsitektur

- **Python + NumPy/PyTorch** — kodenya Python, kerja beratnya C, C++, dan CUDA. Satu operasi vektor NumPy bisa puluhan sampai ratusan kali lebih cepat daripada loop Python biasa. Pola "Python di luar, *native* di dalam" adalah wajah umum HPC dan AI sekarang.
- **GIL Python** — *thread* Python tidak bisa memakai banyak *core* sekaligus untuk kerja CPU. Solusinya *multiprocessing*, *native library*, atau Python *free-threaded* yang mulai diuji sejak Python 3.13.
- ***Garbage collector* dan *tail latency*** — Discord (2020) menulis ulang satu layanannya dari Go ke Rust karena jeda GC berkala menimbulkan lonjakan *latency*.
- **Erlang dan WhatsApp** — proses BEAM yang ringan memungkinkan satu *server* menampung jutaan koneksi, dan supervisi membuat proses yang gagal dihidupkan ulang otomatis.
- **JVM dan *big data*** — Hadoop, Spark, dan Kafka hidup di JVM, sehingga *tuning* memori dan GC menjadi pekerjaan sehari-hari di Data Engineering.

Ringkasan

Tidak ada bahasa terbaik. Yang ada: bahasa yang paling cocok untuk **bagian** tertentu dari sistem. Sistem besar hampir selalu *polyglot*.

Data besar: Hadoop, Spark, dan Kubernetes

Mengapa data besar memaksa kita terdistribusi

Hitungan kasar membaca 1 TB data:

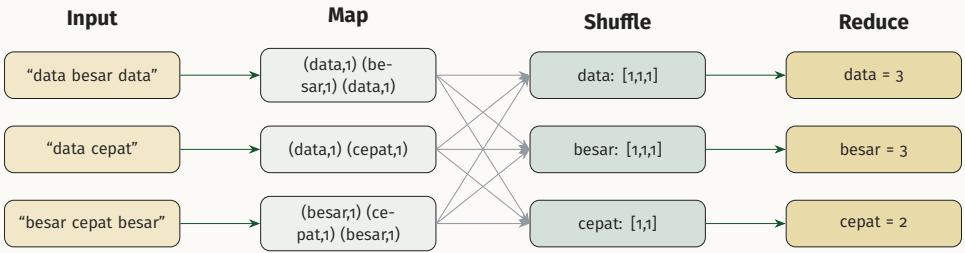
- Satu disk, sekitar 200 MB/s:
  $10^{12} / (2 \times 10^{8}) = 5.000$ detik $\approx$ **83 menit**
- 1.000 disk dibaca paralel:
  $\approx$ **5 detik**

Gagasan kuncinya: **pindahkan komputasi ke data**, bukan data ke komputasi. Program berukuran kilobyte dikirim ke ribuan mesin yang masing-masing menyimpan sepotong data.

Silsilah singkat

2003	Google File System (GFS)
2004	MapReduce (Dean & Ghemawat)
2006	Hadoop: HDFS + MapReduce *open source*
2010	Apache Spark: data di memori, DAG
2011	Apache Kafka: *log* terdistribusi
2014	Kubernetes, penerus Borg di Google

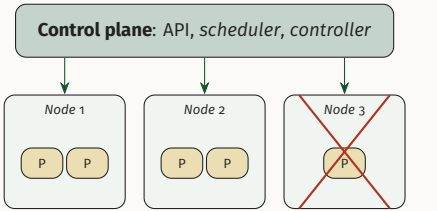
MapReduce: menghitung kata di ribuan mesin



- Pemrogram hanya menulis dua fungsi: `map` dan `reduce`. Pembagian kerja, pengiriman data, dan pengulangan tugas yang gagal diurus oleh *framework*.
- **Spark** menyimpan hasil antara di memori dan merangkai langkah sebagai DAG, sehingga jauh lebih cepat untuk pekerjaan iteratif seperti *machine learning*.

Kubernetes: “sistem operasi” untuk satu *cluster*

- ***Container*** (Docker) — aplikasi beserta semua kebutuhannya dalam satu paket yang berjalan sama di mana pun.
- **Kubernetes** — Anda menulis *desired state* (“jalankan 5 *replica* layanan pembayaran”), Kubernetes terus mencocokkan kenyataan dengannya.
- Pod mati? Dibuat ulang. *Node* mati? Dipindah ke *node* lain. Beban naik? *Replica* ditambah otomatis (*autoscaling*).



Ringkasan

Kubernetes adalah wujud nyata gagasan *design for failure*: ia tidak berusaha mencegah mesin mati, tetapi memastikan sistem pulih sendiri ketika itu terjadi.

CPU, GPU, dan *heterogeneous computing*

CPU dan GPU: dua filosofi prosesor

CPU — *latency-oriented*

- Sedikit *core* (8–128), masing-masing sangat pintar
- *Cache* besar, *branch prediction*, *out-of-order execution*
- Unggul untuk logika bercabang dan tugas berurutan

GPU — *throughput-oriented*

- Ribuan *core* sederhana yang mengerjakan instruksi sama pada data berbeda (SIMT)
- Memori ber-*bandwidth* sangat tinggi (HBM)
- Unggul untuk perkalian matriks, citra, simulasi, *deep learning*

Heterogeneous computing: memakai CPU, GPU, dan akselerator khusus (TPU, FPGA) bersama-sama, masing-masing untuk bagian yang paling cocok. CPU mengatur, GPU mengerjakan bagian paralel yang masif.

Rujukan yang layak diikuti

Kuliah *Heterogeneous Parallel Programming* oleh Prof. Wen-mei Hwu (University of Illinois Urbana-Champaign) pernah tersedia di Coursera. Materinya kini terbuka lewat buku *Programming Massively Parallel Processors* (Kirk, Hwu, El Hajj) dan rekaman kuliah PMPP/ECE 408 yang bisa ditemukan di YouTube.

Dari satu GPU ke HPC: ketika mesin harus bicara sangat cepat

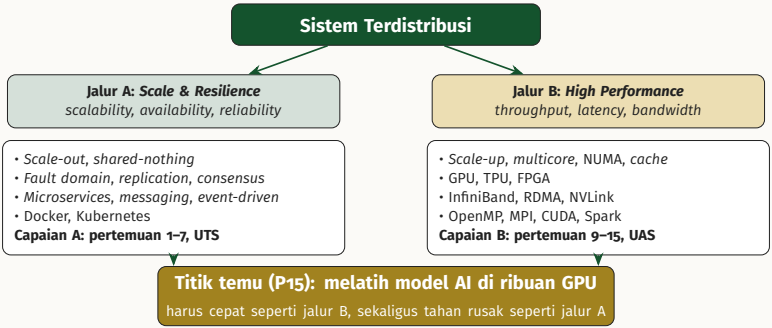
- ***High Performance Computing*** (HPC) — *cluster* ratusan sampai puluhan ribu *node* untuk satu pekerjaan: prakiraan cuaca, simulasi iklim, *training* LLM.
- Di sini TCP/IP biasa terlalu lambat. Dipakai interkoneksi khusus: **InfiniBand**, **RDMA** (memori satu mesin dibaca langsung oleh mesin lain tanpa melewati sistem operasi), **NVLink** antar GPU.
- Model pemrogramannya juga berbeda: **MPI** untuk antar *node*, **OpenMP** untuk antar *core*, **CUDA** untuk GPU, NCCL untuk komunikasi antar GPU.

Ringkasan

Perhatikan bahwa pertanyaannya tetap sama dengan *microservices*: bagaimana banyak komputer berkomunikasi. Yang berbeda adalah **anggaran waktunya**: milidetik untuk layanan web, mikrodetik atau kurang untuk HPC.

Dua jalur: *scale & resilience* vs *high performance*

Peta dua jalur mata kuliah ini



Cara membacanya: jalur A menjaga sistem tetap **hidup dan benar** walau mesinnya rusak; jalur B membuat komputasi **secepat mungkin**. Melatih model seperti ChatGPT atau Llama memakai ribuan GPU sekaligus, dan di situ kedua kemampuan dibutuhkan bersamaan. Karena itu ia menjadi materi penutup semester.

Dua jalur, dua cara berpikir

	Jalur A: *Scale & Resilience*	**Jalur B: *High Performance***
Cara menambah daya	*Scale-out*: tambah ribuan mesin biasa	*Scale-up* + *cluster* padat: mesin sangat khusus
Model memori	*Shared-nothing*: tiap mesin memorinya sendiri	*Shared memory*/koheren: NUMA, HBM, NVLink, CXL
Ukuran keberhasilan	SLA 99,99%, *tail latency* P_{99}	FLOPS, *throughput*, *bandwidth*, *latency* minimum
Sikap terhadap kegagalan	*Node* pasti mati kapan saja; sistem tetap melayani	*Fail-stop*: berhenti, muat ulang *checkpoint*, lanjutkan
Jaringan	Ethernet, TCP/IP, *spine-leaf*	InfiniBand, RoCE/RDMA, *lossless fabric*
Abstraksi khas	REST/gRPC, Raft, *consistent hashing*, Kafka	SIMD, *systolic array*, OpenMP, MPI, NCCL
Contoh sistem	*E-commerce*, perbankan, ojek online	Prakiraan cuaca, simulasi, *training* LLM

Titik temu: melatih model AI raksasa

Melatih satu LLM besar membutuhkan **kedua** jalur sekaligus:

- **Jalur B** — tiap GPU harus bekerja nyaris penuh, data mengalir lewat HBM dan NVLink tanpa tersendat.
- **Jalur A** — belasan ribu GPU harus tetap jalan walau setiap hari ada yang rusak: deteksi otomatis, *checkpoint* cepat, lanjut tanpa mengulang dari awal.

Data nyata

Saat melatih Llama 3 405B di sekitar 16 ribu GPU H100, Meta mencatat **466 gangguan** pekerjaan dalam 54 hari, rata-rata satu kali tiap ± 3 jam, sebagian besar karena *hardware*.

Sumber: Meta, *The Llama 3 Herd of Models*, 2024.

Ringkasan

Pada skala itu, kerusakan terjadi setiap hari, seperti **cuaca**. Sistem yang baik dirancang untuk tetap berjalan dalam kondisi itu, dan ke sana materi semester ini menuju di akhir semester.

Rencana satu semester

Peta mata kuliah

Sebelum UTS — Sistem Terdistribusi

1. Sistem Terdistribusi 101: Sister dan HPC (hari ini)
2. Jaringan: IP, *socket*, TCP, UDP, QUIC
3. HTTP/1.1, HTTP/2, HTTP/3, TLS dan HTTPS
4. RPC, Java RMI, REST, gRPC, GraphQL
5. Arsitektur: *n-tier*, *microservices*, *messaging*
6. *Concurrency*: *thread*, *goroutine*, *actor*, Rx
7. *Replication*, *consistency*, Raft, Kubernetes
8. **UTS — tugas besar**

Setelah UTS — *High Performance Computing*

9. Pengantar HPC: *metric*, Amdahl, *scaling*
10. CPU, *cache*, NUMA, SIMD, OpenMP
11. *HPC cluster*: MPI, InfiniBand, RDMA
12. GPU dan CUDA: SIMT, hierarki memori
13. *Heterogeneous computing*: *GPU optimization*, TPU, FPGA
14. Paralelisme data besar: MapReduce, Spark
15. *AI cluster*: *distributed training*
16. **UAS — tugas besar**

Dua tugas besar

UTS — “Satu sistem, tiga cara bicara”

- Bangun sistem kecil (misalnya pesanan, pembayaran, notifikasi) dari minimal **tiga layanan**
- Layanan berkomunikasi lewat **REST**, **gRPC**, dan satu jalur *asynchronous* (*message queue* atau *actor*)
- Ukur dan bandingkan: *latency* P_{50}/P_{99} dan *throughput* REST vs gRPC
- Jalankan dengan *replica*; matikan satu layanan atau *replica*, laporkan apa yang terjadi dan berapa lama pulih
- Luaran: repositori, `docker compose` atau manifest Kubernetes, laporan, demo

UAS — “Satu soal, makin banyak prosesor”

- Pilih satu komputasi berat: perkalian matriks, pengolahan citra, simulasi *n-body*, atau *aggregation* data besar
- Kerjakan bertahap: serial → *multicore* (OpenMP atau *goroutine*) → banyak *node* (MPI atau Spark) → GPU (CUDA, boleh lewat Colab)
- Ukur *speedup*, efisiensi, *strong*/*weak scaling*; bandingkan dengan hukum Amdahl
- Jelaskan **mengapa** angkanya begitu: *cache*, *bandwidth*, komunikasi, bagian serial
- Luaran: repositori, data pengukuran, laporan, presentasi

Dikerjakan kelompok 3–4 orang. Setiap klaim “lebih cepat” atau “lebih andal” wajib disertai angka hasil pengukuran sendiri.

Penilaian dan aturan main

Komponen nilai

Tugas, praktikum, dan kuis	20%
UTS — tugas besar	35%
UAS — tugas besar	45%

UTS dan UAS berbentuk **tugas besar**, tanpa ujian tertulis: kode, eksperimen terukur, laporan, dan presentasi.

Aturan main

- Praktikum memakai Go, Python, Java/Scala, dan C/CUDA; Docker wajib terpasang
- Setiap klaim performa harus disertai **angka terukur**, bukan kesan
- Boleh memakai alat bantu AI, asalkan **dituliskan** bagian mana yang dibantu dan Anda mampu menjelaskan kodenya
- Menyalin pekerjaan orang lain tanpa menyebut sumber dinilai nol

Referensi

- **Maarten van Steen dan Andrew S. Tanenbaum**, *Distributed Systems*, edisi ke-4, 2023.
  Acuan utama; versi digital pribadi dapat diunduh gratis di `distributed-systems.net`.
- **Martin Kleppmann**, *Designing Data-Intensive Applications*, O'Reilly.
  Replication, *consistency*, dan pemrosesan data besar dari sudut pandang praktisi.
- **James Kurose dan Keith Ross**, *Computer Networking: A Top-Down Approach*.
  Untuk TCP, UDP, HTTP, dan QUIC.
- **Luiz André Barroso, Urs Hölzle, Parthasarathy Ranganathan**, *The Datacenter as a Computer*, edisi ke-3, 2018.
  Jalur A: *warehouse-scale computing*.
- **David Kirk, Wen-mei Hwu, Izzat El Hajj**, *Programming Massively Parallel Processors*, edisi ke-4, 2022.
  Jalur B: GPU dan *heterogeneous computing*.
- **John Hennessy dan David Patterson**, *Computer Architecture: A Quantitative Approach*.
  Jalur B: *cache*, *multicore*, akselerator.
- Kuliah terbuka: **MIT 6.5840** *Distributed Systems* (dulu 6.824) dan kuliah *Distributed Systems* Martin Kleppmann (Cambridge), keduanya ada di YouTube.

Tugas untuk pertemuan berikutnya

1. Baca van Steen & Tanenbaum Bab 1.
2. Pilih satu aplikasi yang Anda pakai setiap hari (ojek online, *e-wallet*, *marketplace*, *streaming*). Gambarkan dugaan arsitekturnya seperti slide “memesan ojek online”, lalu tuliskan:
   - tiga komponen yang menurut Anda paling sering diakses;
   - satu komponen yang bila mati membuat seluruh aplikasi tidak berguna;
   - satu *fallacy* dari delapan yang paling mungkin menjebak pengembangnya.
3. Eksperimen kecil di laptop: jalankan `ping`, `traceroute`, dan `curl -w "%{time_total}"` ke tiga situs (satu di Indonesia, satu di Asia, satu di Eropa/Amerika). Catat angkanya dan cocokkan dengan tabel *latency*.
4. Pasang Docker dan Go di laptop Anda.

Pertemuan depan

Kita turun ke *layer* paling dasar: bagaimana dua program bertukar data lewat *socket*, dan mengapa TCP, UDP, dan QUIC memilih jalan yang berbeda.

Terima kasih

Pertanyaan?