

Pertemuan 7 — Replikasi, Konsistensi, *Consensus*, dan Kubernetes

Tetap benar dan tetap hidup ketika sebagian mesin mati

Hendri Karisma

Semester Ganjil 2026/2027

Program Studi Teknik Informatika, STMIK Tazkia

Alur pertemuan hari ini

1. Waktu dan urutan: *clock drift*, Lamport clock
2. Replikasi: *leader-follower*, *multi-leader*, *leaderless*
3. Quorum: $W + R > N$
4. Model konsistensi: *strong*, *eventual*, *read-your-writes*
5. CAP dan PACELC
6. Consensus dengan Raft; CFT vs BFT dan *blockchain*
7. Partisi data dan *consistent hashing*
8. Container dan Kubernetes yang memulihkan diri
9. Praktikum: matikan *pod*, ukur waktu pulih
10. Rekapitulasi capaian A menjelang UTS

Sasaran pertemuan ini

Anda dapat menjelaskan bagaimana data disalin ke banyak mesin tanpa kehilangan kebenaran, memilih model konsistensi yang tepat, dan men-*deploy* layanan ke Kubernetes yang memulihkan dirinya sendiri ketika *pod* atau *node* mati.

Waktu dan urutan

Jam di tiap mesin berbohong sedikit-sedikit

- ▶ Jam komputer digerakkan kristal kuarsa yang kecepatannya sedikit berbeda antar mesin dan berubah dengan suhu. Selisih puluhan *ppm* berarti bisa bergeser hingga beberapa detik per hari (**clock drift**).
- ▶ **NTP** menyetel ulang jam secara berkala, tetapi ketelitiannya dibatasi *latency* jaringan; jam bahkan bisa **mundur** saat disetel.
- ▶ Akibatnya: dua *event* di dua mesin tidak bisa diurutkan dengan andal hanya dari *timestamp*-nya.

Contoh kehilangan data

Basis data memakai aturan *last-write-wins* berdasarkan *timestamp*. Server A (jamnya 50 ms lebih cepat) menulis `saldo=100`. Sesaat kemudian server B menulis `saldo=80`, tetapi *timestamp*-nya lebih kecil. Penulisan B yang **lebih baru** dibuang diam-diam.

Intinya

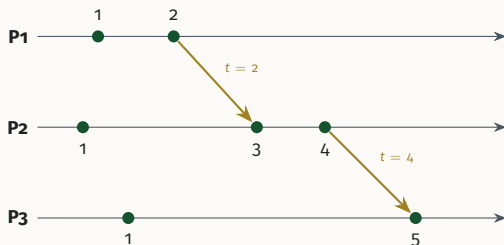
Yang sebenarnya kita butuhkan sering bukan “jam berapa”, melainkan “mana yang terjadi lebih dulu”.

Lamport clock: urutan tanpa jam fisik

Leslie Lamport (1978) mendefinisikan relasi ***happens-before*** ($\rightarrow$) dan penghitung logis C di tiap proses:

1. Sebelum setiap *event* lokal: $C \leftarrow C + 1$.
2. Saat mengirim pesan: sertakan C .
3. Saat menerima pesan bernilai t :
 $C \leftarrow \max(C, t) + 1$.

Jaminannya: bila $a \rightarrow b$ maka $C(a) < C(b)$.



P2 menerima $t = 2$ saat $C = 1$: $\max(1, 2) + 1 = 3$. P3 menerima $t = 4$ saat $C = 1$: $\max(1, 4) + 1 = 5$.

Intinya

Kebalikannya tidak berlaku: $C(a) < C(b)$ **tidak** berarti a terjadi sebelum b . Untuk mendeteksi dua *event* yang benar-benar *concurrent* dipakai *vector clock*, yang menyimpan satu penghitung per proses.

Replikasi

Mengapa menyalin data ke banyak mesin

Availability

Satu salinan mati, salinan lain tetap melayani.

Latency

Salinan diletakkan dekat pengguna: Jakarta, Singapura, Frankfurt.

Throughput baca

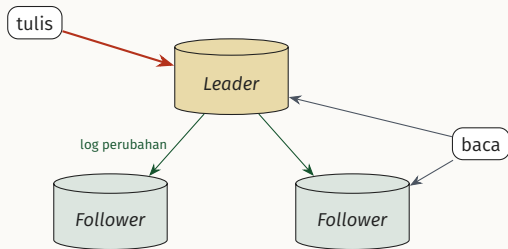
Banyak salinan berarti banyak mesin yang bisa menjawab permintaan baca.

Menyalin data yang **tidak pernah berubah** itu mudah. Semua kesulitan replikasi berasal dari satu pertanyaan:

Bagaimana menyebarkan **perubahan** ke semua salinan, ketika jaringan bisa terlambat, terputus, dan mesin bisa mati di tengah jalan?

Tiga jawaban utama: *leader-follower*, *multi-leader*, dan *leaderless*.

Leader-follower (single-leader)



- ▶ Semua penulisan masuk lewat satu *leader*; *follower* menerapkan log perubahan dengan urutan yang sama.
- ▶ **Synchronous**: *leader* menunggu *follower* mengonfirmasi. Aman, tetapi lambat dan macet bila *follower* mati.
- ▶ **Asynchronous**: *leader* langsung menjawab. Cepat, tetapi bila *leader* mati, penulisan terakhir yang belum tersalin bisa hilang.
- ▶ **Failover**: bila *leader* mati, satu *follower* dipromosikan. Bahaya: dua mesin sama-sama merasa *leader* (*split brain*).
- ▶ Dipakai PostgreSQL, MySQL, MongoDB, Kafka (per partisi).

Multi-leader dan leaderless

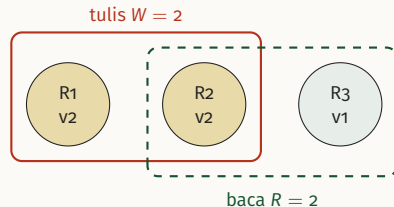
	Multi-leader	Leaderless
Gagasan	Beberapa <i>leader</i> , biasanya satu per <i>data center</i> ; saling menyalin	Tidak ada <i>leader</i> ; klien menulis dan membaca ke beberapa replika sekaligus
Kelebihan	Penulisan cepat di tiap wilayah; tahan bila satu <i>data center</i> terputus	Tidak ada titik tunggal; tetap jalan walau beberapa replika mati
Masalah utama	<i>Write conflict</i> : dua <i>leader</i> mengubah data yang sama	Replika bisa tertinggal; perlu <i>read repair</i> dan <i>anti-entropy</i>
Penyelesaian konflik	<i>Last-write-wins</i> (bisa kehilangan data), penggabungan oleh aplikasi, CRDT	<i>Quorum</i> , <i>version vector</i> , <i>last-write-wins</i>
Contoh	Aplikasi kalender/catatan <i>offline</i> , replikasi antar wilayah	Amazon Dynamo, Cassandra, Riak

Intinya

Semakin banyak pihak yang boleh menulis, semakin tinggi *availability*-nya, dan semakin sulit menjaga semua salinan tetap sama.

Quorum: $W + R > N$

- ▶ N = jumlah replika, W = berapa replika harus mengonfirmasi tulis, R = berapa replika ditanya saat baca.
- ▶ Bila $W + R > N$, himpunan replika yang ditulis dan yang dibaca pasti **beririsan** minimal satu. Pembaca pasti bertemu salinan terbaru.
- ▶ Konfigurasi umum: $N = 3$, $W = 2$, $R = 2$.
Tahan satu replika mati untuk tulis maupun baca.
- ▶ $W = 1$, $R = N$: tulis cepat, baca lambat.
 $W = N$, $R = 1$: kebalikannya.



Pembaca mendapat v2 dari R2 dan v1 dari R3, lalu memilih versi terbaru dan memperbaiki R3 (*read repair*).

Model konsistensi

Seberapa “baru” data yang Anda baca?

- ▶ **Strong consistency** (*linearizability*) — sistem bertingkah seolah hanya ada satu salinan. Begitu sebuah tulis selesai, semua pembaca berikutnya melihatnya. Wajib untuk saldo, stok terakhir, *lock*, pemilihan *leader*.
- ▶ **Eventual consistency** — bila penulisan berhenti, semua salinan **akhirnya** sama. Sebelum itu, pembaca bisa melihat data lama. Cukup untuk jumlah *like*, linimasa, rekomendasi.
- ▶ **Read-your-writes** — setidaknya **penulisnya sendiri** langsung melihat perubahannya. Orang lain boleh terlambat. Pengguna mengganti foto profil, lalu memuat ulang halaman: fotonya harus sudah berganti, walau teman-temannya baru melihat beberapa detik kemudian.

Intinya

Konsistensi bukan ya atau tidak, melainkan spektrum. Semakin kuat jaminannya, semakin banyak koordinasi antar mesin, dan semakin tinggi *latency* serta risiko tidak tersedia.

Satu cerita, tiga hasil

Ani ——— tulis harga=12rb —————> waktu

Ani ————— baca —————>

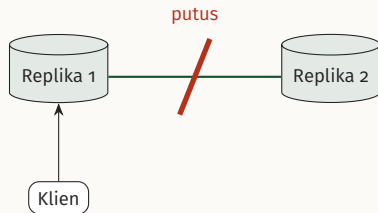
Budi ————— baca —————>

Model	Ani membaca	Budi membaca
<i>Strong</i>	12rb (pasti)	12rb (pasti)
<i>Read-your-writes</i>	12rb (pasti)	10rb atau 12rb
<i>Eventual</i>	10rb atau 12rb	10rb atau 12rb, lama-lama 12rb

Harga semula 10rb. Untuk katalog belanja, *read-your-writes* bagi penjual sering sudah cukup. Untuk *checkout*, harga yang dikunci harus *strong*.

CAP: ketika jaringan terbelah

- ▶ Dugaan Eric Brewer (2000), dibuktikan Gilbert dan Lynch (2002).
- ▶ Saat terjadi **partition** (P) — jaringan memisahkan replika — sistem harus memilih:
 - **C**: tolak permintaan agar tidak memberi data usang, atau
 - **A**: tetap menjawab, walau datanya mungkin usang.
- ▶ Salah kaprah: “pilih dua dari tiga”. Partisi bukan pilihan; ia pasti terjadi. Pilihannya hanya C atau A **selama** partisi berlangsung.



Replika 1 menerima tulis. Apakah ia menjawab “berhasil” (A) atau menolak sampai bisa bicara dengan Replika 2 (C)?

PACELC: pertanyaan yang lebih jujur

Daniel Abadi (2012) melengkapi CAP:

Intinya

Bila ada **P**artisi, pilih **A** atau **C**; **E**lse (keadaan normal), pilih **L**atency atau **C**onsistency.

Sebagian besar waktu jaringan baik-baik saja. Yang terasa setiap hari adalah pilihan kedua: menunggu konfirmasi replika lain (konsisten, lebih lambat) atau langsung menjawab (cepat, mungkin usang).

Sistem	Kelas	Arti
Dynamo, Cassandra (bawaan)	PA/EL	Utamakan tersedia dan cepat
etcd, ZooKeeper, Spanner	PC/EC	Utamakan konsisten, bayar dengan <i>latency</i>

Banyak basis data bisa disetel per permintaan, misalnya Cassandra dengan level konsistensi ONE atau QUORUM.

Consensus dengan Raft

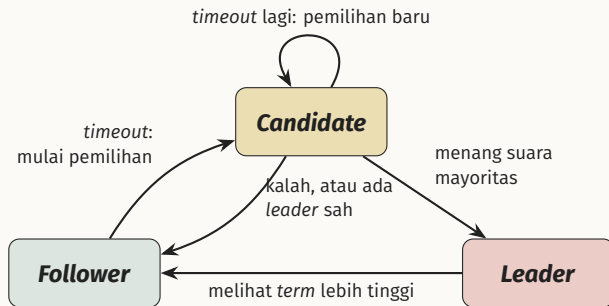
Masalah *consensus*

- ▶ Beberapa mesin harus **sepakat** atas satu nilai atau satu urutan perintah, walau sebagian mati dan pesan terlambat.
- ▶ Dipakai untuk: memilih *leader*, menyimpan konfigurasi klaster, *distributed lock*, memastikan semua replika menjalankan log yang sama.
- ▶ Hasil teoretis FLP (Fischer, Lynch, Paterson, 1985): pada sistem yang sepenuhnya asinkron, tidak ada algoritma deterministik yang **selalu** mencapai kesepakatan bila satu proses saja bisa mati. Algoritma praktis memakai *timeout* untuk keluar dari kebuntuan ini.
- ▶ Paxos (Lamport) terkenal sulit dipahami. **Raft** (Ongaro dan Ousterhout, 2014) dirancang dengan tujuan utama **mudah dipahami**.

Intinya

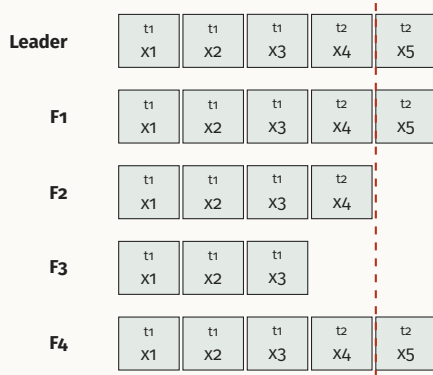
Raft ada di dalam etcd, yang menyimpan seluruh *state* Kubernetes. Setiap kali Anda menjalankan `kubectl apply`, Raft bekerja di belakangnya.

Raft: tiga peran



- ▶ Waktu dibagi menjadi **term** bernomor 1, 2, 3, ... Setiap *term* dimulai dengan pemilihan, paling banyak satu *leader* per *term*.
- ▶ *Leader* mengirim *heartbeat* berkala. *Follower* yang lama tidak mendengar *heartbeat* menjadi *candidate*.
- ▶ *Election timeout* diacak (makalah aslinya memakai rentang sekitar 150–300 ms) agar tidak semua *follower* mencalonkan diri bersamaan.
- ▶ Tiap *node* hanya memberi satu suara per *term*.

Raft: replikasi log



commit: indeks 4 sudah ada di 4 dari 5 node

1. Klien mengirim perintah ke *leader*.
 2. *Leader* menambahkannya ke lognya, lalu mengirim `AppendEntries` ke semua *follower*.
 3. Setelah **mayoritas** menyimpannya, entri dianggap *committed* dan diterapkan ke *state machine*.
 4. *Follower* yang tertinggal (F2, F3) dikejar *leader* sampai lognya sama.
- Kotak berisi *term* (t1, t2) dan perintah (x1-x5).

Mengapa selalu jumlah ganjil?

Jumlah <i>node</i>	Mayoritas	Boleh mati
1	1	0
3	2	1
4	3	1
5	3	2
7	4	3

Secara umum $2f + 1$ *node* tahan f kegagalan. Empat *node* tidak lebih tahan daripada tiga, tetapi lebih mahal.

- ▶ Mayoritas menjamin dua kelompok tidak bisa sama-sama memilih *leader* pada *term* yang sama: dua mayoritas pasti beririsan.
- ▶ Bila klaster terbelah 2 lawan 3, hanya sisi 3 yang bisa melanjutkan. Sisi 2 berhenti menerima tulis: pilihan **C** dalam CAP.
- ▶ Karena itu etcd di klaster Kubernetes produksi umumnya 3 atau 5 *node*.

Crash fault vs Byzantine fault

Crash fault tolerance (CFT)

- ▶ Asumsi: *node* bisa **mati** atau lambat, tetapi tidak pernah berbohong.
- ▶ Raft dan Paxos: $2f + 1$ **node** untuk mentoleransi f *node* mati (3 *node* tahan 1 mati, 5 tahan 2).
- ▶ Cocok di dalam satu organisasi: semua mesin milik kita, kita percaya isinya.
- ▶ Contoh: etcd di Kubernetes, KRaft di Kafka.

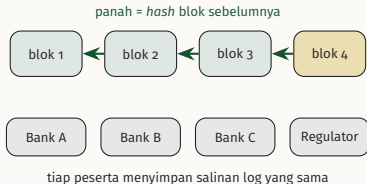
Byzantine fault tolerance (BFT)

- ▶ Asumsi: sebagian *node* bisa **jahat**: berbohong, mengirim pesan berbeda ke *node* berbeda, bersekongkol.
- ▶ PBFT (Castro dan Liskov, OSDI 1999): $3f + 1$ **node** untuk mentoleransi f *node* jahat (4 *node* tahan 1, 7 tahan 2).
- ▶ Harganya: lebih banyak *node* dan jauh lebih banyak pesan (tiap *node* bicara ke semua *node*, orde n^2).
- ▶ Dibutuhkan ketika pesertanya **tidak saling percaya**.

Intinya

Mengapa $3f + 1$? Dari n *node*, f bisa diam dan f lainnya bisa berbohong. Agar *node* jujur tetap menang suara, harus tersisa lebih dari f *node* jujur di antara yang menjawab: $n - f > 2f$, jadi $n \geq 3f + 1$.

Blockchain: replicated log di antara pihak yang tidak saling percaya



Sama seperti log Raft: urutan entri disepakati, lalu disalin ke semua *node*. Bedanya, tiap blok mengunci *hash* blok sebelumnya, sehingga riwayat lama tidak bisa diubah diam-diam.

	Permissioned	Public
Peserta	Dikenal, diundang (konsorsium bank, rantai pasok)	Siapa pun boleh ikut
<i>Consensus</i>	CFT atau BFT di antara sedikit <i>node</i>	Nakamoto <i>consensus</i> : PoW atau PoS
Contoh	Hyperledger Fabric: <i>ordering service</i> Raft (CFT) atau BFT sejak versi 3.x; Hyperledger Besu dengan QBFT	Bitcoin (PoW), Ethereum (PoS)
<i>Finality</i>	Deterministik: sekali disepakati, final	Probabilistik: makin dalam bloknya, makin kecil peluang dibatalkan

Hyperledger Fabric dan Besu adalah proyek *open source* (kini di bawah LF Decentralized Trust).

Finality dan kapan *blockchain* masuk akal

Dua jenis *finality*

- ▶ **Deterministik** (Raft, PBFT, QBFT): begitu entri disepakati *quorum*, ia tidak akan pernah dibatalkan. Cocok untuk transaksi keuangan.
- ▶ **Probabilistik** (PoW Nakamoto): beberapa cabang bisa bersaing; cabang terpanjang yang menang. Penerima menunggu beberapa blok sebelum menganggap transaksi aman.
- ▶ PoS modern seperti Ethereum menambahkan *checkpoint* yang menjadi final setelah disetujui mayoritas *stake*.

Pertanyaan sebelum memilih *blockchain*

1. Apakah ada beberapa pihak yang **tidak saling percaya** tetapi harus berbagi satu catatan?
2. Tidak ada satu pihak yang boleh atau mau menjadi pemegang data pusat?
3. Riwayat harus bisa dibuktikan tidak diubah?

Bila semua jawabannya “tidak”, basis data biasa dengan replikasi Raft lebih cepat, lebih murah, dan lebih mudah dioperasikan.

Intinya

Di sini kita melihat *blockchain* dari sisi *consensus*. Kapan *ledger* atau *blockchain* layak dipilih sebagai *storage*, dibandingkan basis data biasa, dibahas di Data Engineering.

Partisi data dan *consistent hashing*

Partisi: ketika data tidak muat di satu mesin

Partisi berdasarkan rentang

Key A–F di mesin 1, G–M di mesin 2, dan seterusnya.

- ▶ Kueri rentang efisien (“semua pesanan bulan Oktober”)
- ▶ Rawan *hot spot*: semua data hari ini jatuh ke satu partisi

Partisi berdasarkan *hash*

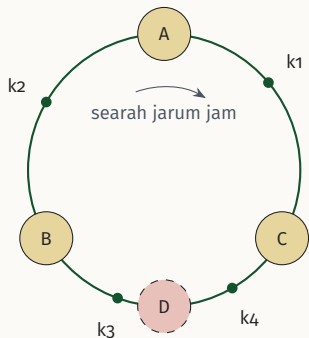
Mesin = $\text{hash}(\text{key}) \bmod N$.

- ▶ Beban merata
- ▶ Kueri rentang harus bertanya ke semua partisi
- ▶ Masalah besar: bila N berubah, hampir **semua** *key* pindah mesin

Intinya

Partisi (membagi data) dan replikasi (menyalin data) hampir selalu dipakai bersama: tiap partisi punya beberapa replika, dan tiap mesin menyimpan replika dari beberapa partisi.

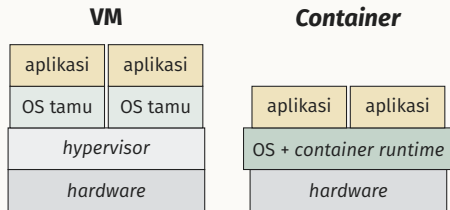
Consistent hashing



- ▶ Kunci dan mesin sama-sama di-hash ke sebuah lingkaran. Kunci disimpan di mesin pertama yang ditemui searah jarum jam.
- ▶ $k1 \rightarrow A$, $k2 \rightarrow B$, $k4 \rightarrow C$, dan $k3$ semula $\rightarrow C$.
- ▶ Tambah mesin D: hanya $k3$ yang pindah ke D. Rata-rata hanya sekitar $1/N$ key yang berpindah.
- ▶ **Virtual node**: tiap mesin ditempatkan di banyak titik pada lingkaran agar bebannya lebih merata.
- ▶ Diperkenalkan Karger dkk. (1997); dipakai Amazon Dynamo, Cassandra, dan banyak *cache* terdistribusi.

***Container* dan Kubernetes**

Container bukan mesin virtual



- ▶ VM membawa sistem operasi sendiri: berat, mulai dalam hitungan puluhan detik.
- ▶ *Container* berbagi *kernel* OS, diisolasi dengan *namespace* dan *cgroup* Linux: ringan, mulai dalam hitungan detik atau kurang.
- ▶ *Image* berisi aplikasi dan semua *library*-nya, sehingga berjalan sama di laptop dan di klaster.
- ▶ Isolasi *container* lebih tipis daripada VM; untuk *multi-tenant* yang tidak saling percaya, VM tetap lebih aman.

Dockerfile multi-stage untuk layanan Go

```
# tahap 1: kompilasi
FROM golang:1.22 AS build
WORKDIR /src
COPY go.mod go.sum ./
RUN go mod download
COPY . .
RUN CGO_ENABLED=0 go build -o /katalog .

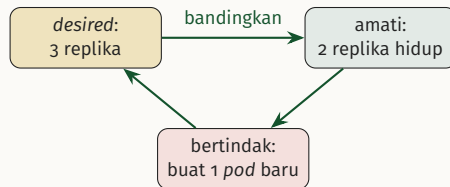
# tahap 2: image kecil untuk produksi
FROM gcr.io/distroless/static
COPY --from=build /katalog /katalog
EXPOSE 8080
USER nonroot
ENTRYPOINT ["/katalog"]
```

- ▶ Tahap pertama membawa seluruh *toolchain* Go; tahap kedua hanya membawa *binary*-nya. *Image* akhir jauh lebih kecil.
- ▶ *Image* kecil berarti lebih cepat diunduh saat *pod* dijadwal ulang, sehingga pemulihan lebih cepat.
- ▶ *Image* tanpa *shell* dan berjalan sebagai nonroot juga memperkecil *attack surface*.
- ▶ Bangun dan muat ke kluster kind:

```
docker build -t katalog:v1 .
kind load docker-image katalog:v1
```

Kubernetes: *desired state* dan *control loop*

- ▶ **Pod** — satu atau beberapa *container* yang selalu dijadwal bersama; unit terkecil yang dijalankan.
- ▶ **Deployment** — menyatakan “jalankan n replika dari *pod* ini”; mengurus *rolling update*.
- ▶ **Service** — alamat dan nama DNS tetap yang menyebar lalu lintas ke *pod* yang **siap**, walau *pod*-nya datang dan pergi.
- ▶ **Controller** — terus membandingkan *actual state* dengan *desired state* di etcd, lalu bertindak untuk menyamakannya.



Kubernetes tidak menjalankan perintah “buat *pod*”; ia terus mengejar *desired state* yang Anda deklarasikan. Inilah sumber kemampuan *self-healing*.

Deployment dengan probe

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: katalog
spec:
  replicas: 3
  selector:
    matchLabels: {app: katalog}
  template:
    metadata:
      labels: {app: katalog}
    spec:
      containers:
      - name: katalog
        image: katalog:v1
        ports: [{containerPort: 8080}]
        readinessProbe:
          httpGet: {path: /healthz, port: 8080}
          periodSeconds: 2
        livenessProbe:
          httpGet: {path: /healthz, port: 8080}
          initialDelaySeconds: 5
        resources:
          requests: {cpu: 100m, memory: 64Mi}
```

- ▶ **readinessProbe** — “sudah siap menerima lalu lintas?” Bila gagal, *pod* dikeluarkan dari *Service*, tetapi tidak dimatikan.
- ▶ **livenessProbe** — “masih hidup?” Bila gagal berulang kali, *container* dimulai ulang.
- ▶ **startupProbe** — untuk aplikasi yang lama menyala, agar tidak dibunuh *liveness* sebelum sempat siap.
- ▶ `resources.requests` dipakai *scheduler* untuk menempatkan *pod* dan oleh HPA untuk menghitung persentase CPU.

Jebakan umum

Liveness probe yang memeriksa basis data: saat basis data lambat, semua *pod* dimulai ulang bersamaan dan masalah menjadi lebih parah.

Service dan HorizontalPodAutoscaler

```
apiVersion: v1
kind: Service
metadata:
  name: katalog
spec:
  selector: {app: katalog}
  ports:
    - port: 80
      targetPort: 8080
```

Di dalam klaster, layanan lain cukup memanggil `http://katalog`.

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: katalog
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: katalog
  minReplicas: 3
  maxReplicas: 10
  metrics:
    - type: Resource
      resource:
        name: cpu
        target: {type: Utilization, averageUtilization: 70}
```

HPA menambah replika bila rata-rata CPU melebihi 70% dari requests. Butuh metrics-server terpasang di klaster.

Apa yang terjadi ketika *pod* atau *node* mati

Kejadian	Reaksi Kubernetes	Yang dirasakan pengguna
Proses di <i>container</i> crash	<i>Kubelet</i> memulai ulang <i>container</i> ; jeda bertambah bila berulang (<i>CrashLoopBackOff</i>)	Permintaan ke <i>pod</i> itu gagal sesaat; replika lain tetap melayani
<i>Pod</i> dihapus	<i>Deployment</i> langsung membuat pengganti di <i>node</i> mana pun	Hampir tidak terasa bila ada replika lain dan <i>readiness</i> benar
<i>Node</i> mati	Setelah <i>node</i> dinyatakan tidak sehat dan masa tunggu habis, <i>pod</i> -nya dijadwal ulang	Bisa butuh beberapa menit; karena itu replika harus tersebar di beberapa <i>node</i>
Beban naik	HPA menambah replika	<i>Latency</i> naik sebentar sampai replika baru siap

Intinya

Self-healing bukan sihir: ia bekerja hanya bila aplikasi Anda *stateless* (atau *state*-nya disimpan di tempat yang direplikasi), punya lebih dari satu replika, dan punya *probe* yang jujur.

Praktikum

Praktikum: matikan *pod*, ukur waktu pulih

```
# 1. klaster lokal dan deploy
kind create cluster --name sister
kind load docker-image katalog:v1 --name sister
kubectl apply -f deployment.yaml -f service.yaml
kubectl get pods -l app=katalog -o wide

# 2. beban terus-menerus DARI DALAM klaster
kubectl run beban --rm -it --image=curlimages/curl \
  -- sh -c 'while true; do
    echo "$(date +%T) $(curl -s -o /dev/null \
      -w %{http_code} http://katalog/healthz)"
    sleep 0.2; done' | tee beban.log

# 3. di terminal lain: matikan satu pod
kubectl delete $(kubectl get pod -l app=katalog \
  -o name | head -1)
kubectl get pods -l app=katalog -w

# 4. hitung permintaan yang gagal
grep -vc ' 200' beban.log
```

1. Ulangi langkah 3 dengan replicas: 1, lalu replicas: 3. Bandingkan jumlah permintaan gagal.
2. Hapus readinessProbe, ulangi. Apa yang berubah?
3. Catat waktu dari *pod* dihapus sampai penggantinya Ready (lihat `kubectl get events --sort-by=.lastTimestamp`).
4. Uji HPA dengan beban tinggi (hey atau k6); amati `kubectl get hpa -w`.

Mengapa beban dari dalam klaster?

`kubectl port-forward svc/...` hanya terhubung ke **satu** *pod*. Bila *pod* itu yang dimatikan, *port-forward* putus dan pengukuran menjadi menyedihkan.

Rekapitulasi capaian A

Capaian A dalam satu halaman

P	Topik	Yang harus bisa Anda tunjukkan di UTS
1	Mengapa terdistribusi	Alasan memecah sistem, ongkosnya, dan <i>fallacy</i> yang Anda waspadai
2	TCP, UDP, QUIC	Pilihan <i>transport</i> tiap jalur komunikasi dan alasannya
3	HTTP/1.1–3, TLS	Layanan diakses lewat HTTP; jelaskan versi dan pengamanannya
4	REST, gRPC, GraphQL, RMI	Implementasi REST dan gRPC, pengukuran P_{50}/P_{99} dan <i>throughput</i>
5	Arsitektur, <i>messaging</i>	Diagram arsitektur, jalur <i>asynchronous</i> (<i>queue</i> atau <i>actor</i>)
6	<i>Concurrency</i>	Model <i>concurrency</i> tiap layanan; bebas <i>race</i> (bukti uji)
7	Replikasi, Kubernetes	Layanan berjalan dengan replika; uji kegagalan dan waktu pulih

Intinya

Benang merahnya: **setiap panah jaringan bisa terlambat, gagal, atau terulang**. Sistem yang baik dirancang dengan anggapan itu sejak awal, dan membuktikannya dengan pengukuran.

UTS: daftar periksa “Satu sistem, tiga cara bicara”

Sistem

- ☐ Minimal tiga layanan dengan tanggung jawab jelas
- ☐ Satu jalur REST, satu jalur gRPC, satu jalur *asynchronous*
- ☐ *Timeout* dan *retry* pada panggilan sinkron; operasi tulis yang di-*retry* bersifat *idempotent*
- ☐ Berjalan dengan docker compose atau Kubernetes, minimal dua replika pada satu layanan

Bukti

- ☐ Tabel P_{50} , P_{99} , *throughput* REST vs gRPC (lengkap dengan alat dan skenario uji)
- ☐ Uji kegagalan: apa yang dimatikan, berapa permintaan gagal, berapa lama pulih
- ☐ Laporan: arsitektur, alasan desain, keterbatasan
- ☐ Demo 10 menit dan tanya jawab; semua anggota mampu menjelaskan kode

Tugas untuk pertemuan berikutnya

1. Selesaikan praktikum: laporan jumlah permintaan gagal dan waktu pulih untuk replicas: 1 dan replicas: 3, dengan dan tanpa *readiness probe*.
2. Mainkan visualisasi Raft di raft.github.io: matikan *leader*, amati pemilihan baru, lalu tulis satu paragraf tentang yang Anda lihat.
3. Baca Kleppmann, *Designing Data-Intensive Applications*, Bab 5 (replikasi) dan Bab 9 (konsistensi dan *consensus*).
4. Rampungkan tugas besar UTS sesuai daftar periksa.

Pertemuan depan

UTS: presentasi dan demo tugas besar “Satu sistem, tiga cara bicara”. Setelah UTS kita berpindah ke capaian B: *High Performance Computing*.

Terima kasih

Pertanyaan?