CS 417 – DISTRIBUTED SYSTEMS

Week 13: Clusters

Scheduling, availability, and service design

Paul Krzyzanowski

© 2026 Paul Krzyzanowski. No part of this content may be reproduced or reposted in whole or in part in any manner without the permission of the copyright owner.

Roadmap

- **Foundations**: what a cluster is and how nodes connect
- **Scheduling and placement**: Borg, Kubernetes
- **High availability**: membership, heartbeats, quorum, failover
- **Load balancing** and **health monitoring**
- Design consequences

Cluster foundations

What is a cluster?

A group of autonomous servers,
built from commodity hardware, that work as one system

- Presents a **single system image**:
  - Clients see one service, not a rack of machines
- **Scales** by adding nodes rather than upgrading one large machine
- **Hides** node additions, removals, and failures from clients
- ***Assumes failure and handles it in software***

Kinds of clusters

HA **HA clusters** · Mask node failures from clients

HPC **HPC clusters** · Run one tightly coupled computation across many nodes

LB **Load-balancing** · Spread stateless requests across identical workers

SC **Storage clusters** · Pool disks from many machines behind one namespace

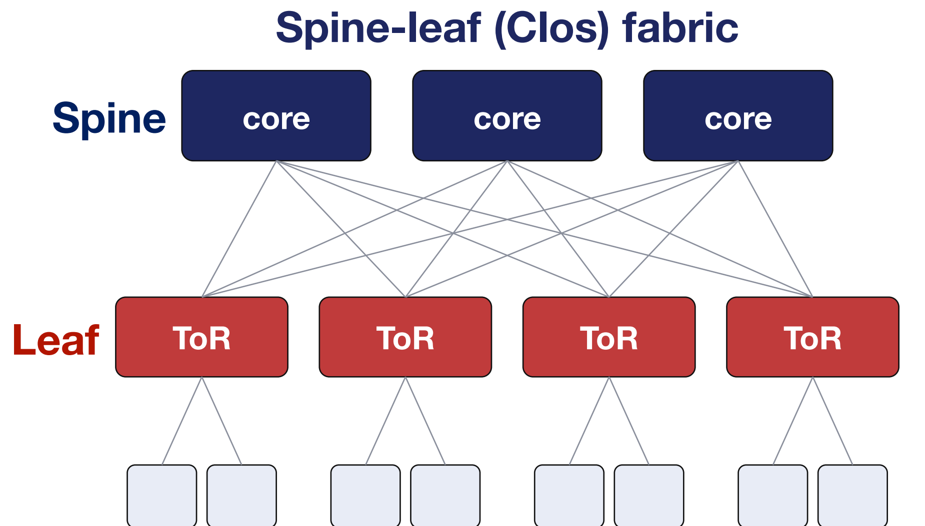
Scheduling · Pack many unrelated jobs onto a shared fleet

Reliability from software

- **At scale, something is always broken**
  - Disks, NICs, power supplies, switches
- **We rely on software & redundancy**
  - Detects failures, re-routes work, and replays or replicates state
  - Replication provides both availability and throughput
- **Prefer commodity servers and networks (usually)**
  - Cost far less than engineered reliable hardware
  - Price per unit of useful work matters more than peak speed of any one node
    - Amount of work per $ (and per watt)

Cluster network layout: Spine-leaf topology

- **Racks of servers (~20-40)**
  - Each with a top-of-rack (ToR) switch
- **Spine layer**
  - ToR switches uplink to core switches
- Any two servers can talk at near full link speed
- Server-to-server (east-west) traffic dominates



High-speed communication

- **TCP adds OS overhead**
  - Interrupts, checksums, retransmits, copies
  - **NIC offloads** reduce packet-processing CPU overhead
- **RDMA** – Remote Direct Memory Access
  - Lets one node read or write remote memory with no CPU involvement
  - InfiniBand and RDMA over Ethernet (RoCE)
    carry GPU training traffic at 400 to 800 Gbps
  - Within an NVIDIA GPU server, **NVLink** and **NVSwitch** give direct GPU-to-GPU paths

Used where microseconds and CPU cycles change the economics of the workload

Datacenter scheduling

The scheduling problem

- A data center runs hundreds of distinct applications
  - Batch jobs, long-running services, ad-hoc queries, ML training
- The scheduler decides which machine runs each task

Goal: keep utilization high without missing service level objectives

What a scheduler must do

- **Track** available CPU, memory, disk, and GPU on every node
- **Place** each task while respecting user-supplied constraints
- **Enforce** priorities, quotas, and fair sharing across teams
- **Detect** task failures and reschedule surviving work
- **Support mixed workloads** without starvation or oscillation

Google Borg

CS 417 © 2026 Paul Krzyzanowski

Borg overview

- Cluster manager behind every Google service since the mid 2000s
  - Paper published in 2015
  - Became the blueprint for Kubernetes
- One **Borg cell** manages tens of thousands of machines
- A data center runs several cells
  - **Jobs** submit to a cell

Large-scale cluster management at Google with Borg

Abhishek Verma[†] Luis Pedrosa[‡] Madhukar Korupolu
David Oppenheimer Eric Tune John Wilkes
Google Inc.

Abstract

Google's Borg system is a cluster manager that runs hundreds of thousands of jobs, from many thousands of different applications, across a number of clusters each with up to tens of thousands of machines.

It achieves high utilization by combining admission control, efficient task-packing, over-commitment, and machine sharing with process-level performance isolation. It supports high-availability applications with runtime features that minimize fault-recovery time, and scheduling policies that reduce the probability of correlated failures. Borg simplifies life for its users by offering a declarative job specification language, name service integration, real-time job monitoring, and tools to analyze and simulate system behavior.

We present a summary of the Borg system architecture and features, important design decisions, a quantitative analysis of some of its policy decisions, and a qualitative examination of lessons learned from a decade of operational experience with it.

1. Introduction

The cluster management system we internally call Borg admits, schedules, starts, restarts, and monitors the full range of applications that Google runs. This paper explains how.

Borg provides three main benefits: it (1) hides the details of resource management and failure handling so its users can focus on application development instead; (2) operates with very high reliability and availability, and supports applications that do the same; and (3) lets us run workloads across tens of thousands of machines effectively. Borg is not the first system to address these issues, but it's one of the few operating at this scale, with this degree of resiliency and completeness. This paper is organized around these topics, concluding with a set of qualitative observations we have made from operating Borg in production for more than a decade.

[†] Work done while author was at Google.
[‡] Currently at University of Southern California.

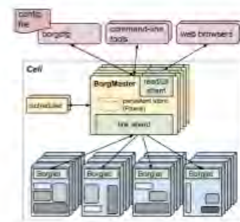


Figure 1: The high-level architecture of Borg. *Only a tiny fraction of the thousands of worker nodes are shown.*

2. The user perspective

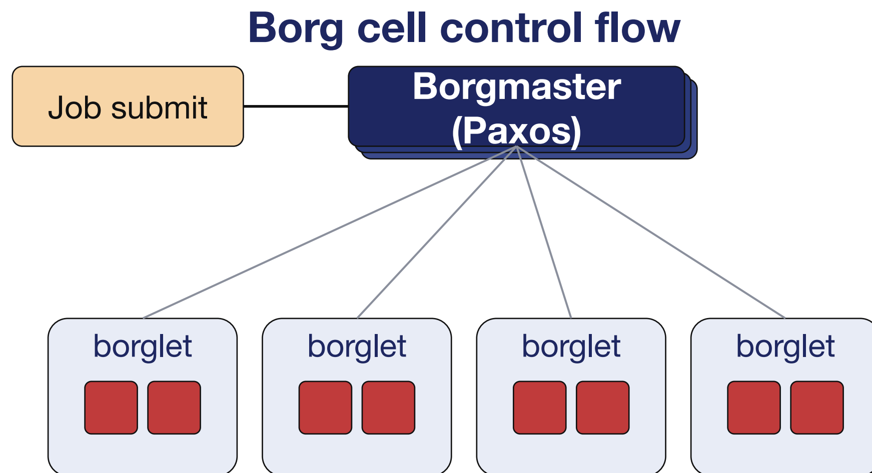
Borg's users are Google developers and system administrators (site reliability engineers or SREs) that run Google's applications and services. Users submit their work to Borg in the form of *jobs*, each of which consists of one or more *tasks* that all run the same program (binary). Each job runs in one Borg *cell*, a set of machines that are managed as a unit. The remainder of this section describes the main features exposed in the user view of Borg.

2.1 The workload

Borg cells run a heterogeneous workload with two main parts. The first is long-running services that should "never" go down, and handle short-lived latency-sensitive requests (a few µs to a few hundred ms). Such services are used for end-user-facing products such as Gmail, Google Docs, and web search, and for internal infrastructure services (e.g., BigTable). The second is batch jobs that take from a few seconds to a few days to complete; these are much less sensitive to short-term performance fluctuations. The workload mix varies across cells, which run different mixes of applications depending on their major tenants (e.g., some cells are quite batch-intensive), and also varies over time: batch jobs

Borg architecture

- **Borgmaster:** replicated master, holds cell state
- **Borglet:** agent on every machine, runs tasks
- **Scheduler:** assigns pending tasks to machines
- State persisted in a Paxos-replicated store



Borgmaster is replicated:
Each machine runs a borglet managing tasks

Jobs and tasks

Job = set of identical tasks, declared in a config file

- Each **task** is a single process or process group inside a container
- Jobs have:
  - Priorities, quotas, resource requests, and placement constraints
- A job can span thousands of tasks across many machines
- **Allocs** reserve resources so tasks can hold them across restarts

Priorities and quota

- **Priority bands**: monitoring, production, batch, best-effort
  - Higher-priority tasks preempt lower-priority tasks when resources run out
- **Quota** is a team's allocation of resources at each priority level
  - Unused production quota can be lent or sold at lower priority
  - **This is how Google fits batch into the slack left by production services**

Scheduling in Borg

Two phases per task

1. **Feasibility pass:** *which machines can host this task?*
2. **Scoring pass:** *which feasible machine is the best fit?*
   - Factors include machine type, existing tasks, locality, and spreading
   - Tasks are spread across failure domains: racks, power, and zones

Scheduler runs continuously:

Re-evaluates on failures and new submissions

Isolation in Borg

- Each task runs in a Linux container
  - Packages its code, libraries, and config on a shared host kernel
- **cgroups** limit CPU, memory, and I/O per task
- **namespaces** give each task its own PID, mount, and network view
- Memory
  - A task that exceeds its memory limit is killed by the kernel and restarted
- CPU
  - Shared, but reservations prioritize production work

Kubernetes

From Borg to Kubernetes

- Borg is proprietary
  - **Kubernetes** is the open-source system most directly influenced by it
- Started inside Google around 2014 as an open-source project
- **Different tradeoffs:** extensibility, portability, lighter config
- Same structure: central control plane, per-node agents, declarative specs, reconciliation loops

Goal: orchestration for anyone running containers at scale, not just Google

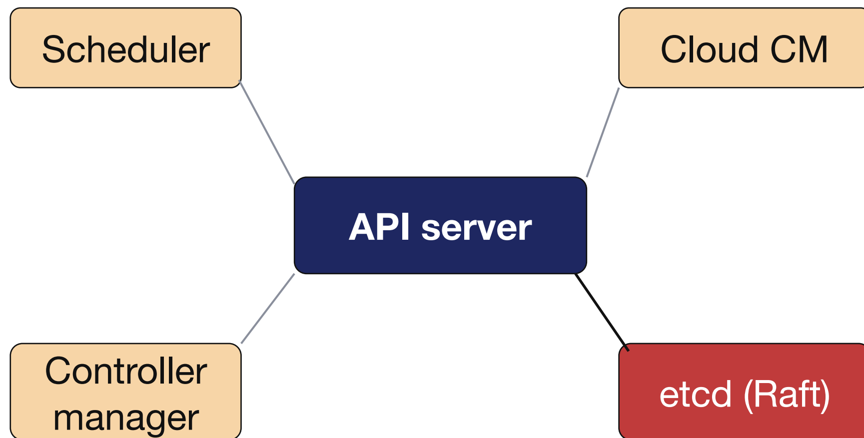
Design principles

- **Declarative configuration**: describe the desired state, not the steps
- **Control loops** reconcile actual state toward desired state, continuously
- **Everything is an API object**: pods, services, volumes, secrets
- Decoupled components **communicate only through the API server**
- **Extensible**: custom resources and controllers add new object types

```
kind: DatabaseCluster
spec:
    replicas: 3
    version: "16"
    storage: "500Gi"
```

Control plane

- **API server:** validates and persists every change
- **etcd:** Raft-replicated key-value store
- **Scheduler:** assigns pending pods to nodes
- **Controllers:** reconcile desired vs actual state
- **Cloud CM:** integrates with the cloud provider



All state flows through the API server
etcd is the store

Worker nodes

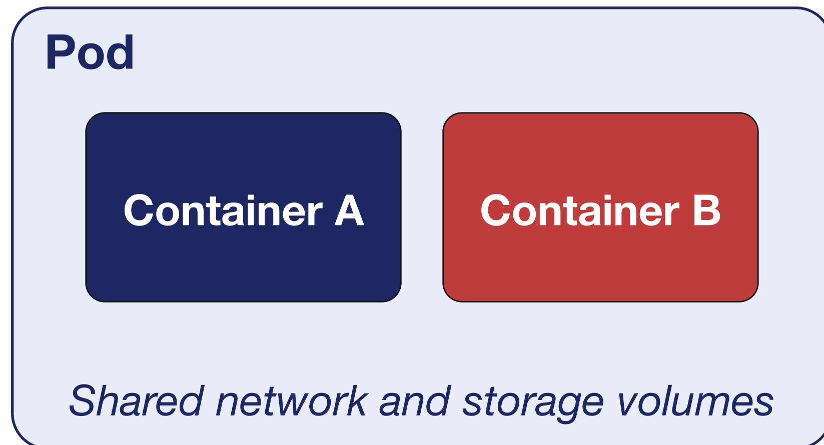
- **kubelet:** local agent that pulls pod specs and reports pod status
- **kube-proxy:** maintains network rules for services on the node
- **Container runtime:** *containerd* or *CRI-O* actually runs the containers

Node resources are reported back to the scheduler through the API server

Pods

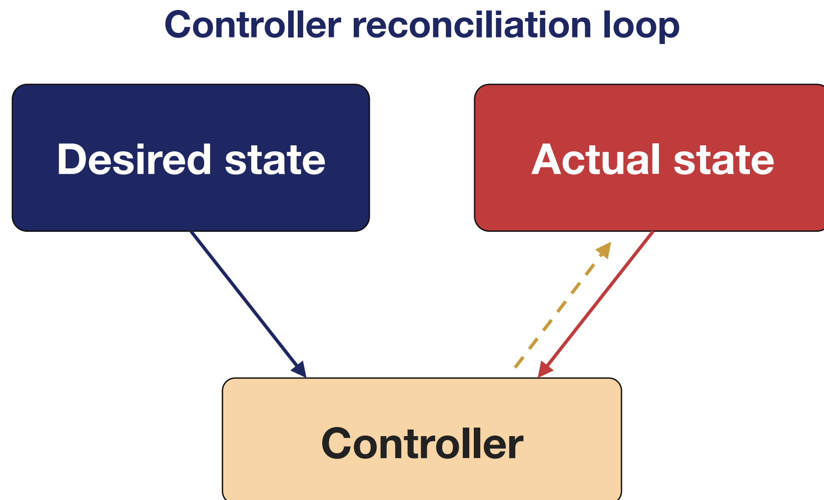
Pod = smallest deployable unit in Kubernetes

- One or more containers sharing a network namespace and volumes
- **Ephemeral by design**:
  - Pods get replaced, not repaired
- A **controller** (for example, a Deployment) manages a set of pods
- The **scheduler** assigns a pod as a single atomic unit



Controllers and reconciliation

- A **controller** watches the API for objects of a given kind
- Compares *desired* state to *observed* state
- Takes actions to close the gap:
  - create, update, delete
- **Idempotent**: the same observation produces the same action



The scheduler

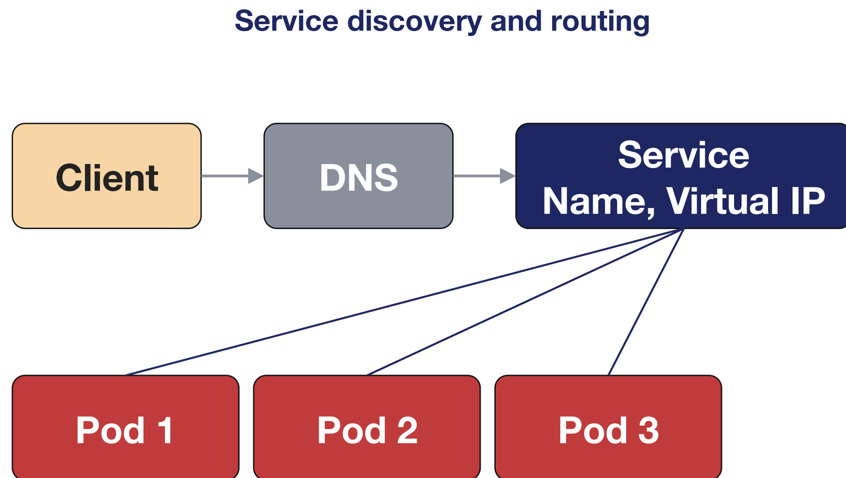
Watches for pods with no assigned node

1. **Filters nodes that cannot run the pod**
2. **Scores remaining nodes:** *by fit, spreading, and placement preferences*

- Writes the chosen node back to the API server
- The kubelet on that node starts the pod

Services and discovery

- Pods are temporary
  - Their IPs change
- A **Service** gives matching pods a stable virtual IP and DNS name
- Kubernetes keeps the backend pod list current.
- **kube-proxy** routes *Service* traffic to one of the current pods
- External traffic enters through *Ingress* or a *LoadBalancer* Service.



Borg and Kubernetes compared

Borg	Kubernetes
Internal to Google	Open source, runs anywhere
One organization and operational model	Multi-tenant, extensible platform
Central control plane	Central control plane
Per-node agent	Per-node agent
Internal config language	YAML, published REST API
Opinionated, closed	Extensible ecosystem

High availability

CS 417 © 2026 Paul Krzyzanowski

Defining High Availability (HA)

- **Availability:** the fraction of time a service is usable
- **High availability:**
  System is designed to keep serving despite expected failures
  - HA may still allow brief disruption during detection, failover, or recovery
- **Fault tolerance** means the system continues operating correctly after a fault, often with stricter guarantees
- **Disaster recovery** handles large-scale loss, such as a region, data center, or storage system failure
- Targets are expressed in nines:
  - Three nines = 99.9% = ~ 8.8 hours of downtime per year
  - Four nines = 99.99% = ~52 minutes
  - Five nines = 99.999% = ~ 5.3 minutes

Heartbeats

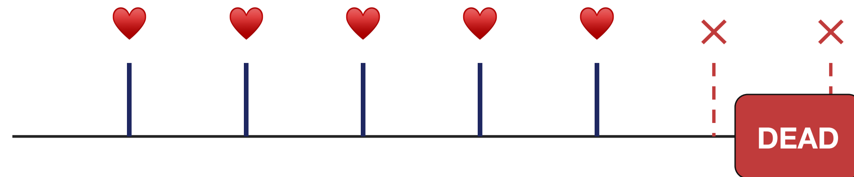
Periodic probe from a controller to each node, or between peers

- Silence for more than a timeout marks the node as dead
- **Timeout too short:** false positives on network hiccups
- **Timeout too long:** slow failover

Heartbeats work together with leases

– Leases make failover safer

Heartbeats and timeout

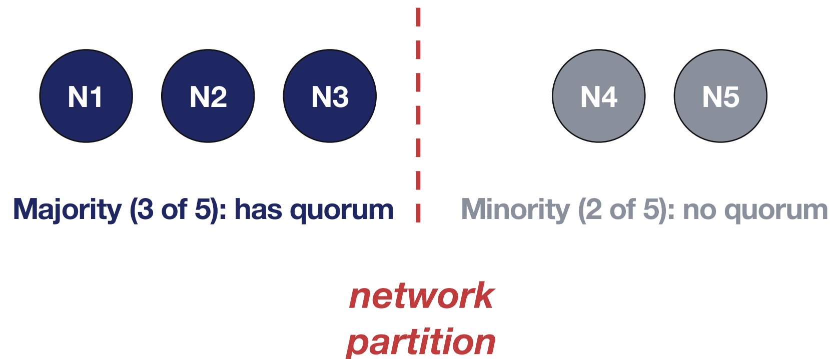


After N missed beats within a window, the node is declared dead

Split-brain and quorum

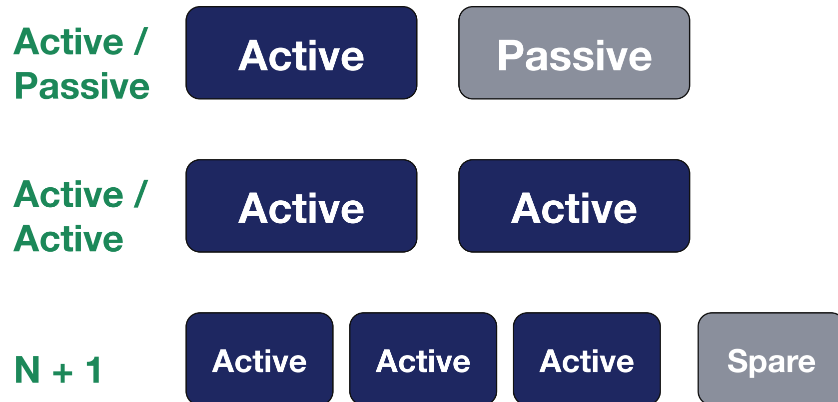
A **network partition** can leave each part thinking the other is dead

- Without coordination, both segments act as primary: **split brain**
- Solution: require a **quorum** (majority) of nodes for any authoritative action
  - Only one side can hold a majority; the other must step down
  - The principle underlies Paxos and Raft consensus



Failover configurations

- **Active/passive:** One node serves traffic
  - A standby replicates state and takes over on failure
- **Active/active:** every node serves traffic
  - Load shifts to survivors on failure
- Active/passive is easier to reason about but wastes capacity
- Active/active requires either stateless work or careful state management
- N+1 and N+2
  - Keep extra capacity so the service survives node loss



Load balancing and health

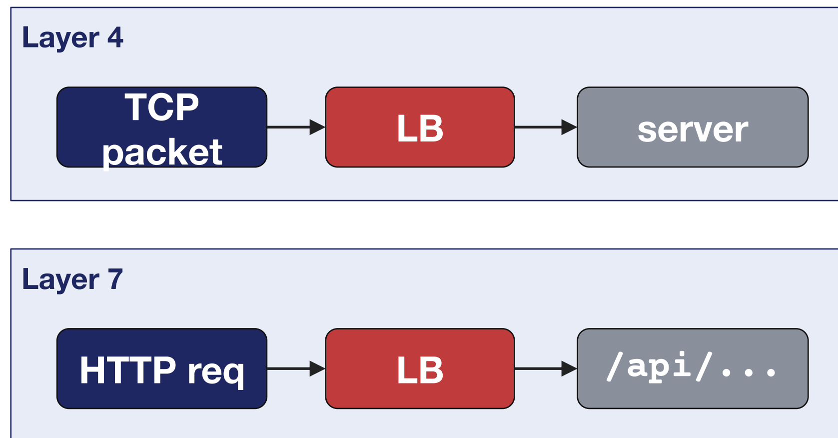
Why load balance?

- Spread load so no replica becomes a hot spot
- Hide individual node failures from clients
- Enable rolling updates: drain traffic before replacement
- Provide a stable address while workers come and go

CS 417 © 2026 Paul Krzyzanowski

L4 vs L7

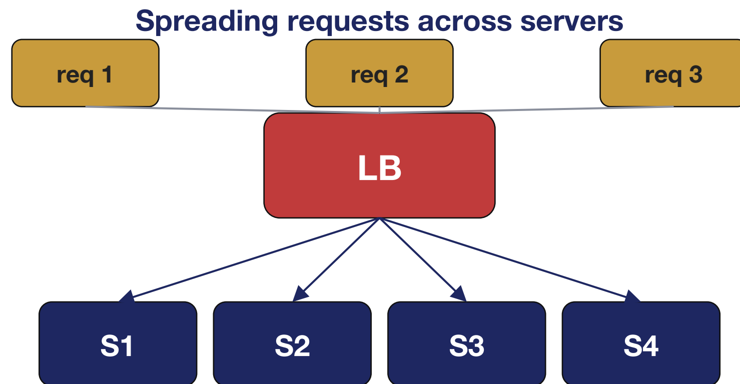
- **Layer 4 (transport)**
  - Balances TCP or UDP traffic
  - Sees IP addresses and ports, not HTTP requests
- **Layer 7 (application)**
  - Balances application requests, usually HTTP
  - can route by URL path, headers, cookies, or method
  - HTTPS inspection requires **TLS termination** at the load balancer
- **Common pattern**
  - L4 in front of L7



L4 routes by IP address and port
L7 reads the HTTP request

Balancing algorithms

- **Round robin**
  Rotate evenly through backends
- **Least connections**
  Prefer the least busy backend
- **Weighted**
  Honor backend capacity differences
- **Latency-aware**
  Pick the fastest responder
- **Power of two choices:** sample two backends & pick the less loaded one



Round robin · Least connections · Weighted · Latency-aware · Power of two

Health checks

- **Liveness:** *is the process still functioning?* If not, restart it
- **Readiness:** *is it ready to serve traffic?* If not, remove from the pool
- **Startup:** give slow-starting processes time before liveness checks start

- Health checks should be cheap, frequent, and service-specific.
- A TCP connection can succeed even when the application is stuck.

Design consequences

Stateless services

- Any instance can serve any request
- Easy to load balance, restart, and scale
- State lives elsewhere: databases, caches, object stores
- Common pattern for modern web and API tiers
- Enables autoscaling, rolling updates, and aggressive rollouts

Handling state

- Some services must hold state: databases, caches, coordinators
- **Common tools:** replication, sharding, external storage
- Consensus protocols such as Raft or Paxos keep replicas consistent
- Kubernetes StatefulSets provide stable identities and ordered rollouts
- Backup, failover, and migration become core design concerns

CS 417 © 2026 Paul Krzyzanowski

Takeaways

- A cluster is a software abstraction over unreliable machines
  - Decisions:
    placement, failure recovery, and traffic routing
  - Core mechanisms:
    schedulers, isolation, reconciliation, load balancing, and health checks
  - Redundancy requires:
    correct coordination: quorum, leases, and consensus
- Stateless services are easy to scale
  - Stateful services require careful recovery and migration

The End