Scaling Web Applications: CDN & Beyond

Jason Kuruzovich

RPI

Jason Kuruzovich, PhD
Associate Professor of Business

Feel free to reach out for help

Book a meeting:
https://bit.ly/jason-rpi

Rensselaer Polytechnic Institute
kuruzj@rpi.edu
518-698-9910

Scaling Web Applications: Key Topics

Scaling Topics We Will Cover

- **Scaling Dimensions**: vertical, horizontal, geographic
- **Content Delivery Networks (CDN)**: edge caching, PoPs, cache headers
- **Load Balancing**: algorithms and routing strategies
- **Database Scaling**: read replicas, sharding, partitioning
- **Application Caching, CAP Theorem, Async Queues**

Theoretical Focus:

- Why systems fail under load — and how distributed design solves it
- Tradeoffs between consistency, availability, and performance

Scaling Learning Objectives

- By the end of class, students should be able to:
- Describe the **dimensions of scalability** (vertical, horizontal, geographic)
- Explain how a **CDN reduces latency** through edge caching and PoPs
- Compare **load balancing algorithms** and when to use each
- Articulate the **CAP theorem** and the PACELC extension
- Distinguish **caching patterns** (cache-aside, write-through, write-behind)
- Evaluate tradeoffs in **database scaling** strategies

Readings

Foundational Texts & Industry References

1. Cloudflare — What is a caching?What is a CDN?
https://www.cloudflare.com/learning/cdn/what-is-a-cdn/
https://www.cloudflare.com/learning/cdn/what-is-caching/

2. Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation
https://doi.org/10.1109/ACCESS.2022.3152803

3. What is Amazon ElastiCache
https://docs.aws.amazon.com/AmazonElastiCache/latest/dg/WhatIs.html

3. Redis — Caching Patterns Documentation

6. NGINX — Load Balancing Guide (nginx.com/resources)

What does “scaling” mean for a web application?

Dimensions of Scalability

Scaling means more than just adding servers:

- **Throughput**: requests per second the system can handle
- **Latency**: time from request to response (track p50, p95, p99)
- **Availability**: % of time the system responds correctly (e.g., 99.99%)
- **Geographic Scale**: serving users across regions with low latency

Two Fundamental Strategies:

- **Vertical Scaling (Scale Up)**: bigger machine, more RAM/CPU
- **Horizontal Scaling (Scale Out)**: more machines, distributed load
- Vertical scaling hits hardware limits; horizontal scaling introduces distributed complexity

Vertical vs. Horizontal Scaling

Vertical Scaling (Scale Up):

- Add CPU, RAM, or faster storage to a single server
- Pros: simple, no code changes required
- Cons: hardware ceiling, single point of failure

Horizontal Scaling (Scale Out):

- Add more servers; distribute load across them
- Requires stateless application design
- Enables near-infinite scale via cloud elasticity
- Introduces consistency, coordination, and failure-mode complexity
- **Key insight**: modern web-scale systems use both — horizontal as primary, vertical to optimize nodes

Content Delivery Networks: Core Concepts

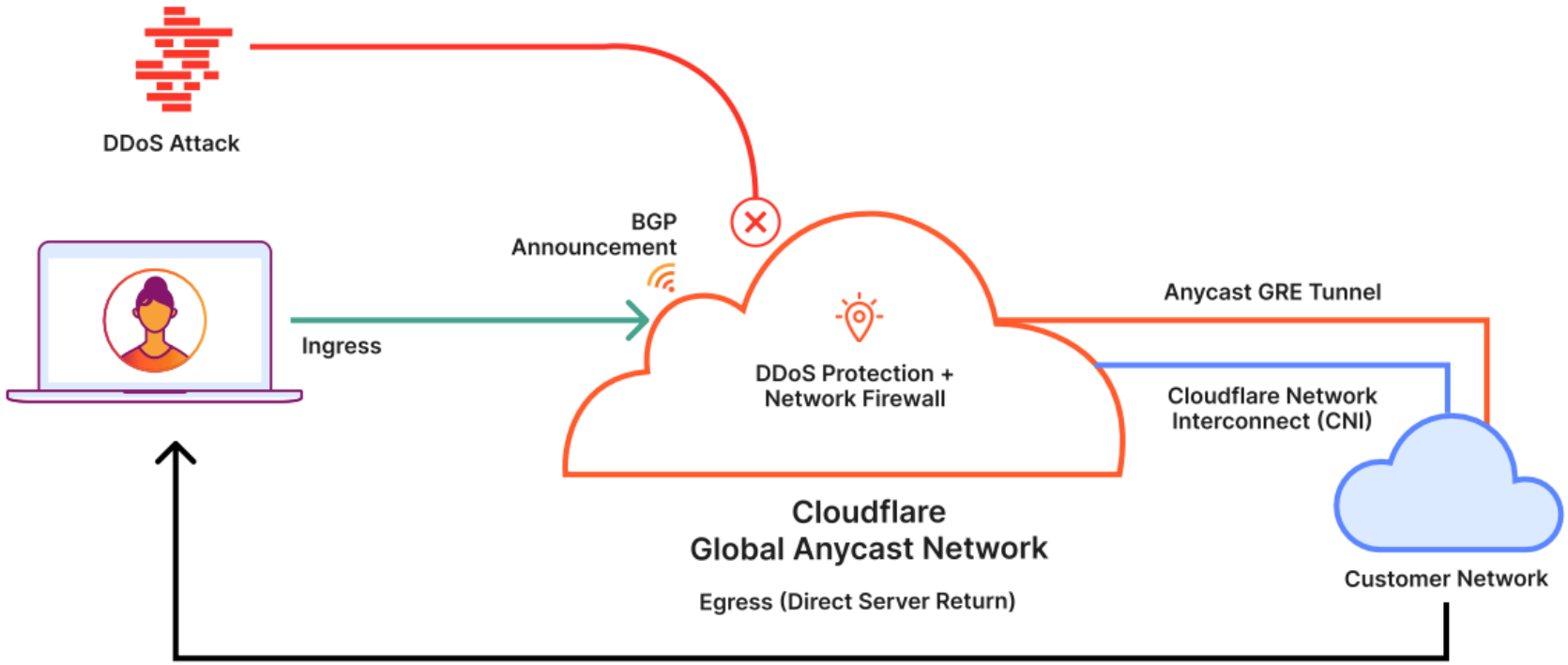
- A CDN is a geographically distributed network of servers that cache and serve content close to users

What a CDN Caches:

- Static assets: images, JS bundles, CSS, fonts
- Video/audio streams, software downloads
- Increasingly: dynamic API responses (edge computing)

Core Benefits:

- **Reduced latency**: user hits a nearby edge node, not your origin server
- **Origin offload**: 60-90% of traffic never reaches your infrastructure
- **Resilience**: CDN absorbs DDoS traffic and shields the origin


DDoS Attack
BGP
Announcement
Ingress
DDoS Protection +
Network Firewall
Anycast GRE Tunnel
Cloudflare Network
Interconnect (CNI)
Cloudflare
Global Anycast Network
Customer Network
Egress (Direct Server Return)

CDN Architecture: PoPs & Edge Nodes

- **Point of Presence (PoP)**: a CDN data center in a geographic region
- Each PoP contains dozens to hundreds of edge servers
- User requests route to the nearest PoP via Anycast DNS

Cache Hierarchy:

- **L1 Edge Cache**: at the PoP closest to user (fastest, smallest)
- **L2 / Regional Cache**: larger, shared across PoPs in a region
- **Origin Server**: your infrastructure — only hit on cache miss

Cache Miss Waterfall:

- L1 miss → check L2 → if miss, fetch from origin → populate caches on return
- Cache hit ratio is the key CDN performance metric

How does a CDN actually deliver content faster?

CDN Caching: Cache-Control & TTLs

- The Cache-Control HTTP header governs what gets cached and for how long

Key Directives:

- **max-age=N**: cache this response for N seconds
- **s-maxage=N**: CDN-specific TTL (overrides max-age for shared caches)
- **no-store**: never cache (for sensitive or personalized data)
- **stale-while-revalidate**: serve stale while fetching fresh in background

Cache Invalidation Problem:

- "There are only two hard things in CS: cache invalidation and naming things"
- Solution: content-hashed filenames (main.a3f2c1.js) enable aggressive caching with instant invalidation on deploy

Load Balancing: Algorithms & Strategies

Common Algorithms:

- **Round Robin**: requests distributed sequentially; simple, ignores server load
- **Least Connections**: routes to the server with fewest active connections
- **IP Hash / Sticky Sessions**: same client always routes to same server
- **Weighted**: routes proportionally based on server capacity
- **Consistent Hashing**: minimizes cache disruption on node add/remove

Layer 4 vs. Layer 7:

- L4 (TCP/UDP): fast, routes on IP/port only, no content awareness
- L7 (HTTP): content-aware; can route by URL path, host header, or cookie value

Database Scaling

Read Replicas:

- One primary handles writes; N replicas handle reads
- Replication lag: replicas may be milliseconds behind primary
- Effective when read:write ratio is high (typical for most web apps)

Sharding (Horizontal Partitioning):

- Split data across multiple database nodes by a shard key
- **Range sharding**: partition by value range (e.g., user IDs 1-1M on shard 1)
- **Hash sharding**: distribute by hash of key; avoids hotspots

Tradeoffs:

- Sharding complicates cross-shard queries and distributed transactions
- Schema changes must be coordinated across all shards

What happens when your database becomes the bottleneck?

Application-Level Caching with Redis

- Redis: in-memory key-value store used as a caching layer in front of the database

Caching Patterns:

- **Cache-Aside (Lazy Loading)**: app checks cache first; on miss, loads DB and writes to cache
- **Read-Through**: cache sits in front of DB; fetches automatically on miss
- **Write-Through**: write to cache and DB synchronously; strong consistency
- **Write-Behind (Write-Back)**: write to cache immediately; async flush to DB later

Key Design Decisions:

- TTL selection: too short = high miss rate; too long = stale data
- Cache stampede: concurrent misses overwhelm origin — solved with mutex locks or probabilistic early expiration

The CAP Theorem

- Brewer (2000): a distributed system can guarantee at most 2 of 3 properties simultaneously
- **Consistency (C)**: every read returns the most recent write or an error
- **Availability (A)**: every request receives a response (may not be the latest data)
- **Partition Tolerance (P)**: system operates despite network partitions

In Practice:

- Network partitions are unavoidable in distributed systems — P must always be tolerated
- Real choice is CP (consistent, may be unavailable) vs AP (available, may return stale data)

PACELC Extension (Abadi, 2012):

- Even without partitions there is a latency vs consistency tradeoff
- Full spectrum: "If Partition, then Consistency or Availability; Else, Latency or Consistency"

Async Processing & Message Queues

- Not all work should happen synchronously in the request/response cycle

Why Async?

- Decouples producers from consumers; absorbs traffic spikes
- Long tasks (email, video processing, ML inference) should not block HTTP responses

Core Concepts:

- **Queue**: ordered list of messages; consumers pull and acknowledge processing
- **Pub/Sub**: one producer, many independent consumers (fan-out pattern)
- **Dead Letter Queue (DLQ)**: captures messages that fail repeated processing attempts

Common Systems:

- Redis (BullMQ), RabbitMQ, AWS SQS/SNS — transient message delivery
- Apache Kafka — durable log; consumers can replay from any offset

Autoscaling & Observability

Autoscaling:

- Automatically add/remove compute instances based on load metrics
- **Horizontal Pod Autoscaler (Kubernetes)**: scales pods on CPU, memory, or custom metrics
- **AWS Auto Scaling Groups**: scales EC2 instances; combined with Application Load Balancer
- Scaling has lag — pre-warming and predictive scaling reduce cold-start penalty

Observability (the 3 Pillars):

- **Metrics**: quantitative signals — RPS, p99 latency, error rate, CPU%
- **Logs**: structured event records; useful for debugging specific incidents
- **Traces**: end-to-end request paths across distributed services

Key Principle:

- You cannot scale what you cannot measure — observability is a prerequisite for effective autoscaling

Lab: In-Memory vs. Redis

Lab Setup: Two Identical Servers

- Same API, different backends — run both and compare:
- server-memory.js (port 3000)
  - Users stored in a JavaScript array: const users = []
  - Refresh tokens in an array: const refreshTokens = []
  - No caching — every data request takes ~2 seconds
- server-redis.js (port 3001)
  - Users stored as Redis Hashes: user:{username}
  - Refresh tokens in a Redis Set: refresh_tokens
  - Cache-aside pattern with 30-second TTL
- npm run compare — launches both simultaneously

Experiment 1: What Happens on Restart?

- Register a user on both servers, then restart both
- In-memory server:
  - Login fails — user data gone
  - Every code change in --watch mode triggers a restart
  - Imagine deploying a fix at 2 AM and logging out all users
- Redis server:
  - Login succeeds — data persists in Redis independently
  - Server process is stateless; Redis holds the state
  - Deploy, scale, crash, restart — no data loss
- This is why production systems externalize state

Experiment 2: Caching Performance

- Both servers have a /api/data/cached endpoint with a simulated 2-second query
- In-memory server — no cache available:
  - Request 1: ~2000ms
  - Request 2: ~2000ms
  - Request 100: ~2000ms
- Redis server — cache-aside pattern:
  - Request 1: ~2000ms (cache MISS — stores result with 30s TTL)
  - Request 2: ~2ms (cache HIT — served from Redis)
  - After 30s: ~2000ms again (TTL expired, fresh fetch)
- 400x speedup on cache hits — same data, dramatically less load on origin

Why This Matters: Horizontal Scaling

- In-memory storage breaks with multiple server instances
  - User registers on Server A, load balancer sends next request to Server B
  - Server B has no idea who this user is — login fails
  - Sticky sessions are a band-aid (breaks on failover)
- Redis as shared external state:
  - All server instances read/write to the same Redis
  - Any instance can handle any request — true statelessness
  - Add or remove instances without data loss
- This is the architectural shift that enables:
  - Autoscaling, blue-green deployments, container orchestration

Redis Data Types: Choosing the Right Tool

- Hash (user storage)
  - Like a JS object — field/value pairs under one key
  - redis.hSet('user:alice', { username, password })
  - O(1) field access — no scanning an array
- Set (refresh token store)
  - Unordered collection of unique strings
  - redis.sIsMember('refresh_tokens', token) — O(1) lookup
  - Compare: Array.includes() is O(n) — gets slower with more tokens
- String with TTL (cache)
  - redis.set(key, data, { EX: 30 }) — auto-deletes after 30s
  - No cleanup code needed — Redis handles expiration

Connecting to Lecture Concepts

- Cache-Aside Pattern (slide 17)
  - Implemented in /api/data/cached — check cache, fetch on miss, store with TTL
  - Students see MISS vs HIT in real-time response data
- CAP Theorem (slide 18)
  - Cache serves stale data during TTL window — trading consistency for speed
  - What if the underlying data changes before the cache expires?
- Database Scaling (slide 15)
  - Caching reduces reads hitting the database — similar benefit to read replicas
  - Often the first scaling step before adding infrastructure
- Async/Queues (slide 19)
  - Redis also powers BullMQ, Pub/Sub — same tool, different patterns