20 System Design Concepts You Must Know

Table of Contents

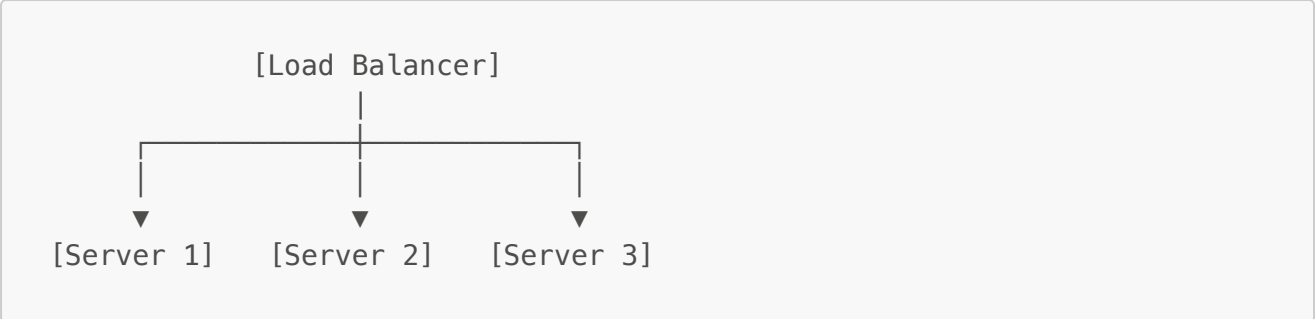
---

1. Load Balancer

What Is It?

A load balancer distributes incoming network traffic across multiple servers to ensure no single server bears too much load.

How It Works



Types of Load Balancing

Layer 4 (Transport Layer)

```
Operates at: TCP/UDP level
Decision based on: IP address, Port
Pros:
✓ Fast (no content inspection)
✓ Simple
✓ Low latency
Cons:
✗ No content-based routing
✗ No SSL termination

Use when: High throughput needed, simple routing
```

Layer 7 (Application Layer)

```
Operates at: HTTP/HTTPS level
Decision based on: URL, Headers, Cookies
Pros:
✓ Content-based routing
✓ SSL termination
✓ Request modification
✓ Caching
Cons:
✗ Slower (content inspection)
✗ More complex

Use when: Need intelligent routing, SSL termination
```

Load Balancing Algorithms

1. Round Robin

```
Request 1 → Server 1
Request 2 → Server 2
Request 3 → Server 3
Request 4 → Server 1 (cycle repeats)

Pros: Simple, even distribution
Cons: Doesn't consider server load
Use when: Servers have similar capacity
```

2. Least Connections

```
Server 1: 5 connections
Server 2: 3 connections ← Next request goes here
Server 3: 7 connections

Pros: Considers actual load
Cons: More complex, needs state
Use when: Long-lived connections (WebSocket)
```

3. IP Hash

```
hash(client_ip) % num_servers = server_id

Same client always goes to same server

Pros: Session affinity, cache friendly
Cons: Uneven distribution possible
Use when: Need sticky sessions
```

4. Weighted Round Robin

```
Server 1 (weight: 5): Gets 50% traffic
Server 2 (weight: 3): Gets 30% traffic
Server 3 (weight: 2): Gets 20% traffic

Pros: Accounts for server capacity
Use when: Heterogeneous server capacities
```

When to Use Load Balancer

✅ **Use when:**

- Multiple servers serving same content
- Need high availability (server redundancy)
- Scaling horizontally
- Traffic exceeds single server capacity
- Need zero-downtime deployments

❌ **Don't need when:**

- Single server is sufficient
- Very low traffic (<100 RPS)
- Cost is primary concern (adds infrastructure)

Real-World Examples

- **Netflix**: ELB for 100K+ concurrent connections
- **Facebook**: Custom load balancer for billions of users
- **AWS ELB/ALB**: Managed service for millions of customers

Configuration Example (NGINX)

```
upstream backend {
    least_conn;  # Algorithm
    server backend1.example.com weight=5;
    server backend2.example.com weight=3;
    server backend3.example.com backup;
}

server {
    listen 80;
    location / {
        proxy_pass http://backend;
        proxy_set_header X-Forwarded-For $remote_addr;
    }
}
```

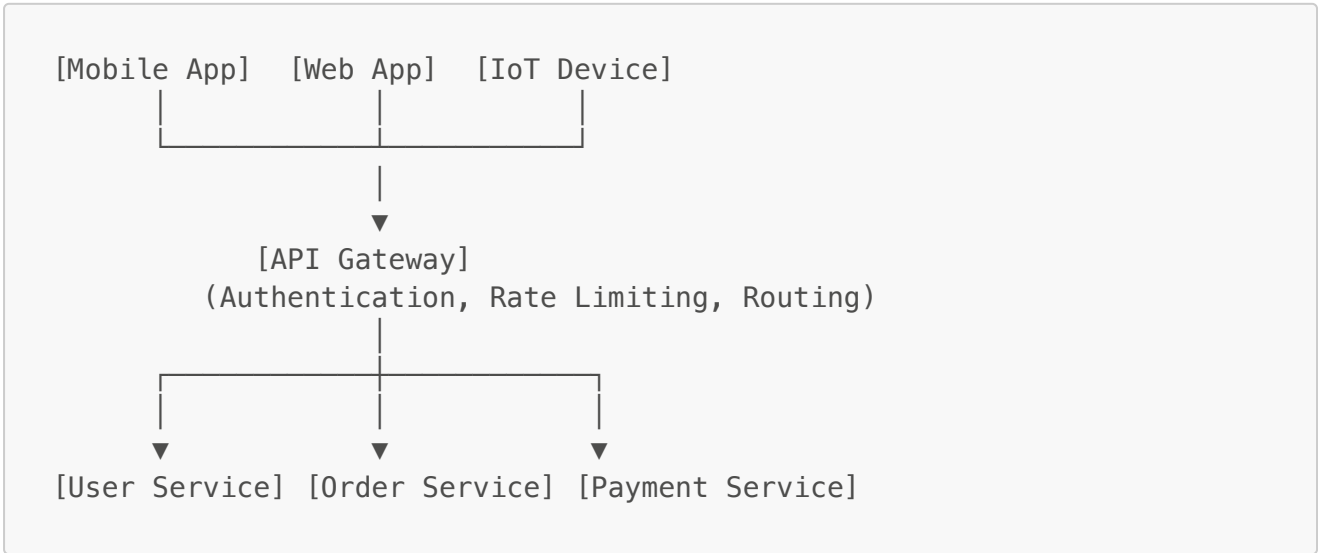
---

2. API Gateway

What Is It?

A single entry point for all client requests that routes to appropriate backend services. Acts as a reverse proxy with additional features.

Architecture



Key Responsibilities

1. Request Routing

```
GET /users/123 → User Service
POST /orders → Order Service
GET /products → Product Service

Benefits:
✓ Clients don't know backend structure
✓ Easy to refactor backend
✓ Version management (v1, v2 APIs)
```

2. Authentication & Authorization

```
Flow:
1. Client sends request with JWT token
2. API Gateway validates token
3. If valid: Route to backend
4. If invalid: Return 401 Unauthorized

Benefits:
✓ Centralized auth logic
✓ Backend services don't handle auth
✓ Single point for security policies
```

3. Rate Limiting

```
Per user/IP limits:
- Free tier: 100 requests/hour
- Premium: 10,000 requests/hour

Implementation at API Gateway level

Benefits:
✓ Protect backend from abuse
✓ Enforce tier limits
✓ Prevent DDoS
```

4. Request/Response Transformation

```
Client Request:
GET /v1/users/123

API Gateway transforms to:
GET /internal/user-service/getUserById?id=123
```

```
Response transformation:
Internal format → Public API format

Benefits:
✓ Hide internal details
✓ Maintain backward compatibility
✓ Aggregate multiple service calls
```

5. Protocol Translation

```
Client (HTTP/REST) → API Gateway → Backend (gRPC)

Benefits:
✓ Clients use simple HTTP
✓ Backend uses efficient gRPC
✓ Best of both worlds
```

When to Use API Gateway

✅ **Use when:**

- Microservices architecture
- Multiple client types (mobile, web, IoT)
- Need centralized authentication
- Rate limiting required
- Want to hide backend complexity
- Need API versioning

❌ **Don't need when:**

- Simple monolithic application
- Internal APIs only
- Very low traffic
- Direct service-to-service communication preferred

Popular API Gateway Solutions

- **AWS API Gateway**: Managed, serverless
- **Kong**: Open source, plugin-based
- **NGINX**: Lightweight, high performance
- **Apigee**: Enterprise, full-featured
- **Spring Cloud Gateway**: Java ecosystem

Implementation Example

```
// Express-based API Gateway
const express = require('express');
const app = express();

// Authentication middleware
app.use(async (req, res, next) => {
  const token = req.headers.authorization;
  if (!validateToken(token)) {
    return res.status(401).send('Unauthorized');
  }
  next();
});

// Rate limiting
const rateLimit = require('express-rate-limit');
app.use(rateLimit({
  windowMs: 60 * 1000, // 1 minute
  max: 100 // limit per window
}));

// Routing
app.get('/users/:id', async (req, res) => {
  const response = await fetch(`http://user-
service/users/${req.params.id}`);
  res.json(await response.json());
});

app.post('/orders', async (req, res) => {
  const response = await fetch('http://order-service/orders', {
    method: 'POST',
    body: JSON.stringify(req.body)
  });
  res.json(await response.json());
});
```

---

3. CDN (Content Delivery Network)

What Is It?

A geographically distributed network of servers that deliver content to users from the nearest location.

Architecture

```
User in California → CDN Edge (San Francisco)
User in New York → CDN Edge (New York)
User in London → CDN Edge (London)
              ↓
```

```
          [Origin Server]
        (Only on cache miss)
```

How CDN Works

1. Pull CDN (Most Common)

```
1. User requests image.jpg
2. CDN checks cache
   - Hit: Return from cache
   - Miss: Fetch from origin, cache, return

Cache Control:
├── TTL: 1 hour (how long to cache)
├── Cache-Control header
└── Purge API for updates

Use when: Content changes infrequently
```

2. Push CDN

```
1. Origin pushes content to CDN
2. Content pre-positioned on edge servers
3. User always gets content from edge

Use when:
- Content known in advance
- Scheduled releases
- High-traffic events
```

What to Cache

✅ **Good Candidates**

```
Static Assets:
├── Images (.jpg, .png, .webp)
├── Videos (.mp4, .webm)
├── JavaScript files (.js)
├── CSS files (.css)
├── Fonts (.woff, .ttf)
└── Documents (.pdf)

Semi-Static:
├── Product catalog
├── Blog posts
```

```
├── User profiles
└── (with short TTL 5-15 min)
```

❌ **Poor Candidates**

```
Dynamic Content:
├── User-specific data
├── Real-time data
├── Frequently changing data
├── Personalized content
└── Authentication tokens
```

CDN Features

1. Geographic Distribution

```
Benefits:
├── Reduced latency (closer to users)
├── Reduced load on origin
├── Better user experience
└── Can serve during origin outage

Example:
├── Origin (US): 200ms latency for EU users
├── CDN Edge (EU): 10ms latency
└── Improvement: 20x faster
```

2. DDoS Protection

```
CDN absorbs attack:
├── Distributed infrastructure
├── Traffic filtering
├── Rate limiting at edge
└── Origin protected
```

3. SSL/TLS Termination

```
Client ←[HTTPS]→ CDN ←[HTTP]→ Origin

Benefits:
├── Offload encryption from origin
```

```
├── Centralized certificate management
├── Better performance
```

When to Use CDN

✅ **Use when:**

- Serving static assets (images, videos, CSS, JS)
- Global user base
- High bandwidth costs
- Origin server under load
- Need DDoS protection
- Want to improve page load times

❌ **Don't need when:**

- All users in single geographic location
- Highly dynamic, personalized content
- Very low traffic
- Cost is primary concern

Cost-Benefit Analysis

```
Without CDN:
├── Bandwidth: $0.08/GB
├── 1 TB/day = $2,400/month
├── Latency: 200ms (global avg)
└── Origin load: High

With CDN:
├── CDN cost: $0.02/GB
├── 1 TB/day = $600/month
├── Origin bandwidth: 50 GB/day = $120/month
├── Total: $720/month
├── Latency: 20ms (edge)
└── Origin load: Low

Savings: 70% + Better performance
```

Popular CDN Providers

- **CloudFlare**: Free tier, DDoS protection
- **AWS CloudFront**: Integrated with AWS
- **Akamai**: Enterprise, extensive network
- **Fastly**: Real-time purging, edge computing
- **Google Cloud CDN**: Integrated with GCP

Configuration Example (CloudFront)

```
{
  "Origins": [{
    "DomainName": "origin.example.com",
    "Id": "my-origin"
  }],
  "DefaultCacheBehavior": {
    "TargetOriginId": "my-origin",
    "ViewerProtocolPolicy": "redirect-to-https",
    "AllowedMethods": ["GET", "HEAD"],
    "CachedMethods": ["GET", "HEAD"],
    "ForwardedValues": {
      "QueryString": false,
      "Headers": ["Origin"]
    },
    "MinTTL": 0,
    "DefaultTTL": 86400,
    "MaxTTL": 31536000
  }
}
```

---

4. Forward Proxy vs Reverse Proxy

Forward Proxy

What: Acts on behalf of **clients**

```
Client → [Forward Proxy] → Internet → Server

Client knows about proxy
Server doesn't know client's real IP
```

Use Cases:

1. **Access Control**

```
Corporate network:
Employee → Forward Proxy (checks policy) → Internet

Policies:
├── Block social media
├── Block malware sites
├── Log all requests
└── Enforce safe browsing
```

2. **Anonymity**

```
User → VPN/Proxy → Website

Benefits:
├── Hide real IP address
├── Bypass geo-restrictions
├── Privacy protection
```

3. **Caching**

```
All employees → Proxy (caches responses) → Internet

Benefits:
├── Faster access to common sites
├── Reduced bandwidth
└── Cost savings
```

Popular Tools: Squid, Privoxy, VPN services

Reverse Proxy

What: Acts on behalf of **servers**

```
Client → Internet → [Reverse Proxy] → Backend Servers

Client doesn't know about backend servers
Server is hidden behind proxy
```

Use Cases:

1. **Load Balancing**

```
Client → Reverse Proxy → Server 1, 2, or 3

Benefits:
├── Distribute traffic
├── High availability
└── Scalability
```

2. **SSL Termination**

```

```

```
Client ←[HTTPS]→ Reverse Proxy ←[HTTP]→ Backend

Benefits:
├── Offload SSL encryption from backend
├── Centralized certificate management
└── Better performance
```

3. **Caching**

```
Client → Reverse Proxy (cache) → Origin

Frequently requested content cached at proxy

Benefits:
├── Reduced backend load
├── Faster response times
└── Handle traffic spikes
```

4. **Security**

```
Client → Reverse Proxy (firewall) → Backend

Benefits:
├── Hide backend IP addresses
├── DDoS protection
├── WAF (Web Application Firewall)
└── Rate limiting
```

Popular Tools: NGINX, HAProxy, Apache, Envoy

Comparison Table

Feature	Forward Proxy	Reverse Proxy
Acts for	Client	Server
Visibility	Server doesn't see client	Client doesn't see server
Use case	Corporate network, VPN	Load balancing, caching
Who configures	Client	Server administrator
Anonymity	Client anonymity	Server anonymity

When to Use

Forward Proxy:

- Corporate network access control
- Bypass geographic restrictions
- Client anonymity (VPN)
- Centralized logging of outbound traffic

Reverse Proxy:

- Load balancing across servers
- SSL termination
- Caching static content
- Hiding backend infrastructure
- DDoS protection

Code Example (NGINX Reverse Proxy)

```
# Reverse proxy configuration
server {
    listen 80;
    server_name example.com;

    # SSL termination
    listen 443 ssl;
    ssl_certificate /path/to/cert.pem;
    ssl_certificate_key /path/to/key.pem;

    # Caching
    proxy_cache_path /data/nginx/cache levels=1:2
keys_zone=my_cache:10m;

    location / {
        proxy_pass http://backend;
        proxy_cache my_cache;
        proxy_cache_valid 200 1h;

        # Headers
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    }

    # Load balancing
    upstream backend {
        server backend1.example.com;
        server backend2.example.com;
        server backend3.example.com;
    }
}
```

---

5. Caching

What Is It?

Storing frequently accessed data in a faster storage layer to reduce latency and backend load.

Cache Hierarchy

```
L1: Browser Cache (0.1ms)
    ↓ miss
L2: CDN/Edge Cache (10-50ms)
    ↓ miss
L3: Application Cache - Redis (1-5ms)
    ↓ miss
L4: Database (50-200ms)
```

Caching Strategies

1. Cache-Aside (Lazy Loading)

```
Read Flow:
1. Check cache
2. If hit: Return data
3. If miss:
   a. Query database
   b. Store in cache
   c. Return data

Write Flow:
1. Write to database
2. Invalidate cache (or update)

Code:
data = cache.get(key)
if data is None:
    data = database.query(key)
    cache.set(key, data, ttl=300)
return data

Pros:
✓ Only cache what's requested
✓ Cache failure doesn't break app
Cons:
✗ Cache miss penalty
✗ Potential stale data

Use when: Read-heavy workload
```

2. Write-Through Cache

```
Write Flow:
1. Write to cache
2. Cache writes to database (synchronously)
3. Return success

Read Flow:
1. Check cache (always has data)
2. Return from cache

Pros:
✓ Cache always consistent with DB
✓ No cache miss penalty on reads
Cons:
✗ Slower writes
✗ Unused data in cache

Use when: Read >> Write, need consistency
```

3. Write-Behind (Write-Back)

```
Write Flow:
1. Write to cache (fast)
2. Return success immediately
3. Async write to database (later)

Pros:
✓ Fast writes
✓ Reduced database load
Cons:
✗ Risk of data loss
✗ Complex implementation

Use when: High write volume, can tolerate some data loss
```

4. Refresh-Ahead

```
Before TTL expires:
1. Check if item accessed recently
2. If yes: Refresh from database
3. Update cache with new data

Pros:
✓ No cache miss for popular items
✓ Always fresh data
```

```
Cons:
✗ Wasted refreshes for unpopular items

Use when: Predictable access patterns
```

Cache Eviction Policies

LRU (Least Recently Used)

```
Cache full? → Remove least recently accessed item

Example:
Access order: A, B, C, D, E (cache size: 3)
Cache: [C, D, E] (A, B evicted)

Use when: Recent items likely to be reused
Most common choice
```

LFU (Least Frequently Used)

```
Cache full? → Remove least frequently accessed item

Tracks access count per item

Use when: Popular items should stay cached
```

FIFO (First In First Out)

```
Cache full? → Remove oldest item

Simple, no access tracking needed

Use when: Simplicity more important than hit rate
```

TTL (Time To Live)

```
Every item has expiration time
Expired items removed automatically

Example:
cache.set("user:123", data, ttl=300)  # 5 minutes
```

```
Use when: Data has natural expiration
```

What to Cache

✅ Good Candidates

```
Database query results:
├── User profiles
├── Product catalogs
├── Configuration data
└── Computed aggregations

Computed values:
├── Recommendation results
├── Search results
├── Report summaries
└── Rendered HTML/JSON

External API responses:
├── Weather data
├── Stock prices
└── Third-party API calls

Characteristics:
- Read frequently
- Write infrequently
- Computation expensive
- Large dataset
```

❌ Poor Candidates

```
Rapidly changing data:
├── Stock trades (real-time)
├── Live sports scores
├── Chat messages
└── Real-time analytics

User-specific sensitive data:
├── Payment information
├── Personal health data
└── Authentication tokens

Already fast operations:
└── Simple database lookups
```

Cache Stampede Problem

Problem:

```
Scenario: Cache expires for popular item
├── 1000 concurrent requests hit at once
├── All see cache miss
├── All query database simultaneously
└── Database overloaded

Timeline:
T0: Cache expires
T1: 1000 requests arrive
T2: All query database
T3: Database crashes
```

Solutions:

Option 1: Mutex/Lock

```python
def get_data(key):
    data = cache.get(key)
    if data is not None:
        return data

    # Try to acquire lock
    if cache.set_nx(f"{key}:lock", 1, ex=10):
        # Got lock, fetch from database
        data = database.query(key)
        cache.set(key, data, ex=300)
        cache.delete(f"{key}:lock")
        return data
    else:
        # Wait and retry
        time.sleep(0.1)
        return get_data(key)  # Recursive retry
```

Option 2: Probabilistic Early Expiration

```python
def get_data(key):
    data, expiry = cache.get_with_expiry(key)
    if data is None:
        return refresh_data(key)

    # Refresh probabilistically before expiry
    time_to_expiry = expiry - time.now()
    if random() < (1.0 / time_to_expiry):
```

```
        return refresh_data(key)

    return data
```

When to Use Caching

✅ **Use when:**

- Read-heavy workload (95%+ reads)
- Same data accessed frequently
- Database queries are slow
- External API calls are expensive
- Computed results are reusable
- Can tolerate slightly stale data

❌ **Don't use when:**

- Write-heavy workload
- Data changes constantly
- Need real-time accuracy
- Data is user-specific and private
- Cache adds more complexity than benefit

Cache Technologies

Technology	Use Case	Speed	Persistence
Redis	General purpose	Very fast	Optional
Memcached	Simple key-value	Very fast	No
Varnish	HTTP caching	Fast	No
CDN	Static assets	Fast	Yes
Browser	Client-side	Fastest	Yes

Best Practices

```
1. Set appropriate TTLs
   - Frequently changing: 5-15 minutes
   - Rarely changing: Hours to days

2. Use cache keys wisely
   - Include version in key: user:123:v2
   - Use consistent key format

3. Monitor cache hit ratio
   - Target: >80% hit rate
   - Alert if drops below threshold
```

```
4. Handle cache failures gracefully
   - App should work if cache is down
   - Degrade to database queries

5. Warm up cache
   - Pre-populate popular items
   - Avoid cold start issues
```

---

6. Database Sharding

What Is It?

Horizontally partitioning data across multiple database instances. Each shard contains a subset of the total data.

Why Shard?

Problem: Single database limitations

```
Single Database:
├── Storage: 1-10 TB per instance
├── Write throughput: Limited by single disk
├── Read throughput: Limited by single CPU
├── Memory: Limited by single machine
└── Eventually hits physical limits
```

Solution: Distribute across multiple databases

```
Total Data: 10 TB
Shards: 10 databases × 1 TB each

Benefits:
✓ No single database bottleneck
✓ Linear scalability
✓ Parallel query execution
```

Sharding Strategies

1. Hash-Based Sharding

```
shard_id = hash(user_id) % num_shards

Example:
```

```
├── user_id: 12345
├── hash(12345) = 8372
├── 8372 % 4 = 0
└── Store in Shard 0

Pros:
✓ Even distribution
✓ Simple to implement
✓ Predictable

Cons:
✗ Range queries span all shards
✗ Adding shards requires rehashing
✗ Related data may be on different shards

Use when: Even distribution is critical
```

2. Range-Based Sharding

```
Shard 0: user_id 1-1,000,000
Shard 1: user_id 1,000,001-2,000,000
Shard 2: user_id 2,000,001-3,000,000

Pros:
✓ Range queries efficient
✓ Easy to add shards (extend range)
✓ Related data likely together

Cons:
✗ Uneven distribution (hotspots)
✗ Need to track ranges

Use when: Range queries are common
```

3. Geographic Sharding

```
Shard US-WEST: US users
Shard EU: European users
Shard ASIA: Asian users

Pros:
✓ Data locality (faster access)
✓ Compliance (GDPR - data stays in region)
✓ Network cost reduction

Cons:
✗ Uneven distribution
```

```
✗ Complex cross-region queries

Use when: Geographic compliance required
```

4. Directory-Based Sharding

```
Lookup Table:
user_id → shard_id
12345 → Shard 2
67890 → Shard 1

Pros:
✓ Flexible (can rebalance)
✓ Control over placement
✓ Easy to add shards

Cons:
✗ Extra lookup (latency)
✗ Lookup table is bottleneck

Use when: Need flexibility in data placement
```

Challenges & Solutions

Challenge 1: Cross-Shard Queries

```
Problem:
Query: "Get all users age > 25"
Must query ALL shards

Solution approaches:
1. Scatter-Gather:
   ├── Query all shards in parallel
   ├── Aggregate results
   └── Return to client

2. Denormalization:
   ├── Duplicate data where needed
   └── Trade: Storage for query speed

3. Search Index:
   ├── Elasticsearch for cross-shard search
   └── Eventual consistency acceptable
```

Challenge 2: Rebalancing

```
Problem: Adding new shard

Before (3 shards):
hash(key) % 3 = shard

After (4 shards):
hash(key) % 4 = shard (Different!)

Solution: Consistent Hashing
└── See section #12 for details
```

Challenge 3: Hotspots

```
Problem: One shard gets all traffic

Example:
Celebrity user with millions of followers
All their data on one shard

Solutions:
1. Further partition hot entities
2. Use hash-based for better distribution
3. Replicate hot data across shards
```

Challenge 4: Transactions

```
Problem: Transaction spans multiple shards

Order on Shard 1
Payment on Shard 2

ACID transaction impossible

Solutions:
1. Avoid cross-shard transactions (design)
2. Two-Phase Commit (2PC) - slow
3. Saga pattern (eventual consistency)
4. Denormalize to keep data together
```

When to Use Sharding

✅ **Use when:**

- Single database can't handle load (>1TB data)
- Write throughput is bottleneck

- Need to scale beyond single machine
- Geographic distribution required
- Cost of vertical scaling too high

❌ **Avoid when:**

- Data fits in single database
- Transactions across entities are common
- Can scale vertically
- Application complexity not worth it
- Team lacks expertise

Example: Instagram Sharding

Approach: Hash-based sharding

```
Sharding Key: user_id

Shards: 4096 PostgreSQL databases

Benefits:
├── Even distribution
├── Linear scalability
├── User data co-located
└── Handles billions of users

Custom ID Generation:
├── 64-bit ID
├── 41 bits: Timestamp
├── 13 bits: Shard ID (4096 shards)
├── 10 bits: Sequence
└── Globally unique, time-sortable
```

---

7. Database Replication

What Is It?

Copying data from one database (primary) to one or more databases (replicas) to improve read performance and availability.

Replication Models

1. Master-Slave (Primary-Replica)

```
          [Primary/Master]
        (Writes)  ↓ Replication
          ┌─────────┴─────────┐
```

```
        ▼       ▼       ▼
    [Replica 1] [Replica 2] [Replica 3]
    (Reads)     (Reads)     (Reads)

Flow:
1. All writes go to Primary
2. Primary replicates to Replicas
3. Reads distributed across Replicas

Pros:
✓ Scale read capacity horizontally
✓ Simple mental model
✓ Widely supported

Cons:
✗ Single point of failure (primary)
✗ Write bottleneck (single primary)
✗ Replication lag

Use when: Read >> Write (95% reads)
```

2. Master-Master (Multi-Primary)

```
[Primary 1] ←→ [Primary 2]
    (R/W)          (R/W)

Both accept writes
Bi-directional replication

Pros:
✓ No single point of failure
✓ Scale writes
✓ Geographic distribution

Cons:
✗ Write conflicts possible
✗ Complex conflict resolution
✗ Higher latency

Use when: Write scaling needed, multiple regions
```

3. Cascading Replication

```
     [Primary]
         ↓
    [Replica 1] ← Primary replicates here
         ↓
```

```
    [Replica 2] ← Replica 1 replicates here

Pros:
✓ Reduces load on primary
✓ Better for many replicas

Cons:
✗ Higher replication lag for downstream
✗ More complex

Use when: Many replicas needed (>5)
```

Replication Methods

Synchronous Replication

```
Write Flow:
1. Write to Primary
2. Wait for Replica to confirm
3. Return success to client

Pros:
✓ Strong consistency
✓ No data loss
Cons:
✗ Slower writes (wait for replica)
✗ Reduced availability (both must be up)

Use when: Data consistency is critical (financial transactions)
```

Asynchronous Replication

```
Write Flow:
1. Write to Primary
2. Return success immediately
3. Replicate to Replicas (async)

Pros:
✓ Fast writes
✓ Better availability
Cons:
✗ Replication lag (seconds to minutes)
✗ Potential data loss on primary failure

Use when: Performance > strict consistency
```

Replication Lag

Problem: Replica data is behind primary

```
T0: Write to Primary (user_age = 30)
T1: Client reads from Replica (user_age = 25) ← Old value
T2: Replication completes (user_age = 30)

Impact:
- Read-after-write inconsistency
- Confusing user experience
```

Solutions:

```
1. Read from Primary after write
   - Guarantees seeing own writes
   - Higher load on primary

2. Session consistency
   - Track last write timestamp
   - Wait for replica to catch up

3. Accept eventual consistency
   - Most common solution
   - Document behavior
```

When to Use Replication

✅ **Use when:**

- Read-heavy workload
- Need high availability (failover)
- Geographic distribution
- Disaster recovery
- Isolate read-only analytics queries

❌ **Don't need when:**

- Write-heavy workload
- Single database handles load
- Strong consistency always required
- Data is already distributed (sharding)

---

8. Database Indexing

What Is It?

A data structure that improves the speed of data retrieval operations on a database table.

How Indexes Work

```
Without Index (Full Table Scan):
SELECT * FROM users WHERE email = 'alice@example.com'

Database scans every row:
Row 1: bob@example.com (no)
Row 2: charlie@example.com (no)
...
Row 1M: alice@example.com (match!) ← Found after 1M checks
Time: ~1 second

With Index (B-Tree):
1. Look up email in index (binary search)
2. Index points to row location
3. Jump directly to row
Time: ~1 millisecond

Improvement: 1000x faster
```

Index Types

1. B-Tree Index (Most Common)

```
Structure:
                [Node]
               /      \
          [Node]      [Node]
          /    \      /    \
     [Leaf] [Leaf] [Leaf] [Leaf]

Properties:
├── Balanced tree
├── Sorted data
├── Logarithmic search time O(log n)
└── Supports range queries

Use for:
- Primary keys
- Equality queries (WHERE id = 123)
- Range queries (WHERE age BETWEEN 25 AND 35)
- Sorting (ORDER BY created_at)
```

2. Hash Index

```
hash(key) → location

Properties:
├── Constant time O(1)
├── Only equality queries
├── No range queries
└── No sorting

Use for:
- Exact match queries
- Cache lookups
- When range queries not needed

Example:
CREATE INDEX idx_email_hash ON users USING HASH(email);
```

3. Composite Index

```
Index on multiple columns:
CREATE INDEX idx_name_age ON users(last_name, first_name, age);

Query optimization:
✓ WHERE last_name = 'Smith'
✓ WHERE last_name = 'Smith' AND first_name = 'John'
✓ WHERE last_name = 'Smith' AND first_name = 'John' AND age = 30
✗ WHERE first_name = 'John' (doesn't use index)
✗ WHERE age = 30 (doesn't use index)

Rule: Leftmost prefix must be used

Use when: Queries filter on multiple columns
```

4. Full-Text Index

```
Index for text search:
CREATE FULLTEXT INDEX idx_description ON products(description);

Query:
SELECT * FROM products WHERE MATCH(description) AGAINST('laptop');

Properties:
├── Tokenizes text
├── Supports relevance ranking
├── Handles stemming (run, running, ran)
└── Language-aware

Use for:
```

```
- Search functionality
- Article/blog search
- Product descriptions
```

5. Spatial Index

```
Index for geographic data:
CREATE SPATIAL INDEX idx_location ON stores(location);

Query:
SELECT * FROM stores
WHERE ST_Distance(location, POINT(lat, lng)) < 5000;

Use for:
- Location-based queries
- "Find stores near me"
- GIS applications
```

Index Trade-offs

Pros:

```
✓ Faster queries (1000x improvement)
✓ Reduced I/O operations
✓ Better user experience
```

Cons:

```
✗ Slower writes (must update index)
✗ Extra storage (10-30% of table size)
✗ Memory overhead
✗ Maintenance cost
```

Write Performance Impact

```
Without Index:
INSERT: 1ms (just write row)

With 5 Indexes:
INSERT: 6ms (write row + update 5 indexes)

Rule of Thumb:
```

```
- Each index adds ~20% write overhead
- Limit to 5-7 indexes per table
```

When to Create Index

✅ **Create index when:**

- Column used in WHERE clause frequently
- Column used in JOIN conditions
- Column used for ORDER BY / GROUP BY
- Query is slow (>100ms)
- Table has >10,000 rows

❌ **Don't create index when:**

- Table is small (<1000 rows)
- Column has low cardinality (few unique values like boolean)
- Column rarely queried
- Write performance more critical
- Too many indexes already exist

Index Best Practices

```
1. Index selectivity matters
   Good: email (unique values)
   Bad: gender (2-3 values)

2. Composite index column order
   - Most selective column first
   - Or most frequently queried

3. Cover indexes
   - Include all columns in SELECT
   - Avoid table lookup entirely

4. Monitor index usage
   - Remove unused indexes
   - Indexes cost storage and write performance

5. Analyze query patterns
   - Use EXPLAIN to see execution plan
   - Identify missing indexes
```

Example: Index Analysis

```
-- Check query execution plan
EXPLAIN SELECT * FROM users WHERE email = 'alice@example.com';
```

```
Without index:
Seq Scan on users  (cost=0.00..1000.00 rows=1 width=100)
  Filter: (email = 'alice@example.com')
Planning time: 0.1ms
Execution time: 1000ms ← Slow!

-- Create index
CREATE INDEX idx_users_email ON users(email);

With index:
Index Scan using idx_users_email  (cost=0.42..8.44 rows=1 width=100)
  Index Cond: (email = 'alice@example.com')
Planning time: 0.2ms
Execution time: 1ms ← 1000x faster!
```

---

9. Message Queue

What Is It?

An asynchronous communication mechanism where messages are stored in a queue until the receiver processes them.

Architecture

```
[Producer] → [Message Queue] → [Consumer]
 (Send)       (Store)          (Process)

Decoupling: Producer and Consumer independent
```

How It Works

```
1. Producer sends message to queue
2. Queue stores message persistently
3. Consumer pulls message (or queue pushes)
4. Consumer processes message
5. Consumer acknowledges completion
6. Queue removes message

Benefits:
├── Asynchronous processing
├── Decoupling of services
├── Load buffering
└── Guaranteed delivery
```

Common Patterns

1. Task Queue

```
Use case: Background job processing

Flow:
Web App → Queue → Worker
"Send email" → Store → Email Service

Example:
- User signs up
- Add "send welcome email" to queue
- Return success immediately
- Email sent asynchronously

Benefits:
✓ Fast user response
✓ Retry on failure
✓ Scale workers independently
```

2. Event Queue

```
Use case: Event-driven architecture

Flow:
Order Service → Queue → [Inventory, Billing, Notification]

Example:
- Order placed
- Queue message: "OrderCreated"
- Multiple services process event independently

Benefits:
✓ Loose coupling
✓ Easy to add new processors
✓ Service independence
```

3. Priority Queue

```
Messages have priority levels:
- Critical: Process immediately
- High: Process within 1 minute
- Normal: Process within 5 minutes
- Low: Process when idle
```

```
Use case:
- Payment processing (critical)
- Welcome emails (low)

Implementation:
Multiple queues or priority field
```

Message Queue vs Pub/Sub

```
Message Queue (Point-to-Point):
Producer → Queue → Single Consumer

Characteristics:
├── One consumer per message
├── Message deleted after consumption
└── Load balancing across consumers

Pub/Sub (Many-to-Many):
Publisher → Topic → Multiple Subscribers

Characteristics:
├── Message copied to all subscribers
├── Broadcast pattern
└── See section #10 for details
```

Popular Message Queues

RabbitMQ

```
Features:
├── AMQP protocol
├── Complex routing
├── Priority queues
├── Delayed messages
└── Mature, stable

Use when: Need complex routing, mature ecosystem
```

Apache Kafka

```
Features:
├── High throughput (millions/sec)
├── Distributed, scalable
├── Message replay
├── Stream processing
```

```
└── Persistent storage

Use when: High volume, need replay, event streaming
```

Amazon SQS

```
Features:
├── Fully managed
├── No server management
├── Auto-scaling
├── Integrated with AWS
└── Simple API

Use when: AWS ecosystem, don't want to manage infrastructure
```

When to Use Message Queue

✅ **Use when:**

- Asynchronous processing needed
- Decouple services
- Handle traffic spikes (buffering)
- Need guaranteed delivery
- Want to retry failed operations
- Background job processing

❌ **Don't need when:**

- Synchronous response required
- Simple request-response pattern
- Low latency critical (<10ms)
- Adds unnecessary complexity

Code Example

```python
# Producer (FastAPI)
from redis import Redis
import json

queue = Redis(host='localhost', port=6379)

@app.post("/users")
async def create_user(user: User):
    # Save user to database
    db.save(user)

    # Queue welcome email (async)
```

```
    message = {
        "type": "welcome_email",
        "user_id": user.id,
        "email": user.email
    }
    queue.lpush("email_queue", json.dumps(message))

    return {"status": "success", "user_id": user.id}

# Consumer (Worker)
while True:
    # Block until message available
    _, message = queue.brpop("email_queue")
    data = json.loads(message)

    try:
        send_welcome_email(data['email'])
        print(f"Sent email to {data['email']}")
    except Exception as e:
        # Retry logic
        print(f"Failed: {e}")
        queue.lpush("email_queue", message)  # Re-queue
```

---

10. Pub/Sub Pattern

What Is It?

Publishers send messages to topics without knowing who will receive them. Subscribers receive messages from topics they're interested in.

Architecture

```
[Publisher 1] ──┐
[Publisher 2] ──┤
[Publisher 3] ──┤→ [Topic: Orders] ──┬→ [Subscriber A]
                                     ├→ [Subscriber B]
                                     └→ [Subscriber C]

Each subscriber gets a copy of the message
```

Pub/Sub vs Message Queue

Feature	Message Queue	Pub/Sub
Pattern	Point-to-point	Broadcast
Consumers	One per message	All subscribers

Feature	Message Queue	Pub/Sub
Use case	Task distribution	Event notification
Message lifetime	Until consumed	Until all subscribers process
Coupling	Tight (sender knows queue)	Loose (sender doesn't know subscribers)

Common Use Cases

1. Event Notification

```
Scenario: Order placed

Publisher (Order Service):
publish("orders", {
  "event": "OrderPlaced",
  "order_id": 12345,
  "user_id": 789,
  "total": 99.99
})

Subscribers:
├── Inventory Service: Reduce stock
├── Billing Service: Charge payment
├── Notification Service: Send email/SMS
├── Analytics Service: Track order
└── Recommendation Service: Update user preferences

Benefits:
✓ Order Service doesn't know about subscribers
✓ Easy to add new services
✓ Services process independently
```

2. Fan-Out Pattern

```
One message → Many consumers

Example: Live stream
├── Streamer publishes frame
├── 1M viewers subscribe
└── All receive same frame

Technologies:
- WebSockets
- Server-Sent Events (SSE)
- WebRTC
```

3. Real-Time Updates

```
Use case: Stock price updates

Publisher: Stock Exchange
Topic: "AAPL" (Apple stock)
Subscribers:
├── Trading apps
├── News services
├── Analytics platforms
└── Alert systems

Benefits:
✓ Real-time data distribution
✓ Scalable to many consumers
```

Message Delivery Semantics

At-Most-Once

```
Delivery: 0 or 1 times
Risk: Message may be lost
Implementation: Fire-and-forget

Use when: Performance > Reliability (e.g., metrics)
```

At-Least-Once

```
Delivery: 1 or more times
Risk: Duplicates possible
Implementation: Retry until acknowledgment

Use when: Cannot lose messages, idempotent operations
```

Exactly-Once

```
Delivery: Exactly 1 time
Implementation: Deduplication + transactions
Cost: Complex, slower

Use when: Financial transactions, critical operations
```

Topic Design

Fine-Grained Topics:

```
topics/users/created
topics/users/updated
topics/users/deleted

Pros:
✓ Subscribers get exactly what they need
✓ Reduced processing
Cons:
✗ Many topics to manage
```

Coarse-Grained Topics:

```
topics/users (all user events)

Message includes event type

Pros:
✓ Fewer topics
✓ Simpler management
Cons:
✗ Subscribers process all events
✗ Filtering needed
```

When to Use Pub/Sub

✅ **Use when:**

- Event-driven architecture
- Multiple services need same data
- Real-time notifications
- Broadcasting updates
- Microservices communication
- Want loose coupling

❌ **Don't need when:**

- Simple request-response
- Only one consumer
- Need guaranteed ordering
- Synchronous processing required

Technologies

- **Apache Kafka**: High throughput, persistent
- **Redis Pub/Sub**: Simple, in-memory
- **AWS SNS**: Managed, integrated
- **Google Cloud Pub/Sub**: Global, scalable
- **RabbitMQ**: Flexible, feature-rich

Code Example

```
# Publisher
import redis

r = redis.Redis()

def publish_event(topic, message):
    r.publish(topic, json.dumps(message))

# Publish order event
publish_event("orders", {
    "event": "OrderPlaced",
    "order_id": 12345,
    "user_id": 789
})

# Subscriber 1: Inventory Service
pubsub = r.pubsub()
pubsub.subscribe("orders")

for message in pubsub.listen():
    data = json.loads(message['data'])
    if data['event'] == 'OrderPlaced':
        reduce_inventory(data['order_id'])

# Subscriber 2: Email Service
pubsub2 = r.pubsub()
pubsub2.subscribe("orders")

for message in pubsub2.listen():
    data = json.loads(message['data'])
    if data['event'] == 'OrderPlaced':
        send_confirmation_email(data['user_id'])
```

---

11. Rate Limiting

What Is It?

Controlling the rate of requests a client can make to an API to prevent abuse and ensure fair usage.

Why Rate Limiting?

Problems without rate limiting:

```
1. Abuse/DoS:
   - Malicious user sends 1M requests/sec
   - Server overwhelmed
   - Legitimate users can't access service

2. Resource exhaustion:
   - Buggy client in infinite loop
   - Consumes all database connections
   - Affects all users

3. Cost control:
   - Excessive API calls
   - High cloud costs
   - Need to limit free tier users
```

Rate Limiting Algorithms

1. Token Bucket

```
Concept: Bucket with tokens, refilled at constant rate

Capacity: 100 tokens
Refill: 10 tokens/second

T0: [100 tokens] (full)
    ↓ 50 requests
T1: [50 tokens]
    ↓ (refill 10)
T2: [60 tokens]
    ↓ 0 requests
T3: [70 tokens]

Algorithm:
if bucket.tokens >= 1:
    bucket.tokens -= 1
    allow_request()
else:
    reject_request()

bucket.tokens = min(capacity, bucket.tokens + refill_rate *
time_elapsed)

Pros:
✓ Handles bursts
✓ Smooth traffic
✓ Memory efficient
```

```
Use when: Bursty traffic is legitimate (mobile apps sync)
```

2. Leaky Bucket

```
Concept: Requests processed at constant rate

Bucket processes requests at fixed rate (10/sec)
Excess requests overflow and are rejected

T0: 100 requests arrive
    ↓ Process 10/sec
T1: 90 in queue
T2: 80 in queue
...

Pros:
✓ Smooth output rate
✓ Predictable performance

Cons:
✗ No burst handling
✗ Queue can overflow

Use when: Need steady rate (video streaming)
```

3. Fixed Window Counter

```
Window: 1 hour starting at :00

09:00-10:00: 100 requests allowed
10:00-11:00: 100 requests allowed (reset)

Problem: Boundary burst
09:30: 100 requests
10:00: Reset
10:00: 100 requests
Result: 200 requests in 30 minutes!

Pros:
✓ Simple implementation
✓ Memory efficient

Cons:
✗ Boundary issue
✗ Uneven traffic
```

```
Use when: Simple solution acceptable
```

4. Sliding Window Log

```
Store timestamp of each request

Window: Last 1 hour
Check: Count requests in last 60 minutes

Example at 10:30:
Count requests from 09:30-10:30

Pros:
✓ Accurate
✓ No boundary issue

Cons:
✗ Memory intensive (store all timestamps)
✗ Slower (count operation)

Use when: Need precision, low traffic
```

5. Sliding Window Counter (Best Balance)

```
Hybrid approach:

Current window: 45 requests (0-60min)
Previous window: 80 requests (60-120min ago)
Current time: 30 minutes into window (50%)

Estimated = 45 + (80 × 50%) = 45 + 40 = 85

If limit is 100: ALLOW (85 < 100)

Pros:
✓ Accurate (~95%)
✓ Memory efficient (2 counters)
✓ Handles boundaries well

Use when: Production systems (recommended default)
```

Implementation Patterns

Distributed Rate Limiting

```
Challenge: Multiple servers, shared limits

Solution: Central

 Redis store

All servers → Redis → Atomic counters

Implementation:
INCR ratelimit:user_123:endpoint:/api:window:1640995200
EXPIRE ratelimit:user_123:endpoint:/api:window:1640995200 7200

if count > limit:
    reject()
```

When to Use Rate Limiting

✅ **Use when:**

- Public API exposed to internet
- Different user tiers (free, premium)
- Need to prevent abuse/DoS
- Resource protection (database, CPU)
- Cost control (cloud APIs)
- Fair usage enforcement

❌ **Don't need when:**

- Internal services only
- Fully trusted clients
- Very low traffic
- Other protection mechanisms sufficient

---

12. Consistent Hashing

What Is It?

A distributed hashing scheme that minimizes key remapping when hash table is resized.

The Problem with Traditional Hashing

```
Traditional: server = hash(key) % num_servers

With 3 servers:
hash("user1") % 3 = 2 → Server 2
hash("user2") % 3 = 0 → Server 0
```

```
Add 4th server:
hash("user1") % 4 = 1 → Server 1 (Changed!)
hash("user2") % 4 = 0 → Server 0 (Same)

Result: 75% of keys remapped!
```

Consistent Hashing Solution

```
Concept: Hash ring (0 to 2^32-1)

         Server 1 (hash: 100)
              ↓
      [Hash Ring]
    ↙            ↘
Server 3          Server 2
(hash: 300)       (hash: 200)

Key placement:
- hash("user1") = 150 → Clockwise → Server 2
- hash("user2") = 250 → Clockwise → Server 3

Add Server 4 (hash: 175):
- Only keys between 150-175 remapped
- 90%+ keys stay in place!

Benefits:
✓ Minimal remapping (only K/n keys, where K = total keys, n = servers)
✓ Smooth scaling
✓ Predictable impact
```

Virtual Nodes

```
Problem: Uneven distribution

Servers at: 100, 200, 201 (Server 2 and 3 close together)
Result: Uneven key distribution

Solution: Virtual nodes
Each physical server gets multiple positions on ring

Server 1: hash("S1-1")=50, hash("S1-2")=150, hash("S1-3")=250
Server 2: hash("S2-1")=75, hash("S2-2")=175, hash("S2-3")=275
Server 3: hash("S3-1")=100, hash("S3-2")=200, hash("S3-3")=300

Result: More even distribution (150-200 virtual nodes typical)
```

When to Use Consistent Hashing

✅ **Use when:**

- Distributed caching (Memcached cluster)
- Database sharding (adding/removing shards)
- Load balancing (servers come and go)
- Content delivery (CDN node selection)
- Peer-to-peer networks

❌ **Don't need when:**

- Fixed number of servers
- Can afford full data redistribution
- Simple modulo hashing works fine

Real-World Examples

Amazon DynamoDB:

- Uses consistent hashing for partitioning
- Virtual nodes for even distribution
- Automatic rebalancing

Cassandra:

- Token-based consistent hashing
- Each node owns range of tokens
- Scales to hundreds of nodes

Code Example

```
import hashlib

class ConsistentHash:
    def __init__(self, nodes=None, virtual_nodes=150):
        self.virtual_nodes = virtual_nodes
        self.ring = {}
        self.sorted_keys = []

        if nodes:
            for node in nodes:
                self.add_node(node)

    def _hash(self, key):
        return int(hashlib.md5(key.encode()).hexdigest(), 16)

    def add_node(self, node):
        for i in range(self.virtual_nodes):
            virtual_key = f"{node}:{i}"
            hash_val = self._hash(virtual_key)
            self.ring[hash_val] = node
```

```
            self.sorted_keys.append(hash_val)

        self.sorted_keys.sort()

    def remove_node(self, node):
        for i in range(self.virtual_nodes):
            virtual_key = f"{node}:{i}"
            hash_val = self._hash(virtual_key)
            del self.ring[hash_val]
            self.sorted_keys.remove(hash_val)

    def get_node(self, key):
        if not self.ring:
            return None

        hash_val = self._hash(key)

        # Find first node clockwise
        for key in self.sorted_keys:
            if key >= hash_val:
                return self.ring[key]

        # Wrap around
        return self.ring[self.sorted_keys[0]]

# Usage
ch = ConsistentHash(['server1', 'server2', 'server3'])
print(ch.get_node('user123'))  # server2
print(ch.get_node('user456'))  # server1

# Add server - minimal remapping
ch.add_node('server4')
```

---

13. CAP Theorem

What Is It?

A distributed system can only guarantee 2 out of 3 properties: Consistency, Availability, Partition Tolerance.

The Three Properties

Consistency (C)

```
All nodes see the same data at the same time

Example:
User writes: balance = $100
All reads return: $100 (not stale $50)
```

```
Strong Consistency:
├── Write completes when all nodes updated
├── Reads always get latest value
└── May be slower/unavailable
```

Availability (A)

```
Every request receives a response (success or failure)

Example:
System always responds even if some nodes are down

High Availability:
├── Request gets response quickly
├── May return stale data
└── Better user experience
```

Partition Tolerance (P)

```
System continues operating despite network failures

Example:
Datacenter connection lost
├── System keeps running
├── Each partition operates independently
└── Eventually reconciles when healed

Always Required:
└── Networks are unreliable, partitions will happen
```

The Trade-off

You MUST choose Partition Tolerance (networks fail)

Then choose between:

```
CP System (Consistency + Partition Tolerance):
├── Sacrifice: Availability
├── Behavior: Return error if can't guarantee consistency
├── Example: MongoDB (default), HBase, Redis
└── Use when: Correctness > availability (banking)

AP System (Availability + Partition Tolerance):
├── Sacrifice: Consistency
```

```
├── Behavior: Always respond, may return stale data
├── Example: Cassandra, CouchDB, DynamoDB
└── Use when: Availability > strict consistency (social media)

CA System (Consistency + Availability):
├── Not possible in distributed systems!
├── Network partitions WILL happen
└── Only viable for single-node systems
```

Real-World Examples

CP System: Banking

```
Scenario: Transfer $100 A → B

Network partition occurs

Behavior:
├── System detects partition
├── Cannot guarantee consistency across partitions
├── Rejects transaction
└── Returns error

Trade-off: Better to be unavailable than inconsistent
```

AP System: Social Media Feed

```
Scenario: Like a post

Network partition occurs

Behavior:
├── System accepts like
├── Updates local partition
├── Shows updated count immediately
├── Syncs when partition heals
└── Brief inconsistency acceptable

Trade-off: Better to show stale data than be unavailable
```

Consistency Models Spectrum

```
Strong Consistency (CP)
├── Linearizability
├── All reads see latest write
```

```
└── Example: Spanner, etcd

Eventual Consistency (AP)
├── Given time, all replicas converge
├── Short-term inconsistency possible
└── Example: DynamoDB, Cassandra

Causal Consistency (Middle Ground)
├── Causally related operations ordered
├── Independent operations can be reordered
└── Example: MongoDB with causal consistency
```

When to Choose

Choose CP when:

- Financial transactions
- Inventory management
- Booking systems
- Configuration management
- Correctness is critical

Choose AP when:

- Social media feeds
- Analytics data
- Caching layers
- Content delivery
- User experience > strict accuracy

---

14. Distributed Consensus

What Is It?

Achieving agreement among distributed nodes on a single data value, even in the presence of failures.

The Challenge

Problem:

```
3 nodes need to agree on a value

Node 1 proposes: "value1"
Node 2 proposes: "value2"
Node 3 crashes

How to reach consensus?
```

Requirements:

```
1. Agreement: All nodes decide on same value
2. Validity: Decided value was proposed by some node
3. Termination: All nodes eventually decide
4. Fault Tolerance: Works despite F failures
```

Consensus Algorithms

1. Paxos

```
Classic consensus algorithm (complex but correct)

Roles:
├── Proposer: Proposes values
├── Acceptor: Votes on proposals
├── Learner: Learns decided value

Phases:
1. Prepare: Proposer sends prepare request
2. Promise: Acceptors promise not to accept older proposals
3. Accept: Proposer sends accept request
4. Accepted: Acceptors accept if no newer proposal

Pros:
✓ Proven correct
✓ Handles failures
Cons:
✗ Complex to understand
✗ Difficult to implement

Use when: Need proven correctness (not recommended for most)
```

2. Raft (Understandable Consensus)

```
Easier to understand than Paxos

Roles:
├── Leader: Handles all writes
├── Follower: Replicates from leader
└── Candidate: Competes to become leader

Operations:
1. Leader Election:
   - If leader fails, followers become candidates
   - Candidate with most votes becomes leader
```

```
   - Requires majority (quorum)

2. Log Replication:
   - Leader receives write
   - Replicates to followers
   - Commits when majority acknowledge

Pros:
✓ Easier to understand
✓ Easier to implement
✓ Widely adopted

Use when: Need distributed consensus (etcd, Consul)
```

Quorum-Based Systems

Concept:

```
N nodes total
W nodes for write success
R nodes for read success

Ensure: W + R > N (guarantees overlap)

Example:
N = 5 nodes
W = 3 (write to 3 nodes)
R = 3 (read from 3 nodes)
W + R = 6 > 5 ✓

Guarantees: Read sees latest write
```

Trade-offs:

```
Strong Consistency:
W = N, R = 1 (write all, read any)
- Slow writes, fast reads
- Use when: Read-heavy

Eventual Consistency:
W = 1, R = 1 (write one, read one)
- Fast writes, fast reads
- May read stale data

Balanced:
W = 2, R = 2, N = 3
- Balance of speed and consistency
```

Use Cases

1. Leader Election

```
Scenario: Microservices need to elect a leader

Use: Raft or ZooKeeper
- Only leader performs certain operations
- Automatic failover if leader dies
- Prevents split-brain

Example: Kafka broker election
```

2. Distributed Configuration

```
Scenario: Share configuration across services

Use: etcd, Consul
- All services read from same source
- Atomic updates
- Watch for changes

Example: Feature flags, service discovery
```

3. Distributed Lock

```
Scenario: Only one process should perform task

Use: Redis (Redlock), ZooKeeper
- Acquire lock before critical section
- Release lock after completion
- Automatic expiration

Example: Cron job should run on only one server
```

When to Use Distributed Consensus

✅ **Use when:**

- Leader election needed
- Distributed configuration management
- Service discovery
- Distributed locking
- Need strong consistency guarantees

❌ **Don't need when:**

- Single server deployment
- Eventual consistency acceptable
- Performance is critical (consensus adds latency)
- Adds operational complexity not worth it

Technologies

etcd (Raft):

- Kubernetes uses for cluster state
- Distributed key-value store
- Watch mechanism for changes

Apache ZooKeeper (ZAB - like Paxos):

- Used by Kafka, Hadoop
- Coordination service
- Mature, battle-tested

Consul (Raft):

- Service discovery
- Configuration management
- Health checking

---

15. Circuit Breaker

What Is It?

A design pattern that prevents an application from repeatedly trying to execute an operation that's likely to fail, allowing it to recover.

The Problem

Without Circuit Breaker:

```
Service A calls Service B (which is down)

Every request:
1. Try to connect to Service B
2. Wait for timeout (30 seconds)
3. Fail and retry
4. Repeat...

Result:
├── Wasted resources (threads waiting)
├── Cascading failures
```

```
├── Poor user experience
└── System-wide outage
```

How Circuit Breaker Works

```
Three States:

┌─────────────────┐
│   CLOSED        │  Normal operation
│  (Working)      │  Requests pass through
└─────────────────┘  Track failures
        │
        │ Failure threshold reached (5 failures)
        ▼
┌─────────────────┐
│    OPEN         │  Stop calling service
│  (Broken)       │  Fail fast (immediately)
└─────────────────┘  Start timeout timer
        │
        │ After timeout (60 seconds)
        ▼
┌─────────────────┐
│ HALF-OPEN       │  Try limited requests
│  (Testing)      │  Check if service recovered
└─────────────────┘
        │
        ├──→ Success → CLOSED
        └──→ Failure → OPEN
```

Implementation Example

```python
class CircuitBreaker:
    def __init__(self, failure_threshold=5, timeout=60,
success_threshold=2):
        self.failure_threshold = failure_threshold
        self.timeout = timeout  # seconds
        self.success_threshold = success_threshold

        self.failure_count = 0
        self.success_count = 0
        self.last_failure_time = None
        self.state = 'CLOSED'  # CLOSED, OPEN, HALF_OPEN

    def call(self, func, *args, **kwargs):
        if self.state == 'OPEN':
            if time.time() - self.last_failure_time > self.timeout:
                self.state = 'HALF_OPEN'
                self.success_count = 0
```

```
            else:
                raise CircuitBreakerOpenError("Circuit breaker is OPEN")

        try:
            result = func(*args, **kwargs)
            self.on_success()
            return result
        except Exception as e:
            self.on_failure()
            raise e

    def on_success(self):
        self.failure_count = 0

        if self.state == 'HALF_OPEN':
            self.success_count += 1
            if self.success_count >= self.success_threshold:
                self.state = 'CLOSED'

    def on_failure(self):
        self.failure_count += 1
        self.last_failure_time = time.time()

        if self.failure_count >= self.failure_threshold:
            self.state = 'OPEN'

# Usage
breaker = CircuitBreaker(failure_threshold=5, timeout=60)

try:
    result = breaker.call(call_external_service, user_id=123)
except CircuitBreakerOpenError:
    # Fail fast, use fallback
    result = get_cached_data(user_id=123)
```

Benefits

```
1. Fail Fast:
   - Immediate error instead of waiting
   - Better user experience
   - Free up resources

2. Prevent Cascading Failures:
   - Don't overwhelm failing service
   - Give it time to recover
   - Protect entire system

3. Automatic Recovery:
   - Test service periodically
   - Resume when healthy
```

```
   - No manual intervention

4. Monitoring:
   - Circuit breaker state is observable
   - Clear signal of health
   - Alert on OPEN state
```

When to Use Circuit Breaker

✅ **Use when:**

- Calling external services (third-party APIs)
- Microservices communication
- Operations can fail temporarily
- Want to prevent cascading failures
- Need graceful degradation

❌ **Don't need when:**

- Calling internal database (use connection pooling)
- Immediate response critical
- No fallback available
- Simple retry is sufficient

Real-World Example: Netflix Hystrix

```
@HystrixCommand(
    fallbackMethod = "defaultRecommendations",
    commandProperties = {
        @HystrixProperty(
            name = "circuitBreaker.requestVolumeThreshold",
            value = "10"
        ),
        @HystrixProperty(
            name = "circuitBreaker.errorThresholdPercentage",
            value = "50"
        )
    }
)
public List<Movie> getRecommendations(String userId) {
    return recommendationService.get(userId);
}

// Fallback method
public List<Movie> defaultRecommendations(String userId) {
    // Return generic popular movies
    return getCachedPopularMovies();
}
```

16. Bloom Filter

What Is It?

A space-efficient probabilistic data structure used to test whether an element is a member of a set. Can have false positives but never false negatives.

How It Works

```
Bloom Filter: Bit array + Hash functions

Initialization:
[0][0][0][0][0][0][0][0]  (8-bit array)

Add "alice":
hash1("alice") = 2 → Set bit 2
hash2("alice") = 5 → Set bit 5
[0][0][1][0][0][1][0][0]

Add "bob":
hash1("bob") = 3 → Set bit 3
hash2("bob") = 7 → Set bit 7
[0][0][1][1][0][1][0][1]

Check "alice":
hash1("alice") = 2 → bit is 1 ✓
hash2("alice") = 5 → bit is 1 ✓
Result: Probably in set

Check "charlie":
hash1("charlie") = 2 → bit is 1
hash2("charlie") = 4 → bit is 0 ✗
Result: Definitely NOT in set

Check "dave":
hash1("dave") = 3 → bit is 1
hash2("dave") = 5 → bit is 1
Result: Probably in set (FALSE POSITIVE!)
```

Characteristics

```
Guarantees:
✓ If says "no" → Definitely not in set (100% accurate)
✗ If says "yes" → Probably in set (false positive possible)

Space Efficiency:
- 1% false positive rate: ~10 bits per element
- Traditional set: 32-64 bits per element
```

```
- Savings: 3-6x less memory

No Deletion:
- Can't remove elements (bits shared)
- Use counting Bloom filter if deletion needed
```

Use Cases

1. Check Before Expensive Operation

```
Scenario: Check if username exists

Without Bloom Filter:
Every username check → Database query
- 1M requests/day
- 1M database queries
- Expensive!

With Bloom Filter:
1. Check Bloom filter (in-memory, fast)
2. If "no" → Username available (no DB query)
3. If "yes" → Check database (might be false positive)

Result:
├── 95% usernames don't exist
├── 95% of queries avoided
├── Only 5% + false positives hit database
└── 10x reduction in database load
```

2. URL Shortener - Check Collision

```
Scenario: Generate short URL

Without Bloom Filter:
1. Generate random short code
2. Check database if exists
3. If exists, regenerate
4. Repeat...

With Bloom Filter:
1. Generate random short code
2. Check Bloom filter
3. If "no" → Definitely unused, save to DB
4. If "yes" → Might be used, check DB

Result: Avoid most database lookups
```

3. Distributed Cache - Reduce Misses

```
Scenario: Multi-level cache

L1 Cache (local) → L2 Cache (Redis) → Database

With Bloom Filter:
1. Check Bloom filter for key
2. If "no" → Skip L2, go straight to database
3. If "yes" → Check L2 cache

Result: Reduce unnecessary L2 cache lookups
```

4. Malicious URL Detection

```
Bloom filter contains known malicious URLs

Check URL before loading:
1. Hash URL
2. Check Bloom filter
3. If "yes" → Block (might be false positive)
4. If "no" → Safe (definitely safe)

Trade-off: Occasional false positive acceptable for security
```

When to Use Bloom Filter

✅ **Use when:**

- Need to check set membership
- Can tolerate false positives
- Memory is constrained
- Speed is critical
- Negative queries are common (item not in set)

❌ **Don't use when:**

- False positives unacceptable
- Need to delete elements
- Set is small (traditional set is fine)
- Need exact counts

Code Example

```
import mmh3  # MurmurHash
from bitarray import bitarray

class BloomFilter:
    def __init__(self, size=1000, hash_count=3):
        self.size = size
        self.hash_count = hash_count
        self.bit_array = bitarray(size)
        self.bit_array.setall(0)

    def add(self, item):
        for i in range(self.hash_count):
            index = mmh3.hash(item, i) % self.size
            self.bit_array[index] = 1

    def contains(self, item):
        for i in range(self.hash_count):
            index = mmh3.hash(item, i) % self.size
            if self.bit_array[index] == 0:
                return False  # Definitely not in set
        return True  # Probably in set

    def __len__(self):
        return self.bit_array.count()

# Usage
bf = BloomFilter(size=1000, hash_count=3)

# Add usernames
bf.add("alice")
bf.add("bob")
bf.add("charlie")

# Check
print(bf.contains("alice"))    # True (definitely exists)
print(bf.contains("dave"))     # False (definitely doesn't exist)
print(bf.contains("unknown"))  # Might be True (false positive)
```

Size Calculation

```
Formula:
m = -n * ln(p) / (ln(2)^2)
k = (m/n) * ln(2)

Where:
m = number of bits
n = number of elements
p = desired false positive rate
k = number of hash functions
```

```
Example:
n = 1,000,000 elements
p = 0.01 (1% false positive)

m = −1,000,000 ∗ ln(0.01) / (ln(2)^2) ≈ 9,585,059 bits ≈ 1.2 MB
k = (9,585,059 / 1,000,000) ∗ ln(2) ≈ 7 hash functions

For 1M elements, only 1.2 MB!
Traditional set: 1M × 64 bits = 8 MB
```

---

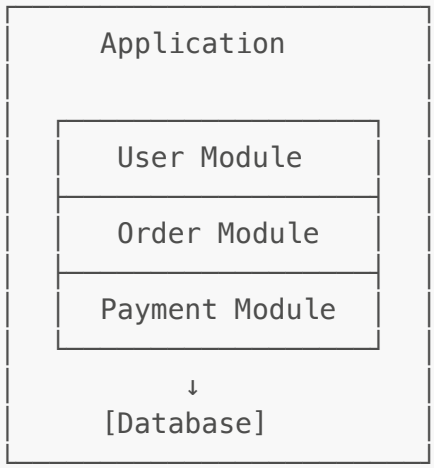
17. Microservices Architecture

What Is It?

An architectural style that structures an application as a collection of loosely coupled, independently deployable services.

Monolith vs Microservices

Monolith




```
Pros:
✓ Simple deployment
✓ Easy to develop initially
✓ Simple transactions
✓ No network overhead

Cons:
✗ Tight coupling
✗ Scale entire app (can't scale just one module)
✗ Single point of failure
✗ Long deployment cycles
```

Microservices

```
[User Service]  [Order Service]  [Payment Service]
      ↓               ↓                 ↓
   [User DB]      [Order DB]        [Payment DB]

Communication: REST APIs, gRPC, Message Queue

Pros:
✓ Independent scaling
✓ Technology diversity
✓ Independent deployment
✓ Fault isolation
✓ Team autonomy

Cons:
✗ Network latency
✗ Distributed transactions complex
✗ Operational overhead
✗ Data consistency challenges
```

When to Use Microservices

✅ **Use when:**

- Large team (>50 developers)
- Different parts need different scaling
- Want to use different technologies
- Need frequent deployments
- Can handle operational complexity
- Have DevOps expertise

❌ **Don't use when:**

- Small team (<10 developers)
- Simple application
- Don't need independent scaling
- Team lacks distributed systems experience
- Network latency is critical

Best Practices

```
1. Domain-Driven Design:
   - Services around business domains
   - Not technical layers

2. Database per Service:
   - Each service owns its data
```

```
   - No shared database

3. API Gateway:
   - Single entry point
   - Handle cross-cutting concerns

4. Service Discovery:
   - Dynamic service locations
   - Automatic registration

5. Distributed Tracing:
   - Track requests across services
   - Essential for debugging

6. Circuit Breakers:
   - Prevent cascading failures
   - Graceful degradation
```

---

18. Database Connection Pooling

What Is It?

Maintaining a cache of database connections that can be reused, avoiding the overhead of creating a new connection for each request.

The Problem

Without Connection Pooling:

```
For each request:
1. Create new database connection (10-100ms)
2. Execute query (10ms)
3. Close connection (5ms)

Total: 25-115ms (mostly connection overhead!)
```

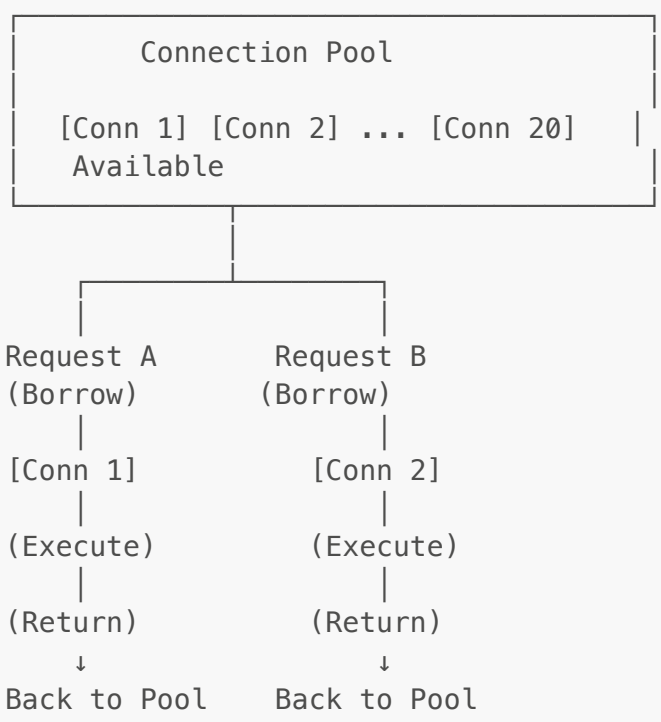
With Connection Pooling:

```
Startup:
- Create pool of 20 connections
- Keep connections open

For each request:
1. Borrow connection from pool (0.1ms)
2. Execute query (10ms)
3. Return connection to pool (0.1ms)
```

```
Total: 10.2ms (10x faster!)
```

How It Works



Configuration Parameters

Min Connections

```
Minimum connections to maintain

Example: min=5
- Always keep 5 connections open
- Ready for immediate use
- Trade-off: Resources vs response time

Use when: Predictable baseline load
```

Max Connections

```
Maximum connections allowed

Example: max=50
- Prevent resource exhaustion
- Queue requests if all busy
```

```
- Balance: Too low = waiting, too high = resource waste

Calculation:
max = (available_memory / connection_memory) * 0.8
```

Connection Timeout

```
How long to wait for available connection

Example: timeout=30s
- If no connection available in 30s → Error
- Prevents infinite waiting

Use when: Need to fail fast
```

Idle Timeout

```
How long to keep idle connection

Example: idle_timeout=300s (5 min)
- Close connections unused for 5 minutes
- Reduce resource usage
- Recreate when needed

Use when: Traffic varies (scale down during low traffic)
```

Max Lifetime

```
Maximum time connection can live

Example: max_lifetime=1800s (30 min)
- Refresh connections periodically
- Prevent stale connections
- Handle database restarts

Use when: Database behind load balancer, regular maintenance
```

Best Practices

```
1. Size pool appropriately:
   connections = (core_count * 2) + disk_count
```

```
   Example:
   4 cores + 2 disks = 10 connections

2. Monitor pool metrics:
   - Active connections
   - Idle connections
   - Wait time
   - Connection errors

3. Test connections:
   - Validate before use
   - Handle stale connections
   - Automatic retry

4. Environment-specific sizing:
   - Development: Small pool (5-10)
   - Production: Larger pool (20-100)
   - Load testing: Find optimal size
```

When to Use Connection Pooling

✅ **Always use for:**

- Web applications
- API servers
- Any high-traffic system
- Database connections
- HTTP client connections

❌ **Don't need when:**

- Single-threaded application
- Very low traffic (<1 RPS)
- Batch jobs (long-running)

Code Example

```python
# SQLAlchemy (Python)
from sqlalchemy import create_engine
from sqlalchemy.pool import QueuePool

engine = create_engine(
    'postgresql://user:password@localhost/db',
    poolclass=QueuePool,
    pool_size=20,          # Max connections
    max_overflow=10,       # Extra connections if needed
    pool_timeout=30,       # Wait time for connection
    pool_recycle=1800,     # Refresh after 30 min
    pool_pre_ping=True     # Test connection before use
)
```

```
# Usage - connection automatically returned
with engine.connect() as conn:
    result = conn.execute("SELECT * FROM users")

# HikariCP (Java) - Fastest connection pool
HikariConfig config = new HikariConfig();
config.setJdbcUrl("jdbc:postgresql://localhost/db");
config.setUsername("user");
config.setPassword("password");
config.setMaximumPoolSize(20);
config.setMinimumIdle(5);
config.setConnectionTimeout(30000);
config.setIdleTimeout(600000);
config.setMaxLifetime(1800000);

HikariDataSource ds = new HikariDataSource(config);
```

---

19. Heartbeat Mechanism

What Is It?

A periodic signal sent between systems to indicate they are still alive and functioning.

How It Works

```
[Client] ————heartbeat every 30s————→ [Server]

Timeline:
T0:  Client sends heartbeat
T30: Client sends heartbeat
T60: Client sends heartbeat
T90: No heartbeat received
     Server marks client as dead
```

Heartbeat Patterns

1. Client → Server Heartbeat

```
Use case: Connection keep-alive

Example: WebSocket connection
- Client sends ping every 30s
- Server responds with pong
- If no ping for 60s → Close connection
```

```
Implementation:
setInterval(() => {
    ws.send({type: 'heartbeat'});
}, 30000);
```

2. Server → Client Heartbeat

```
Use case: Leader notification

Example: Distributed system
- Leader sends heartbeat to followers
- If no heartbeat for 60s → Start election
- Ensures leader is alive

Implementation:
while (isLeader) {
    broadcast_to_followers({type: 'heartbeat'});
    sleep(30);
}
```

3. Bidirectional Heartbeat

```
Use case: Mutual monitoring

Both sides send heartbeats
Either can detect failure
Redundant detection

Use when: Critical systems, both sides need to know
```

Heartbeat vs Health Check

```
Heartbeat (Push):
- Service actively sends "I'm alive"
- Periodic signal
- Continuous monitoring

Health Check (Pull):
- External system queries "/health"
- On-demand check
- Used by load balancers

Comparison:
├── Heartbeat: Better for peers, distributed systems
└── Health Check: Better for client-server, load balancing
```

Timeout Strategies

Fixed Timeout

```
No heartbeat for 60 seconds → Declare dead

Simple but can cause false positives:
- Network hiccup
- Temporary GC pause
- Heavy load
```

Adaptive Timeout

```
Adjust timeout based on history

Normal response time: 30ms
Timeout = avg_response_time * 3 + buffer

If network slows down → Increase timeout
Reduces false positives
```

Phi Accrual Failure Detector

```
Used by Cassandra, Akka

Instead of binary (alive/dead):
- Calculate suspicion level (0-1)
- Gradual transition
- More sophisticated

phi = -log10(P(t_now - t_last_heartbeat))

if phi > threshold:
    mark_as_suspected()
```

When to Use Heartbeat

✅ **Use when:**

- Distributed systems (peer-to-peer)
- Long-lived connections (WebSocket)
- Leader election systems
- Cluster membership

- Need to detect failures quickly
- Both sides need liveness information

❌ **Don't use when:**

- Short-lived connections (HTTP requests)
- Load balancer handles health checks
- Central monitoring system exists
- Adds unnecessary network traffic

Real-World Examples

Kafka:

- Broker sends heartbeat to ZooKeeper
- Controller sends heartbeat to brokers
- Failure detection for rebalancing

Cassandra:

- Gossip protocol includes heartbeats
- Phi Accrual failure detector
- Dynamic cluster membership

Kubernetes:

- Kubelet sends heartbeat to control plane
- Node marked as NotReady if heartbeat fails
- Automatic pod rescheduling

Code Example

```
# Server-side heartbeat receiver
class HeartbeatMonitor:
    def __init__(self, timeout=60):
        self.timeout = timeout
        self.last_heartbeat = {}

    def receive_heartbeat(self, client_id):
        self.last_heartbeat[client_id] = time.time()

    def check_alive(self, client_id):
        if client_id not in self.last_heartbeat:
            return False

        elapsed = time.time() - self.last_heartbeat[client_id]
        return elapsed < self.timeout

    def get_dead_clients(self):
        now = time.time()
```

```
        dead = []
        for client_id, last_time in self.last_heartbeat.items():
            if now - last_time > self.timeout:
                dead.append(client_id)
        return dead

# Client-side heartbeat sender
class HeartbeatSender:
    def __init__(self, server_url, interval=30):
        self.server_url = server_url
        self.interval = interval
        self.running = False

    def start(self):
        self.running = True
        threading.Thread(target=self._send_loop).start()

    def _send_loop(self):
        while self.running:
            try:
                requests.post(f"{self.server_url}/heartbeat",
                              json={"client_id": self.client_id})
            except Exception as e:
                print(f"Heartbeat failed: {e}")

            time.sleep(self.interval)

    def stop(self):
        self.running = False

# Usage
monitor = HeartbeatMonitor(timeout=60)
monitor.receive_heartbeat("client123")

# Check periodically
if not monitor.check_alive("client123"):
    handle_client_failure("client123")
```

---

20. Checksum & CRC

What Is It?

Algorithms to detect errors in data transmission or storage by computing a small fixed-size value from the data.

How It Works

```
Sender:
1. Compute checksum of data
```

```
2. Send data + checksum

Receiver:
1. Receive data + checksum
2. Compute checksum of received data
3. Compare with sent checksum
4. If match: Data is intact
5. If mismatch: Data corrupted
```

Common Algorithms

1. Simple Checksum (Addition)

```
Data: [5, 10, 15, 20]
Checksum: (5 + 10 + 15 + 20) % 256 = 50

Pros:
✓ Simple to compute
✓ Very fast

Cons:
✗ Weak (doesn't detect many errors)
✗ Order-independent (5+10 = 10+5)

Use when: Speed > accuracy, internal use only
```

2. CRC (Cyclic Redundancy Check)

```
CRC-32: 32-bit value
Uses polynomial division

Example:
Data: "Hello World"
CRC-32: 0x4A17B156

Pros:
✓ Detects most errors
✓ Fast to compute
✓ Hardware support

Cons:
✗ Not cryptographically secure
✗ Collisions possible

Use when: Data integrity (files, network packets)
```

3. MD5 Hash

```
128-bit hash value

Example:
Data: "Hello World"
MD5: b10a8db164e0754105b7a99be72e3fe5

Pros:
✓ Strong error detection
✓ Fixed size output
✓ Widely supported

Cons:
✗ Cryptographically broken
✗ Slower than CRC

Use when: File integrity, not security (use SHA for security)
```

4. SHA (Secure Hash)

```
SHA-256: 256-bit hash

Example:
Data: "Hello World"
SHA-256:
a591a6d40bf420404a011733cfb7b190d62c65bf0bcda32b57b277d9ad9f146e

Pros:
✓ Cryptographically secure
✓ No known collisions
✓ Tamper detection

Cons:
✗ Slower to compute
✗ Larger output

Use when: Security critical, tamper detection
```

Use Cases

1. Data Transmission

```
Network packet:
┌──────────┬──────────┐
│  Data    │ Checksum │
```

```
└──────────┴──────────┘

If packet corrupted:
- Checksum mismatch
- Request retransmission
- Ensures data integrity
```

2. File Integrity

```
Download file:
1. Download file.zip
2. Download file.zip.md5
3. Compute MD5 of downloaded file
4. Compare with file.zip.md5
5. If match: File intact
6. If mismatch: File corrupted, re-download

Example: Linux ISO downloads
```

3. Data Storage

```
Database:
- Store data + checksum
- Periodically verify checksums
- Detect disk corruption
- Alert if mismatch found

Example: Cassandra uses CRC for SSTable validation
```

4. Distributed Systems

```
Message replication:
- Compute checksum at source
- Send message + checksum
- Verify on all replicas
- Ensures consistent replication

Example: Kafka message integrity
```

5. Cache Validation

```
Redis cache entry:
key: user:123
value: {data}
checksum: abc123

Before returning cached data:
1. Compute checksum of cached data
2. Compare with stored checksum
3. If match: Return data
4. If mismatch: Cache corrupted, fetch from DB

Prevents serving corrupted cache data
```

When to Use Checksum/CRC

✅ **Use when:**

- Transmitting data over network
- Storing data long-term
- Need to detect corruption
- File downloads/uploads
- Distributed replication
- Cache integrity verification

❌ **Don't use when:**

- Data in-memory only (temporary)
- Performance is critical (checksums add overhead)
- Errors impossible (trusted source)
- Security needed (use cryptographic hash instead)

Code Examples

```python
# CRC-32
import zlib

def calculate_crc32(data):
    return zlib.crc32(data.encode())

data = "Hello World"
checksum = calculate_crc32(data)
print(f"CRC32: {checksum}")  # 1243066710

# Verify
received_data = "Hello World"
received_checksum = 1243066710

if calculate_crc32(received_data) == received_checksum:
```

```
    print("Data intact")
else:
    print("Data corrupted!")

# MD5
import hashlib

def calculate_md5(data):
    return hashlib.md5(data.encode()).hexdigest()

data = "Hello World"
md5_hash = calculate_md5(data)
print(f"MD5: {md5_hash}")  # b10a8db164e0754105b7a99be72e3fe5

# SHA-256
def calculate_sha256(data):
    return hashlib.sha256(data.encode()).hexdigest()

data = "Hello World"
sha_hash = calculate_sha256(data)
print(f"SHA-256: {sha_hash}")
```

Performance Comparison

```
Algorithm    Speed        Strength      Output Size   Use Case
──────────────────────────────────────────────────────────────
Simple Sum   Fastest      Weakest       8-32 bits     Internal only
CRC-32       Very Fast    Good          32 bits       Data integrity
MD5          Fast         Weak (broken) 128 bits      Legacy, non-
security
SHA-256      Moderate     Strong        256 bits      Security, tamper
detection
SHA-512      Slower       Very Strong   512 bits      High security

Recommendation:
- Data integrity: CRC-32
- File verification: SHA-256
- Security: SHA-256 or SHA-512
- Performance critical: CRC-32
```

Summary & Quick Reference

Decision Matrix

Scenario	Recommended Concept
Multiple servers, need traffic distribution	Load Balancer

Scenario	Recommended Concept
Microservices, need single entry point	API Gateway
Static content, global users	CDN
Corporate network control	Forward Proxy
Hide backend servers	Reverse Proxy
Frequently accessed data	Caching
Single DB too small	Database Sharding
Read-heavy workload	Database Replication
Slow queries	Database Indexing
Async processing needed	Message Queue
Event broadcasting	Pub/Sub Pattern
Prevent API abuse	Rate Limiting
Dynamic server pool	Consistent Hashing
Distributed system design	CAP Theorem
Cluster coordination	Distributed Consensus
Prevent cascading failures	Circuit Breaker
Fast set membership check	Bloom Filter
Large application, independent teams	Microservices
Database performance	Connection Pooling
Detect node failures	Heartbeat
Verify data integrity	Checksum/CRC

Concept Combinations

High-Traffic Web Application:

```
1. Load Balancer → Distribute traffic
2. API Gateway → Centralized auth/routing
3. Caching (Redis) → Reduce DB load
4. Database Replication → Scale reads
5. Database Indexing → Fast queries
6. Message Queue → Async processing
7. Rate Limiting → Prevent abuse
8. Connection Pooling → Efficient DB access
```

Global Video Streaming:

```
1. CDN → Fast content delivery globally
2. Load Balancer → Distribute API traffic
3. Caching → Reduce origin load
4. Database Sharding → Scale storage
5. Consistent Hashing → CDN node selection
6. Checksum → Verify video integrity
7. Heartbeat → Monitor CDN node health
```

Microservices Platform:

```
1. API Gateway → Single entry point
2. Load Balancer → Per-service scaling
3. Message Queue → Async communication
4. Pub/Sub → Event-driven architecture
5. Circuit Breaker → Fault isolation
6. Distributed Consensus → Leader election
7. Database per Service → Data ownership
8. Caching → Reduce latency
```

Performance Impact Comparison

Concept	Latency Impact	Throughput Impact	When to Apply
Caching	-90% (100ms → 10ms)	+10x	Read-heavy
Indexing	-99% (1s → 10ms)	+100x	Large tables
CDN	-80% (200ms → 40ms)	+5x	Static assets
Load Balancer	+5ms	+Nx (N servers)	>1 server
Replication	Reads: -50%	Reads: +Nx	Read-heavy
Sharding	±0	Writes: +Nx	Write-heavy
Connection Pool	-90% (100ms → 10ms)	+10x	DB-heavy
Message Queue	Async (no wait)	Handle spikes	Background jobs

Cost-Benefit Analysis

```
Low-Hanging Fruit (Easy wins):
1. Caching: 10-100x performance, low complexity
2. Database Indexing: 100-1000x for slow queries, trivial to add
3. Connection Pooling: 5-10x performance, one config change
4. CDN: 70% cost savings + better performance
```

```
Medium Complexity:
5. Load Balancer: High availability, moderate setup
6. Database Replication: 3-5x read capacity, manage lag
7. Rate Limiting: Protect resources, some complexity
8. Message Queue: Decouple services, new infrastructure

High Complexity:
9. Database Sharding: 10x capacity, high complexity
10. Microservices: Ultimate scaling, high operational cost
11. Distributed Consensus: Strong guarantees, complex
12. Consistent Hashing: Smooth scaling, algorithmic complexity
```

Common Mistakes to Avoid

```
❌ Premature Optimization
- Don't add sharding when single DB works
- Don't use microservices for small app
- Start simple, scale when needed

❌ Over-Engineering
- Don't use all 20 concepts at once
- Choose what solves your problem
- Complexity has a cost

❌ Wrong Tool for Job
- Don't use Bloom filter when false positives unacceptable
- Don't use AP system for banking
- Don't cache rapidly changing data

❌ Ignoring Trade-offs
- Every concept has pros/cons
- Understand what you're sacrificing
- Document decisions

❌ Not Monitoring
- Add metrics for every component
- Monitor cache hit rates
- Alert on circuit breaker opens
- Track replication lag
```

Interview Strategy

When Asked "Design [System]":

```
1. Clarify Requirements (5 min)
   - Scale (users, requests, data)
   - Performance needs
```

```
   - Consistency requirements

2. High-Level Design (10 min)
   - Draw architecture
   - Identify these 20 concepts where applicable
   - Explain why each is needed

3. Deep Dive (20 min)
   - Pick 2-3 concepts interviewer interested in
   - Explain trade-offs
   - Discuss alternatives

4. Bottlenecks & Scale (10 min)
   - Identify bottlenecks
   - How to scale each component
   - Cost considerations
```

Talking Points:

```
Always mention:
✓ "We'll use a load balancer for high availability"
✓ "Caching will reduce database load by 90%"
✓ "Rate limiting prevents abuse"
✓ "CDN reduces latency for global users"

Explain trade-offs:
✓ "Sharding improves write throughput but complicates queries"
✓ "We choose AP over CP because availability matters more"
✓ "Microservices add complexity but enable independent scaling"

Reference real systems:
✓ "Similar to how Netflix uses..."
✓ "Instagram shards by user_id..."
✓ "Following the pattern used by Uber..."
```

Cheat Sheet

Quick Decision Tree

```
1. Need to distribute traffic?
   → Load Balancer

2. Have microservices?
   → API Gateway
   → Message Queue
   → Circuit Breaker
   → Service Discovery
```

```
3. Serving static content globally?
   → CDN
   → Caching

4. Database slow?
   → Check: Indexing
   → Read-heavy: Replication
   → Write-heavy: Sharding
   → Frequently accessed: Caching

5. Need to prevent abuse?
   → Rate Limiting
   → Authentication

6. Scaling horizontally?
   → Consistent Hashing (if dynamic)
   → Database Sharding (if needed)

7. Distributed system?
   → CAP Theorem (choose CP or AP)
   → Consensus (if coordination needed)
   → Heartbeat (failure detection)

8. Data integrity?
   → Checksum/CRC
```

Key Formulas

```
Load Balancer:
servers_needed = peak_rps / server_capacity

Caching:
memory_needed = hot_data_size * 1.2 (overhead)
cost_savings = (1 - cache_hit_ratio) * db_cost

Sharding:
shards_needed = total_data_size / shard_capacity
shard_id = hash(key) % num_shards

Replication:
read_capacity = replicas * single_server_capacity

Indexing:
query_improvement = table_size / index_depth
write_overhead = num_indexes * 0.2

Connection Pool:
pool_size = (cores * 2) + disks
```

```
Rate Limiting:
requests_per_window = limit * window_size_seconds

Bloom Filter:
bits_needed = -n * ln(p) / (ln(2)^2)
false_positive_rate = (1 - e^(-kn/m))^k
```

Resources for Further Learning

Books

1. **"Designing Data-Intensive Applications"** by Martin Kleppmann
    - Deep dive into all concepts
    - Must-read for system design
2. **"System Design Interview"** by Alex Xu (Volume 1 & 2)
    - Interview-focused
    - Real-world examples
3. **"Building Microservices"** by Sam Newman
    - Comprehensive microservices guide
4. **"Site Reliability Engineering"** by Google
    - Production systems best practices

Online Resources

ByteByteGo (Alex Xu)

- System design videos and diagrams
- Clear visual explanations
- Real-world examples

High Scalability Blog

- Architecture of real systems
- Netflix, Uber, Facebook, etc.

Martin Fowler's Blog

- Software architecture patterns
- Microservices insights

AWS Architecture Blog

- Cloud-native patterns
- Scalability strategies

Practice Platforms

1. **LeetCode System Design**
    - Interview questions
    - Community solutions
2. **System Design Primer (GitHub)**
    - Comprehensive guide
    - Links to resources
3. **Grokking the System Design Interview**
    - Step-by-step walkthroughs
    - Common patterns

Document Version: 1.0
Last Updated: January 9, 2025
Author: System Design Interview Prep
Source: ByteByteGo-inspired with detailed explanations
Status: Complete & Interview-Ready ✅

Key Takeaways:

1. **Load Balancer** - First step in scaling horizontally
2. **API Gateway** - Essential for microservices architecture
3. **CDN** - 70% cost savings + 20x faster global delivery
4. **Caching** - 90% database load reduction with proper strategy
5. **Database Sharding** - Linear scaling but adds complexity
6. **Database Replication** - Scale reads, high availability
7. **Indexing** - 1000x query speed improvement
8. **Message Queue** - Decouple services, handle async work
9. **Pub/Sub** - Event-driven architecture, loose coupling
10. **Rate Limiting** - Protect resources, prevent abuse
11. **Consistent Hashing** - Smooth scaling, minimal data movement
12. **CAP Theorem** - Choose CP or AP based on requirements
13. **Consensus** - Coordination in distributed systems
14. **Circuit Breaker** - Prevent cascading failures
15. **Bloom Filter** - Memory-efficient set membership
16. **Microservices** - Independent scaling and deployment
17. **Connection Pooling** - 10x database performance
18. **Heartbeat** - Detect failures in distributed systems
19. **Checksum/CRC** - Data integrity verification

Remember: There's no one-size-fits-all solution. Understand the trade-offs, choose what solves YOUR specific problem, and be able to explain WHY you made each decision.

Good luck with your system design interview! 🚀