

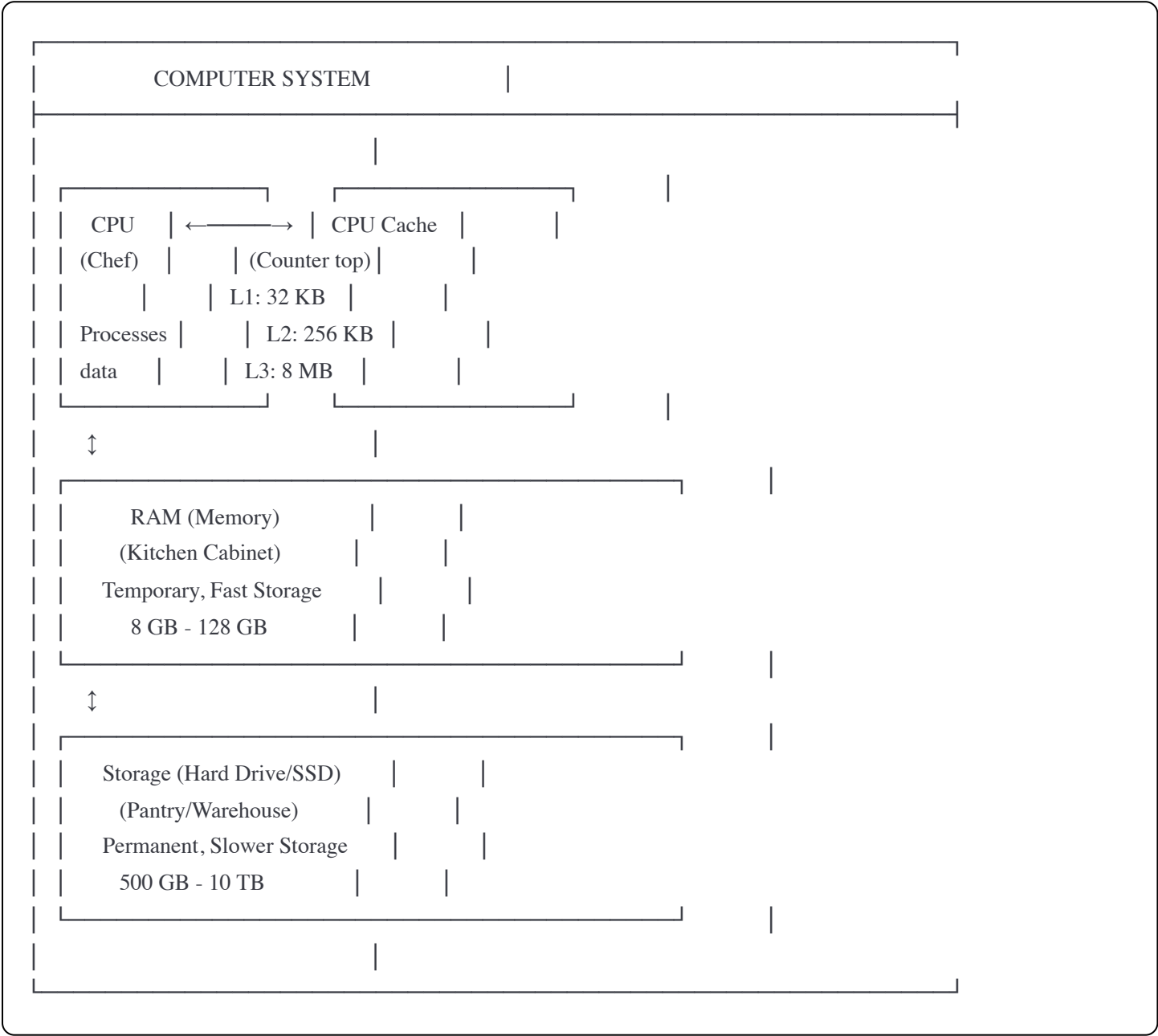
Chapter 2: Computer Architecture Basics

1. How Computers Work: CPU, Memory, Storage

Understanding computer architecture is crucial for system design because **performance bottlenecks** often come from these components.

The Computer Hierarchy

Think of a computer like a kitchen:



CPU (Central Processing Unit) - The Brain

What it does: Executes instructions and performs calculations

Key Characteristics:

- **Clock Speed:** How fast it runs (e.g., 3.5 GHz = 3.5 billion cycles/second)
- **Cores:** Number of independent processing units (2, 4, 8, 16+ cores)
- **Threads:** Virtual cores that allow parallel task execution

Real-World Example:

Single-Core CPU (Old computer):

Task 1: Process image [████████████████████] 10 seconds

Task 2: Compress video [████████████████████] 10 seconds

Total: 20 seconds (sequential)

Quad-Core CPU (Modern computer):

Task 1: Process image [██████] 2.5 seconds ← Core 1

Task 2: Compress video [██████] 2.5 seconds ← Core 2

Task 3: Download file [██████] 2.5 seconds ← Core 3

Task 4: Run antivirus [██████] 2.5 seconds ← Core 4

Total: 2.5 seconds (parallel)

System Design Implication:

```
python

# CPU-bound operation (heavy computation)
def calculate_fibonacci(n):
    """This needs CPU power"""
    if n <= 1:
        return n
    return calculate_fibonacci(n-1) + calculate_fibonacci(n-2)

# For system design:
# - CPU-intensive tasks need more cores
# - Consider using worker servers for heavy computations
# - Examples: Video encoding, image processing, machine learning
```

Memory (RAM) - Short-Term Memory

What it does: Stores data that's currently being used

Characteristics:

- **Fast access** (nanoseconds)
- **Volatile** (data disappears when power is off)
- **Limited size** (8 GB - 128 GB typical)

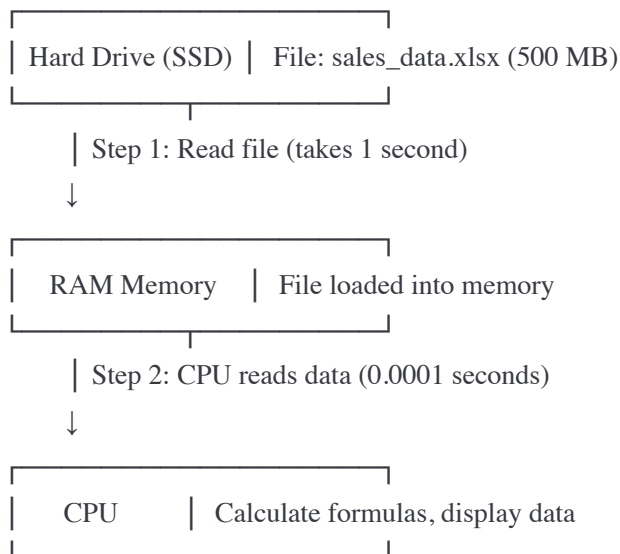
- **Expensive** compared to storage

The Memory Access Pattern:

Program Execution:

1. Load program from storage → RAM
2. CPU reads data from RAM
3. CPU processes data
4. CPU writes results back to RAM
5. Eventually save to storage

Example: Opening a large Excel file



Why RAM Matters in System Design:

Scenario: Web Application Server

With 8 GB RAM:

- Can handle ~200 concurrent users
- Each user session: ~40 MB
- $200 \times 40 \text{ MB} = 8 \text{ GB}$

With 32 GB RAM:

- Can handle ~800 concurrent users
- Same calculation

If you exceed RAM:

- System uses "swap" (hard drive as fake RAM)
- Performance drops by 100-1000x
- Users experience timeouts

Code Example - Memory Usage:

```
python

import sys

# Small data structure
small_list = [1, 2, 3, 4, 5]
print(f"Small list memory: {sys.getsizeof(small_list)} bytes")
# Output: ~120 bytes

# Large data structure
large_list = list(range(1000000))
print(f"Large list memory: {sys.getsizeof(large_list)} bytes")
# Output: ~8 MB

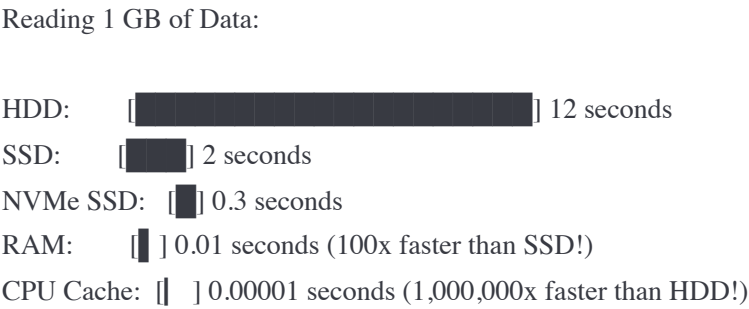
# System design consideration:
# If each user session holds 8 MB of data,
# and you have 1,000 concurrent users,
# you need 8 GB RAM just for user data!
```

Storage (Hard Drive/SSD) - Long-Term Memory

Types of Storage:

Type	Speed	Cost	Use Case
HDD (Hard Disk Drive)	80-160 MB/s	\$0.02/GB	Cheap bulk storage
SSD (Solid State Drive)	500-3,500 MB/s	\$0.10/GB	OS, databases, apps
NVMe SSD	3,000-7,000 MB/s	\$0.15/GB	High-performance DBs
Cloud Storage (S3)	Varies	\$0.023/GB/month	Scalable, redundant

Visual Comparison:



Real-World System Design Example:

E-commerce Database Design:

Option 1: All on HDD

- 10,000 product queries/second
- Each query: 10ms (slow)
- Users complain: "Site is slow!"
- Monthly cost: \$100

Option 2: Database on SSD

- 10,000 product queries/second
- Each query: 2ms (fast)
- Users happy: "Site is fast!"
- Monthly cost: \$300

Option 3: Hot data in RAM (Redis Cache)

- 9,000 queries hit cache: 0.1ms
- 1,000 queries hit SSD: 2ms
- Average: 0.29ms (super fast!)
- Monthly cost: \$400
- Best user experience!

Performance Numbers Every Programmer Should Know

These are **approximate** but crucial for system design:

L1 cache reference	0.5 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
Mutex lock/unlock	25 ns
Main memory reference	100 ns
Compress 1KB with Zippy	3,000 ns (3 μ s)
Send 1KB over 1 Gbps network	10,000 ns (10 μ s)
Read 4KB randomly from SSD	150,000 ns (150 μ s)
Read 1 MB sequentially from memory	250,000 ns (250 μ s)
Round trip within same datacenter	500,000 ns (0.5 ms)
Read 1 MB sequentially from SSD	1,000,000 ns (1 ms)
Disk seek	10,000,000 ns (10 ms)
Read 1 MB sequentially from disk	20,000,000 ns (20 ms)
Send packet CA→Netherlands→CA	150,000,000 ns (150 ms)

What This Means:

- **Memory is ~100x faster than SSD**
- **SSD is ~100x faster than HDD**

- **Network calls are expensive** (0.5ms in same datacenter, 150ms across continents)

2. Understanding Latency and Throughput

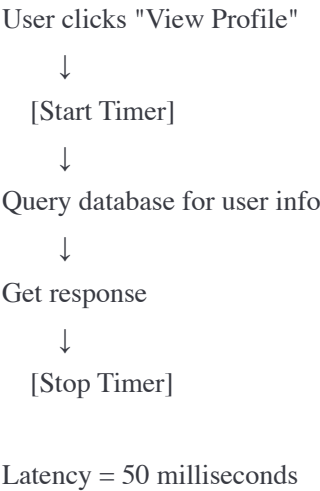
These are the two most important performance metrics in system design.

Latency - "How long does it take?"

Definition: The time it takes to complete a single operation

Analogy: Latency is like the time it takes for a single car to drive from point A to B.

Example: Database Query Latency



Real-World Latency Targets:

Application Type	Target Latency	User Experience
Search engine results	< 100 ms	Feels instant
Page load	< 1 second	Good
Button click response	< 100 ms	Responsive
Video stream start	< 2 seconds	Acceptable
API response	< 200 ms	Smooth
Database query	< 10 ms	Efficient

Throughput - "How many per second?"

Definition: The number of operations completed per unit of time

Analogy: Throughput is like how many cars can pass through a highway per hour.

Example: Web Server Throughput

Server can handle:

- └ 1,000 requests per second (low traffic)
- └ 10,000 requests per second (medium traffic)
- └ 100,000 requests per second (high traffic)

Measuring Throughput:

```
python

import time

def measure_throughput(duration_seconds=10):
    """Measure how many operations we can do per second"""
    start_time = time.time()
    operations = 0

    while time.time() - start_time < duration_seconds:
        # Simulate operation (e.g., database query)
        result = sum(range(1000))
        operations += 1

    elapsed = time.time() - start_time
    throughput = operations / elapsed

    print(f"Completed {operations} operations in {elapsed:.2f} seconds")
    print(f"Throughput: {throughput:.0f} operations/second")

# Example output:
# Completed 1,547,832 operations in 10.00 seconds
# Throughput: 154,783 operations/second
```

The Latency-Throughput Relationship

Important Concept: Low latency doesn't always mean high throughput!

Scenario 1: Fast Food Restaurant (Low Latency, Low Throughput)

One cashier serves customers very quickly:

- Latency per customer: 30 seconds (fast!)
- Throughput: 2 customers/minute (only 1 cashier)

Scenario 2: Buffet Restaurant (Higher Latency, High Throughput)

Multiple stations serve customers:

- Latency per customer: 5 minutes (slower)
- Throughput: 60 customers/minute (many parallel stations)

System Design Application:

Web Server Example:

Design A: Single powerful server

Mega Server			
32 cores	Latency:	50ms per request	
128 GB RAM	Throughput:	1,000 requests/sec	

Design B: Multiple smaller servers

Server 1		Server 2		Server 3		Server 4	
8 cores	8 cores	8 cores	8 cores	8 cores	8 cores	8 cores	8 cores
32GB RAM	32GB RAM	32GB RAM	32GB RAM	32GB RAM	32GB RAM	32GB RAM	32GB RAM

Latency: 60ms per request (slightly slower)

Throughput: 4,000 requests/sec (4x higher!)

Cost: Similar

Benefit: If one fails, others keep running!

Key Insight for System Design:

- For **user-facing** features: Optimize for **latency** (users feel speed)
- For **batch processing**: Optimize for **throughput** (process more data)
- Often need to balance both

Practical Example: Image Processing Service

```
python

# Scenario: Process 1,000 images

# Approach 1: Sequential (optimize for latency per image)
def process_images_sequential(images):
    results = []
    for image in images:
        result = process_image(image) # 100ms per image
        results.append(result)
    return results

# Time: 1,000 images × 100ms = 100 seconds
# Throughput: 10 images/second

# Approach 2: Parallel (optimize for throughput)
from concurrent.futures import ThreadPoolExecutor

def process_images_parallel(images):
    with ThreadPoolExecutor(max_workers=10) as executor:
        results = executor.map(process_image, images)
    return list(results)

# Time: 1,000 images ÷ 10 workers × 100ms = 10 seconds
# Throughput: 100 images/second
# Latency per image: Still 100ms, but overall much faster!
```

3. Network Basics

Networks are the backbone of distributed systems. Understanding them is crucial for system design.

Bandwidth - "How wide is the pipe?"

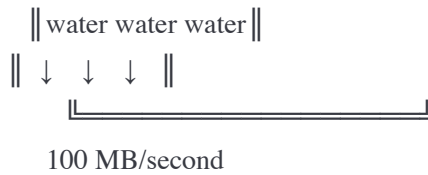
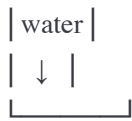
Definition: The maximum amount of data that can be transferred per second

Analogy: Water pipe width

Pipe Analogy:

Small Pipe (Low Bandwidth)

Large Pipe (High Bandwidth)



Common Bandwidth Speeds:

Connection Type	Bandwidth	Download 1 GB File
Dial-up (ancient)	56 Kbps	4 hours
DSL (old)	10 Mbps	13 minutes
Cable Internet	100 Mbps	1.3 minutes
Fiber Internet	1 Gbps	8 seconds
Data Center Link	10 Gbps	0.8 seconds
Server to Server (same)	100 Gbps	0.08 seconds

System Design Implication:

Video Streaming Service:

User has 5 Mbps connection:

- Can stream 720p (3 Mbps) ✓
- Cannot stream 4K (25 Mbps) ✗

Solution: Adaptive Bitrate Streaming

Detect user's bandwidth	
Serve appropriate quality:	
- High bandwidth → 4K	
- Medium bandwidth → 1080p	
- Low bandwidth → 480p	

Network Protocols

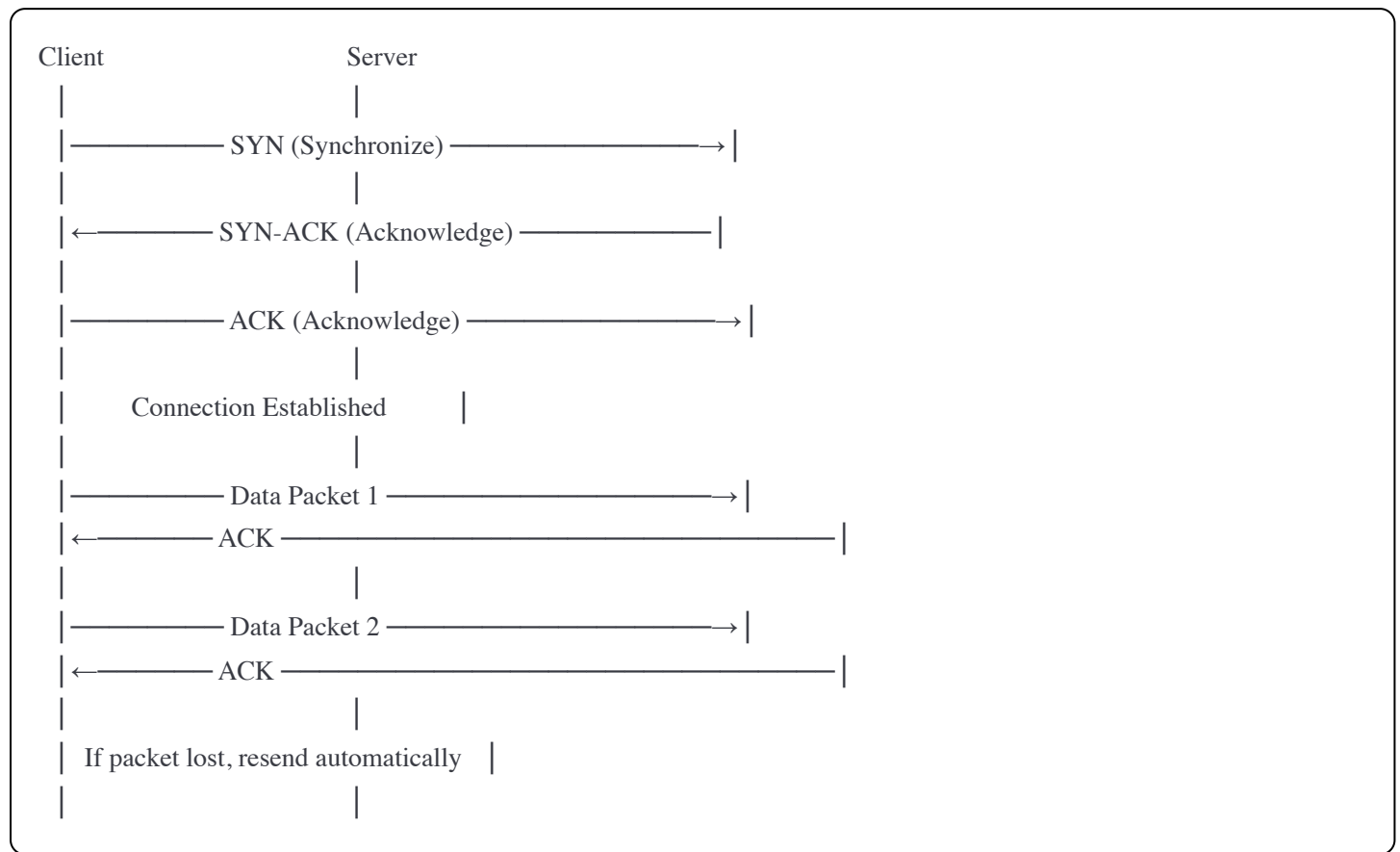
TCP/IP - The Foundation of the Internet

TCP (Transmission Control Protocol):

- **Reliable:** Guarantees data delivery

- **Ordered:** Data arrives in correct sequence
- **Error-checked:** Detects and fixes errors
- **Slower:** Because of all these guarantees

How TCP Works:



UDP (User Datagram Protocol):

- **Fast:** No handshake, just send
- **Unreliable:** No delivery guarantee
- **No ordering:** Packets may arrive out of order
- **Use case:** Video streaming, gaming, DNS lookups

TCP vs UDP Comparison:

Sending "HELLO WORLD":

TCP:

1. Establish connection (3-way handshake)
2. Send "HELLO"
3. Wait for acknowledgment
4. Send "WORLD"
5. Wait for acknowledgment
6. Close connection

Total: ~100ms

UDP:

1. Send "HELLO WORLD"

Total: ~10ms

But with UDP:

- Might arrive as "HELLO WROLD" (wrong order)
- Might lose "WORLD" entirely
- Faster but less reliable!

HTTP - How Web Works

HTTP (HyperText Transfer Protocol):

- Built on top of TCP
- Request-response model
- Stateless (each request is independent)

HTTP Request Structure:

GET /api/users/123 HTTP/1.1

Host: api.example.com

User-Agent: Mozilla/5.0

Accept: application/json

Authorization: Bearer eyJhbGc...

(Request Body for POST/PUT)

HTTP Response Structure:

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 145
Cache-Control: max-age=3600
```

```
{
  "id": 123,
  "name": "John Doe",
  "email": "john@example.com"
}
```

HTTP Methods:

Method	Purpose	Example
GET	Retrieve data	Get user profile
POST	Create new data	Create new user
PUT	Update entire data	Update user info
PATCH	Update partial data	Change user email
DELETE	Remove data	Delete user

HTTP Status Codes:

Code	Meaning	Example
200	OK	Success!
201	Created	New user created
400	Bad Request	Invalid input
401	Unauthorized	Login required
403	Forbidden	No permission
404	Not Found	Page doesn't exist
500	Internal Server Error	Server crashed
503	Service Unavailable	Server overloaded

Real Code Example:

```
python
```

```
import requests

# GET request
response = requests.get('https://api.example.com/users/123')
print(f"Status: {response.status_code}")
print(f>Data: {response.json()}")

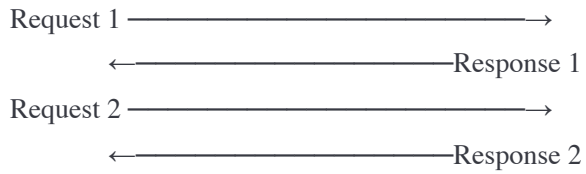
# POST request
new_user = {
    "name": "Jane Smith",
    "email": "jane@example.com"
}
response = requests.post(
    'https://api.example.com/users',
    json=new_user,
    headers={'Authorization': 'Bearer token123'}
)

# System Design Note:
# Each HTTP request has overhead:
# - DNS lookup: 20-100ms
# - TCP handshake: 50-200ms
# - TLS handshake: 100-300ms
# - Request/response: 50-500ms
# Total: 220-1,100ms for ONE request!
```

HTTP/1.1 vs HTTP/2 vs HTTP/3:

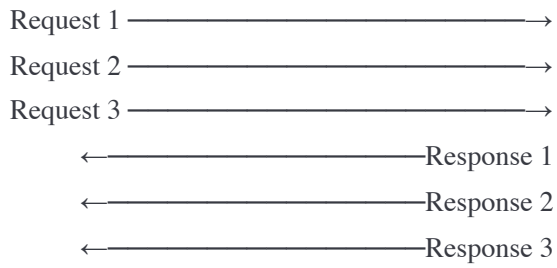
HTTP/1.1 (Old):

- One request at a time per connection
- Multiple connections needed (6-8)
- Text-based protocol



HTTP/2 (Modern):

- Multiple requests simultaneously
- Binary protocol (faster)
- Server push capability



HTTP/3 (Latest):

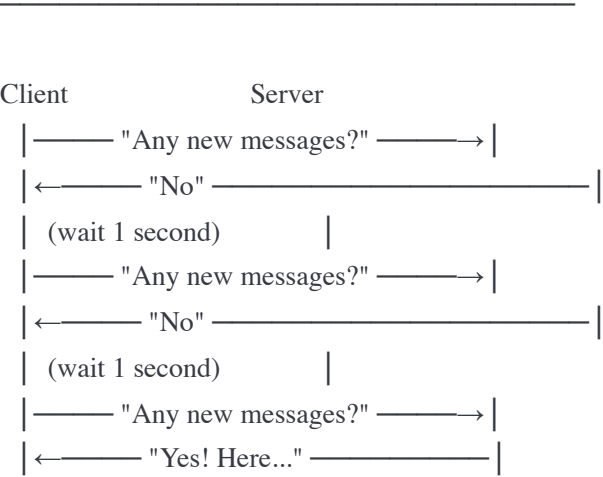
- Built on UDP (QUIC)
- Even faster
- Better mobile performance

WebSockets - Real-Time Communication

Problem with HTTP: Request-response only. Server can't push data to client.

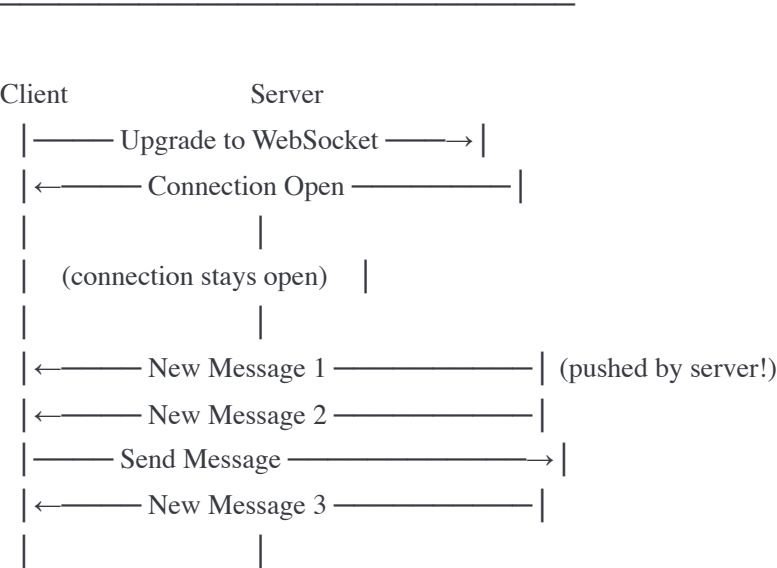
WebSocket Solution: Persistent, bidirectional connection

HTTP (Polling - inefficient):



Problem: Wasteful! Many unnecessary requests.

WebSocket (efficient):



WebSocket Code Example:

javascript

// Client-side JavaScript

```
const socket = new WebSocket('wss://chat.example.com');
```

// Connection opened

```
socket.onopen = function(event) {  
  console.log('Connected to chat server');  
  socket.send(JSON.stringify({  
    type: 'join',  
    username: 'John'  
  }));  
};
```

// Receive messages

```
socket.onmessage = function(event) {  
  const data = JSON.parse(event.data);  
  console.log('New message:', data.message);  
  displayMessage(data);  
};
```

// Send message

```
function sendMessage(text) {  
  socket.send(JSON.stringify({  
    type: 'message',  
    text: text,  
    timestamp: Date.now()  
  }));  
}
```

// Connection closed

```
socket.onclose = function(event) {  
  console.log('Disconnected from server');  
};
```

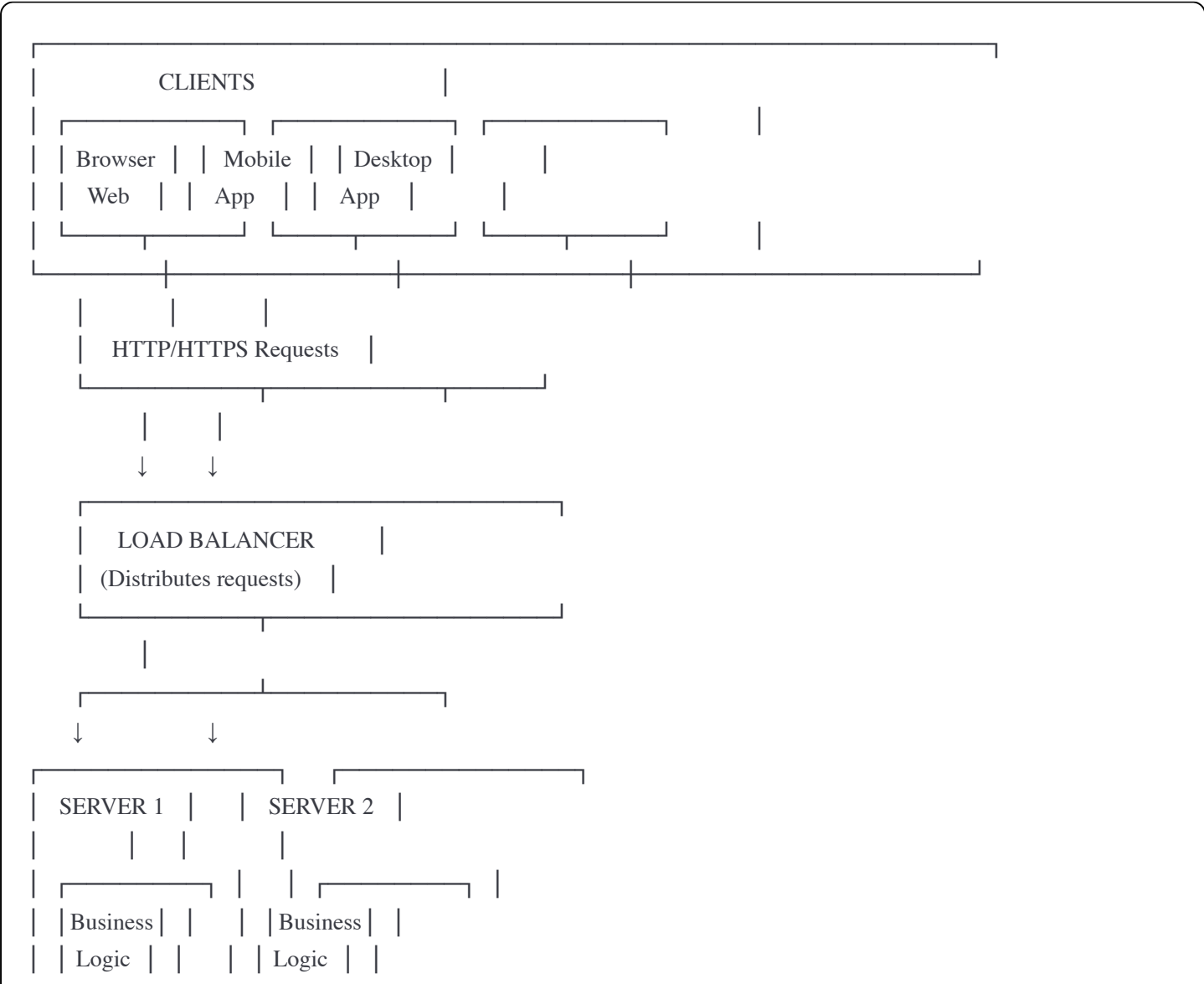
When to Use What:

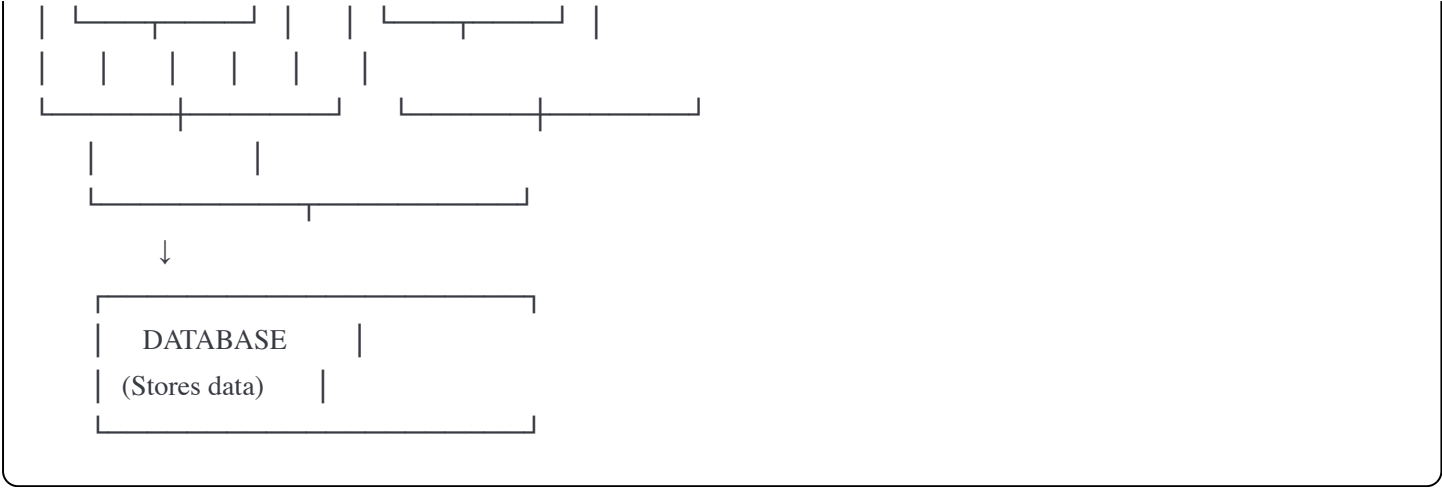
Protocol	Use Case	Example
HTTP	Standard web requests	Loading a webpage REST API calls
WebSocket	Real-time, 2-way Low latency	Chat applications Live sports scores Collaborative editing Stock price updates
SSE (Server-Sent Events)	Server to client only Simpler than WebSocket	News feeds Notifications Live blog updates

4. Client-Server Architecture

The foundation of most web applications.

Basic Client-Server Model

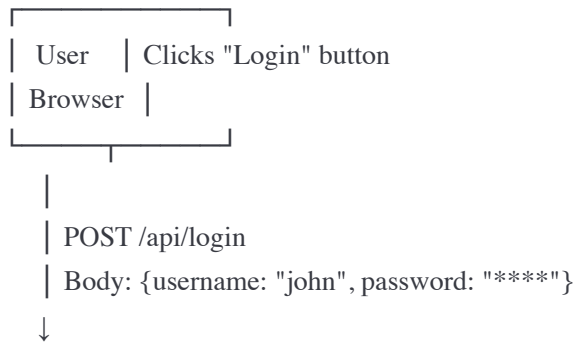




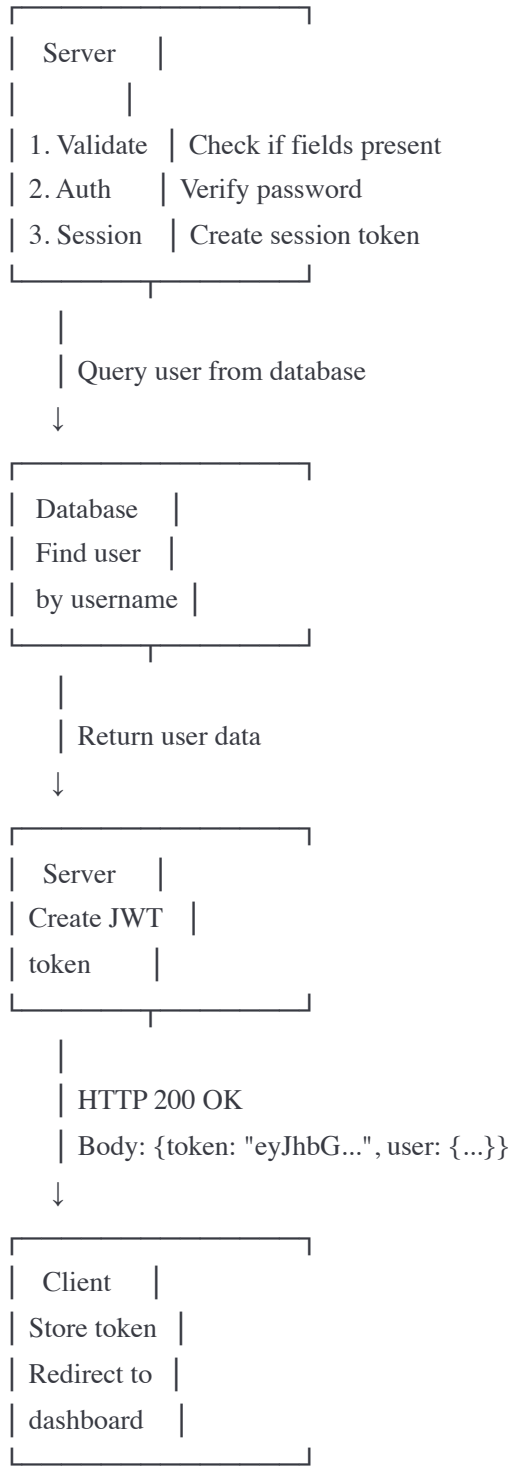
Request-Response Flow

Step-by-Step Example: User Logs In

Step 1: Client initiates request



Step 2: Request reaches server



Total time: ~100-300ms

Three-Tier Architecture

Most modern web applications use this structure:

PRESENTATION TIER
(Client Side)

HTML/CSS/JavaScript
React, Vue, Angular
Mobile: iOS, Android

Responsibilities:

- Display data
- Handle user input
- Client-side validation

HTTP/HTTPS



APPLICATION TIER
(Server Side)

Business Logic
Node.js, Python, Java, Go
REST APIs, GraphQL

Responsibilities:

- Process requests
- Business rules
- Authentication/Authorization
- Data transformation

SQL/NoSQL



DATA TIER
(Storage Layer)

Database (PostgreSQL, MySQL)
Cache (Redis, Memcached)
File Storage (S3, CDN)

Responsibilities:

- Store data persistently
- Data integrity

- Efficient querying

Complete Real-World Example: E-Commerce Product Page

python

```
# APPLICATION TIER - Python/Flask Server
```

```
from flask import Flask, jsonify, request
import redis
import psycopg2
```

```
app = Flask(__name__)
```

```
# Redis cache connection
```

```
cache = redis.Redis(host='localhost', port=6379)
```

```
# Database connection
```

```
db = psycopg2.connect(
    host="localhost",
    database="ecommerce",
    user="admin",
    password="password"
)
```

```
@app.route('/api/products/<product_id>')
```

```
def get_product(product_id):
```

```
    """
```

```
    Get product details
```

```
    Flow: Check cache → If miss, query DB → Update cache → Return
```

```
    """
```

```
# Step 1: Try cache first (fast!)
```

```
cached_product = cache.get(f'product:{product_id}')
```

```
if cached_product:
```

```
    print(f"Cache HIT for product {product_id}")
```

```
    return jsonify(json.loads(cached_product))
```

```
print(f"Cache MISS for product {product_id}")
```

```
# Step 2: Query database (slower)
```

```
cursor = db.cursor()
```

```
cursor.execute("""
```

```
    SELECT id, name, price, description, stock
```

```
    FROM products
```

```
    WHERE id = %s
```

```
""", (product_id,))
```

```
row = cursor.fetchone()
```

```
if not row:
```

```
    return jsonify({"error": "Product not found"}), 404
```

Step 3: Format response

```
product = {  
    "id": row[0],  
    "name": row[1],  
    "price": row[2],  
    "description": row[3],  
    "stock": row[4]  
}
```

Step 4: Store in cache for 1 hour

```
cache.setex(  
    f'product:{product_id}',  
    3600, # 1 hour TTL  
    json.dumps(product)  
)
```

```
return jsonify(product)
```

```
@app.route('/api/products/<product_id>/purchase', methods=['POST'])
```

```
def purchase_product(product_id):
```

```
    """
```

```
    Handle product purchase
```

```
    Demonstrates transaction across multiple systems
```

```
    """
```

```
data = request.json
```

```
user_id = data.get('user_id')
```

```
quantity = data.get('quantity', 1)
```

Step 1: Check stock

```
cursor = db.cursor()
```

```
cursor.execute("""
```

```
    SELECT stock, price FROM products WHERE id = %s
```

```
""", (product_id,))
```

```
stock, price = cursor.fetchone()
```

```
if stock < quantity:
```

```
    return jsonify({"error": "Insufficient stock"}), 400
```

Step 2: Begin transaction

```
try:
```

```
    # Deduct stock
```

```
    cursor.execute("""
```

```
        UPDATE products
```

```
        SET stock = stock - %s
```

```
WHERE id = %s
""" , (quantity, product_id))

# Create order
cursor.execute("""
    INSERT INTO orders (user_id, product_id, quantity, total)
    VALUES (%s, %s, %s, %s)
    RETURNING id
""", (user_id, product_id, quantity, price * quantity))

order_id = cursor.fetchone()[0]

# Commit transaction
db.commit()

# Step 3: Invalidate cache
cache.delete(fproduct:{product_id}')

# Step 4: Send to message queue for further processing
# (email confirmation, inventory update, etc.)
# queue.send('order_created', {'order_id': order_id})

return jsonify({
    "success": True,
    "order_id": order_id,
    "message": "Purchase successful"
})

except Exception as e:
    db.rollback()
    return jsonify({"error": str(e)}), 500
```

javascript

// PRESENTATION TIER - React Frontend

```
import React, { useEffect, useState } from 'react';

function ProductPage({ productId }) {
  const [product, setProduct] = useState(null);
  const [loading, setLoading] = useState(true);
  const [purchasing, setPurchasing] = useState(false);

  // Fetch product on component mount
  useEffect(() => {
    fetchProduct();
  }, [productId]);

  async function fetchProduct() {
    try {
      const response = await fetch(
        `https://api.example.com/api/products/${productId}`
      );
      const data = await response.json();
      setProduct(data);
    } catch (error) {
      console.error('Error fetching product:', error);
    } finally {
      setLoading(false);
    }
  }

  async function handlePurchase() {
    setPurchasing(true);

    try {
      const response = await fetch(
        `https://api.example.com/api/products/${productId}/purchase`,
        {
          method: 'POST',
          headers: {
            'Content-Type': 'application/json',
            'Authorization': `Bearer ${localStorage.getItem('token')}`
          },
          body: JSON.stringify({
            user_id: getCurrentUserId(),
            quantity: 1
          })
        }
      );
    }
  }
}
```

```

const data = await response.json();

if (data.success) {
  alert('Purchase successful!');
  fetchProduct(); // Refresh to show updated stock
} else {
  alert('Purchase failed: ' + data.error);
}
} catch (error) {
  alert('Error: ' + error.message);
} finally {
  setPurchasing(false);
}
}

if (loading) return <div>Loading...</div>;
if (!product) return <div>Product not found</div>;

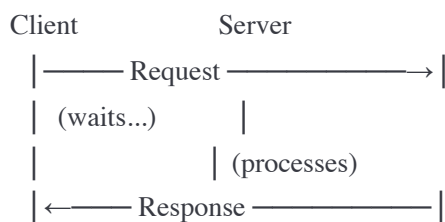
return (
  <div className="product-page">
    <h1>{product.name}</h1>
    <p className="price">${product.price}</p>
    <p className="description">{product.description}</p>
    <p className="stock">In stock: {product.stock}</p>

    <button
      onClick={handlePurchase}
      disabled={purchasing || product.stock === 0}
    >
      {purchasing ? 'Processing...' : 'Buy Now'}
    </button>
  </div>
);
}

```

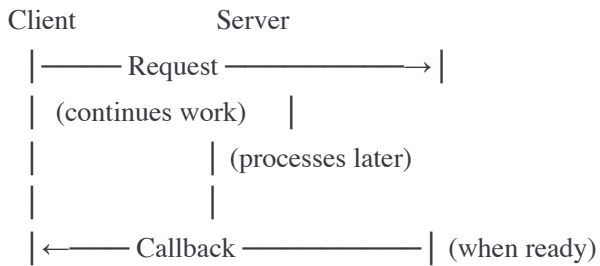
Client-Server Communication Patterns

1. SYNCHRONOUS (Request-Response)



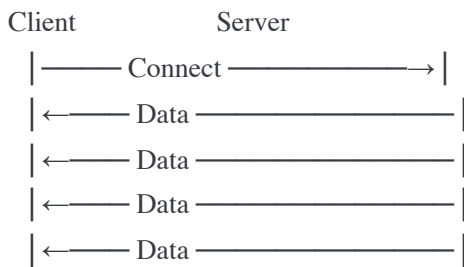
Use: Traditional web pages, API calls

2. ASYNCHRONOUS (Fire and Forget)



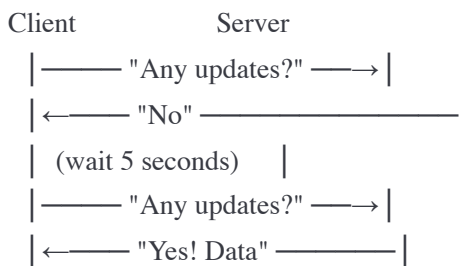
Use: File uploads, long-running tasks

3. STREAMING (Continuous Data)



Use: Video/audio streaming, live updates

4. POLLING (Repeated Requests)



Use: When WebSocket not available

Key Takeaways

1. Hardware Understanding:

- CPU for computation

- RAM for active data (fast but limited)
- Storage for persistence (large but slow)
- Each has different speed/cost trade-offs

2. Performance Metrics:

- **Latency:** Time for one operation (user experience)
- **Throughput:** Operations per second (system capacity)
- Both matter, optimize based on use case

3. Network Fundamentals:

- **Bandwidth:** Capacity of the pipe
- **TCP:** Reliable but slower
- **UDP:** Fast but unreliable
- **HTTP:** Standard web protocol
- **WebSocket:** Real-time bidirectional

4. Client-Server:

- Separation of concerns (presentation, logic, data)
- Scalability through multiple servers
- Different communication patterns for different needs

Practice Questions

1. If a database query takes 50ms, how many queries can you handle per second on one server?
2. Your users are uploading 100 MB files. They have 10 Mbps connections. How long does upload take?
3. Design a system where the client needs to know immediately when data changes on the server. Which protocol would you use and why?

Next Chapter Preview

In Chapter 3, we'll cover **Scalability Fundamentals**:

- Vertical vs horizontal scaling
- Identifying bottlenecks
- Back-of-the-envelope calculations
- Performance metrics

These concepts build directly on what you learned here about computer architecture and networks!

