Chapter 21: Real-Time Systems

Introduction: What is Real-Time?

Real-Time Systems: Applications that require immediate, bidirectional communication with minimal latency.

```
Traditional Web (Request-Response):
Client: "Any updates?"
Server: "No"
[wait 5 seconds]
Client: "Any updates?"
Server: "No"
[wait 5 seconds]
Client: "Any updates?"
Server: "Yes! Here's new message"

Problem: Wasteful polling, high latency

Real-Time Web (Push):
Client: Opens WebSocket connection
Server: Pushes updates immediately when available
        "New message!" (instant)
        "User typing..." (instant)

Benefit: Instant updates, efficient
```

---

1. Real-Time Communication Technologies

WebSockets (Already covered in Chapter 9, deeper dive)

Full-Duplex Communication:

```
HTTP: Half-Duplex (one direction at a time)

┌──────────┐                    ┌──────────┐
│  Client  │──── Request ──────→│  Server  │
│          │                    │          │
│          │←─── Response ──────│          │
└──────────┘                    └──────────┘


WebSocket: Full-Duplex (both directions simultaneously)

┌──────────┐                    ┌──────────┐
│  Client  │←───── Data ────────│  Server  │
│          │                    │          │
│          │────── Data ───────→│          │
└──────────┘                    └──────────┘

Can send/receive at same time!
```

Scaling WebSockets:

```
Challenge: WebSocket connections are stateful

Single Server (Limited):
┌─────────────────┐
│ Web Server     │
│ Max: 10K       │
│ connections    │
└─────────────────┘

Multiple Servers (How to route?):
┌─────────────────┐  ┌─────────────────┐
│  Server 1     │  │  Server 2     │
│ User A conn   │   │ User B conn   │
└─────────────────┘  └─────────────────┘

Problem: User A sends message to User B
Server 1 has User A connection
Server 2 has User B connection
How to deliver message?

Solution: Message Broker (Redis Pub/Sub)
┌─────────────────┐    ┌─────────────────┐
│  Server 1     │     │  Server 2     │
│ User A conn   │◄──────→│ User B conn   │
└────────┬────────┘    └────────┬────────┘
         │  Subscribe to        │
         │  channels           │
         └───────────┬─────────────┘
           ↓
    ┌─────────────────┐
    │ Redis Pub/Sub │
    └─────────────────┘

Server 1: Publishes message to channel
Server 2: Receives from channel, sends to User B
```

Implementation:

javascript

```
```

```
const WebSocket = require('ws');
const redis = require('redis');
const { createAdapter } = require('@socket.io/redis-adapter');
const { Server } = require('socket.io');

class ScalableWebSocketServer {
  constructor(port) {
    this.io = new Server(port);

    // Redis adapter for multi-server coordination
    const pubClient = redis.createClient({ host: 'redis' });
    const subClient = pubClient.duplicate();

    this.io.adapter(createAdapter(pubClient, subClient));

    this.setupHandlers();
  }

  setupHandlers() {
    this.io.on('connection', (socket) => {
      console.log('Client connected:', socket.id);

      // Join room based on user ID
      socket.on('join', (userId) => {
        socket.join(`user:${userId}`);
        socket.userId = userId;

        // Notify others
        socket.broadcast.emit('user_online', userId);
      });

      // Send message
      socket.on('send_message', async (data) => {
        const { recipientId, message } = data;

        // Save to database
        await db.saveMessage({
          senderId: socket.userId,
          recipientId,
          message,
          timestamp: new Date()
        });

        // Emit to recipient (works across all servers!)
        this.io.to(`user:${recipientId}`).emit('new_message', {
          senderId: socket.userId,
          message,
          timestamp: new Date()
        });
      });

      // Typing indicator
      socket.on('typing', (recipientId) => {
        this.io.to(`user:${recipientId}`).emit('user_typing', socket.userId);
      });

      // Disconnect
      socket.on('disconnect', () => {
        socket.broadcast.emit('user_offline', socket.userId);
      });
    });
  }
}

// Start servers on multiple instances
const server1 = new ScalableWebSocketServer(3001);  // Instance 1
const server2 = new ScalableWebSocketServer(3002);  // Instance 2

// User A connects to Server 1
// User B connects to Server 2
// They can still message each other via Redis!
```

Server-Sent Events (SSE)

One-way streaming from server to client.

```
SSE vs WebSocket:

SSE:
Server ──────────────────────→ Client
       (only server sends)

Use cases:
- News feeds
- Stock tickers
- Notifications
- Live scores

Pros:
✓ Simpler than WebSocket
✓ Auto-reconnection
✓ HTTP/2 multiplexing
✓ Works through firewalls

Cons:
✗ One-way only
✗ Text data only
```

Implementation:

```javascript
```

```
// Server
app.get('/events', (req, res) => {
  // Set SSE headers
  res.setHeader('Content-Type', 'text/event-stream');
  res.setHeader('Cache-Control', 'no-cache');
  res.setHeader('Connection', 'keep-alive');

  // Send initial message
  res.write('data: Connected to event stream\n\n');

  // Send events periodically
  const intervalId = setInterval(() => {
    const data = {
      timestamp: new Date().toISOString(),
      value: Math.random()
    };

    res.write(`data: ${JSON.stringify(data)}\n\n`);
  }, 1000);

  // Cleanup on disconnect
  req.on('close', () => {
    clearInterval(intervalId);
    console.log('Client disconnected from SSE');
  });
});

// Client
const eventSource = new EventSource('/events');

eventSource.onmessage = (event) => {
  const data = JSON.parse(event.data);
  console.log('Received:', data);
  updateUI(data);
};

eventSource.onerror = (error) => {
  console.error('SSE error:', error);
  // Browser automatically reconnects
};
```

---

2. Chat Application Architecture

Complete WhatsApp-like System:

```
┌─────────────────────────────────────────────────────────┐
│       Real-Time Chat Architecture           │
├─────────────────────────────────────────────────────────┤
│                          │
│  Clients (Mobile, Web)                │
│  └───────────┬────────────────┬───────────┘      │
│       │       │          │
│   WebSocket connections │           │
│       ↓       ↓           │
│  ┌────────────────┐  ┌────────────────┐      │
│  │WebSocket Srv │  │WebSocket Srv │      │
│  │  1    │ │  2    │     │
│  └───────┬───────┘  └───────┬───────┘      │
│      │       │         │
│      └───────────┬──────────┘        │
│          ↓           │
│      ┌──────────────────┐         │
│      │ Redis Pub/Sub  │         │
│      │ (Message broker)│         │
│      └────────┬────────┘         │
│          │        │
│      ┌────────┴──────────┐        │
│      ↓      ↓        │
│  ┌───────────────┐  ┌───────────────┐      │
│  │ Kafka    │  │ Cassandra │     │
│  │ (Messages) │  │ (History) │     │
│  └───────────────┘  └───────────────┘      │
│   Persistent    Long-term        │
│   queue      storage         │
└─────────────────────────────────────────────────────────┘
```

Message Flow:
1. User A sends message → WebSocket Server 1
2. Server 1 → Redis Pub/Sub (room/user channel)
3. Server 2 (has User B) receives from Redis
4. Server 2 → User B via WebSocket (instant!)
5. Server 1 → Kafka (persistent queue)
6. Background worker → Cassandra (permanent storage)

Complete Implementation:

```javascript
```

```
const io = require('socket.io');
const redis = require('redis');
const { Kafka } = require('kafkajs');

class ChatServer {
  constructor(port) {
    this.io = io(port);

    // Redis for real-time messaging
    this.pubClient = redis.createClient();
    this.subClient = redis.createClient();

    // Kafka for persistence
    this.kafka = new Kafka({ brokers: ['kafka:9092'] });
    this.producer = this.kafka.producer();

    // User presence tracking
    this.onlineUsers = new Map();

    this.setupSocketHandlers();
    this.setupRedis();
  }

  async setupRedis() {
    await this.producer.connect();

    // Subscribe to all message channels
    this.subClient.psubscribe('chat:*', (err) => {
      if (err) console.error('Subscribe error:', err);
    });

    this.subClient.on('pmessage', (pattern, channel, message) => {
      const data = JSON.parse(message);

      // Emit to specific room via Socket.IO
      this.io.to(channel).emit('message', data);
    });
  }

  setupSocketHandlers() {
    this.io.on('connection', (socket) => {
      console.log('User connected:', socket.id);

      // Authenticate
      socket.on('authenticate', async (token) => {
        try {
```

```
      const user = await this.verifyToken(token);
      socket.userId = user.id;
      socket.username = user.username;

      // Track online status
      this.onlineUsers.set(user.id, {
        socketId: socket.id,
        lastSeen: Date.now()
      });

      // Broadcast online status
      socket.broadcast.emit('user_online', {
        userId: user.id,
        username: user.username
      });

      socket.emit('authenticated', { userId: user.id });

    } catch (error) {
      socket.emit('auth_error', { error: 'Invalid token' });
      socket.disconnect();
    }
  });

  // Join chat room
  socket.on('join_room', (roomId) => {
    socket.join(`chat:${roomId}`);
    console.log(`User ${socket.userId} joined room ${roomId}`);

    // Load recent messages
    this.loadRecentMessages(roomId).then(messages => {
      socket.emit('message_history', messages);
    });
  });

  // Send message
  socket.on('send_message', async (data) => {
    const { roomId, message } = data;

    const messageData = {
      id: this.generateId(),
      roomId,
      senderId: socket.userId,
      senderName: socket.username,
      message,
      timestamp: new Date().toISOString()
    };
```

```
      // Publish to Redis (for real-time delivery)
      await this.pubClient.publish(
        `chat:${roomId}`,
        JSON.stringify(messageData)
      );

      // Persist to Kafka (for storage)
      await this.producer.send({
        topic: 'chat-messages',
        messages: [{
          key: roomId,
          value: JSON.stringify(messageData)
        }]
      });

      console.log(`Message sent in room ${roomId}`);
    });

    // Typing indicator
    socket.on('typing', (roomId) => {
      socket.to(`chat:${roomId}`).emit('user_typing', {
        userId: socket.userId,
        username: socket.username
      });
    });

    // Disconnect
    socket.on('disconnect', () => {
      this.onlineUsers.delete(socket.userId);

      socket.broadcast.emit('user_offline', {
        userId: socket.userId
      });
    });
  });
}

async loadRecentMessages(roomId) {
  // Load last 50 messages from Cassandra
  const messages = await db.query(
    'SELECT * FROM messages WHERE room_id = ? ORDER BY timestamp DESC LIMIT 50',
    [roomId]
  );

  return messages.reverse();  // Oldest first
}
```

```
  async verifyToken(token) {
    // Verify JWT token
    const decoded = jwt.verify(token, SECRET_KEY);
    return decoded;
  }

  generateId() {
    return Date.now().toString(36) + Math.random().toString(36).substring(2);
  }
}

// Start chat server
const chatServer = new ChatServer(3000);

// Background worker to persist messages from Kafka to Cassandra
class MessagePersistenceWorker {
  async start() {
    const consumer = kafka.consumer({ groupId: 'message-persistence' });
    await consumer.connect();
    await consumer.subscribe({ topic: 'chat-messages' });

    await consumer.run({
      eachMessage: async ({ message }) => {
        const data = JSON.parse(message.value);

        await db.query(
          'INSERT INTO messages (id, room_id, sender_id, message, timestamp) VALUES (?, ?, ?, ?, ?)',
          [data.id, data.roomId, data.senderId, data.message, data.timestamp]
        );
      }
    });
  }
}

const worker = new MessagePersistenceWorker();
worker.start();
```

Presence System

Track who's online.

```javascript
```

```javascript
class PresenceSystem {
  constructor(redis) {
    this.redis = redis;
    this.heartbeatInterval = 30000;  // 30 seconds
    this.offlineThreshold = 60000;   // 60 seconds
  }

  async setOnline(userId) {
    const key = `presence:${userId}`;
    const now = Date.now();

    // Set with expiry
    await this.redis.setex(key, 60, now.toString());

    // Add to online set
    await this.redis.sadd('online_users', userId);

    console.log(`User ${userId} is online`);
  }

  async setOffline(userId) {
    await this.redis.del(`presence:${userId}`);
    await this.redis.srem('online_users', userId);

    // Store last seen
    await this.redis.set(`last_seen:${userId}`, Date.now());

    console.log(`User ${userId} is offline`);
  }

  async isOnline(userId) {
    const exists = await this.redis.exists(`presence:${userId}`);
    return exists === 1;
  }

  async getOnlineUsers() {
    return await this.redis.smembers('online_users');
  }

  async getLastSeen(userId) {
    const timestamp = await this.redis.get(`last_seen:${userId}`);

    if (!timestamp) {
      return null;
    }
    return new Date(parseInt(timestamp));
  }

  // Heartbeat to keep connection alive
  startHeartbeat(socket, userId) {
    const interval = setInterval(async () => {
      await this.setOnline(userId);
      socket.emit('heartbeat', { timestamp: Date.now() });
    }, this.heartbeatInterval);

    socket.on('disconnect', () => {
      clearInterval(interval);
      this.setOffline(userId);
    });
  }
}

// Usage
const presence = new PresenceSystem(redisClient);

io.on('connection', (socket) => {
  socket.on('authenticate', async (userId) => {
    // Mark user online
    await presence.setOnline(userId);

    // Start heartbeat
    presence.startHeartbeat(socket, userId);

    // Send list of online users
    const onlineUsers = await presence.getOnlineUsers();
    socket.emit('online_users', onlineUsers);
  });
});
```

---

2. Collaborative Editing (Google Docs)

Operational Transformation (OT)

Problem: Concurrent edits conflict.

```
Scenario: Two users edit same document

Initial: "Hello"
Position: 01234

User A:              User B:
Insert "!" at position 5   Delete "o" at position 4

User A's view:         User B's view:
"Hello!"               "Hell"

Problem: How to merge?

Result without OT: "Hell!" or "Hello" (one change lost)
Result with OT: "Hell!" (both changes applied correctly)
```

Operational Transformation:

```
Operations:

Insert(position, char):
  "Hello" + Insert(5, "!") = "Hello!"

Delete(position):
  "Hello" + Delete(4) = "Hell"

Transform operations to handle concurrency:

User A operation: Insert(5, "!")
User B operation: Delete(4)

Transform User A's op against User B's op:
- Delete at position 4 affects positions after 4
- Insert at position 5 must shift to position 4
- Transformed: Insert(4, "!")

Transform User B's op against User A's op:
- Insert at position 5 doesn't affect position 4
- No transformation needed
- Keep: Delete(4)

Apply transformed operations:
"Hello"
  → Delete(4) = "Hell"
  → Insert(4, "!") = "Hell!"

Both users converge to same state! ✓
```

Implementation:

```
javascript
```

```javascript
class Operation {
  constructor(type, position, char = null) {
    this.type = type;  // 'insert' or 'delete'
    this.position = position;
    this.char = char;
  }

  apply(text) {
    if (this.type === 'insert') {
      return text.slice(0, this.position) +
             this.char +
             text.slice(this.position);
    } else if (this.type === 'delete') {
      return text.slice(0, this.position) +
             text.slice(this.position + 1);
    }

    return text;
  }

  // Transform this operation against another operation
  transform(other) {
    if (this.type === 'insert' && other.type === 'insert') {
      if (this.position < other.position) {
        return this;  // No change
      } else if (this.position > other.position) {
        // Shift position
        return new Operation('insert', this.position + 1, this.char);
      } else {
        // Same position, arbitrarily put this after
        return new Operation('insert', this.position + 1, this.char);
      }
    }

    if (this.type === 'insert' && other.type === 'delete') {
      if (this.position <= other.position) {
        return this;  // No change
      } else {
        // Shift left
        return new Operation('insert', this.position - 1, this.char);
      }
    }

    if (this.type === 'delete' && other.type === 'insert') {
      if (this.position < other.position) {
        return this;  // No change
      } else {
        // Shift right
        return new Operation('delete', this.position + 1);
      }
    }

    if (this.type === 'delete' && other.type === 'delete') {
      if (this.position < other.position) {
        return this;  // No change
      } else if (this.position > other.position) {
        // Shift left
        return new Operation('delete', this.position - 1);
      } else {
        // Same position, operation already done
        return null;  // No-op
      }
    }

    return this;
  }
}

class CollaborativeEditor {
  constructor() {
    this.text = '';
    this.version = 0;
    this.history = [];  // All operations
  }

  applyOperation(op, fromVersion) {
    // Transform against operations since fromVersion
    let transformed = op;

    for (let i = fromVersion; i < this.history.length; i++) {
      const historicOp = this.history[i];
      transformed = transformed.transform(historicOp);

      if (transformed === null) {
        return;  // Operation became no-op
      }
    }

    // Apply transformed operation
    this.text = transformed.apply(this.text);
    this.history.push(transformed);
    this.version++;
    console.log(`Applied op at version ${this.version}: "${this.text}"`);

    return transformed;
  }

  getText() {
    return this.text;
  }

  getVersion() {
    return this.version;
  }
}

// Server manages document
const document = new CollaborativeEditor();

io.on('connection', (socket) => {
  // Send current state
  socket.emit('init', {
    text: document.getText(),
    version: document.getVersion()
  });

  // Receive operations from clients
  socket.on('operation', (data) => {
    const { type, position, char, version } = data;

    const op = new Operation(type, position, char);

    // Transform and apply
    const transformed = document.applyOperation(op, version);

    if (transformed) {
      // Broadcast to all other clients
      socket.broadcast.emit('operation', {
        type: transformed.type,
        position: transformed.position,
        char: transformed.char,
        version: document.getVersion()
      });
    }
  });
});

// Client
const socket = io('http://localhost:3000');
```

```
let localText = '';
let localVersion = 0;

socket.on('init', (data) => {
  localText = data.text;
  localVersion = data.version;
  displayText(localText);
});

// User types
function onKeyPress(position, char) {
  // Apply locally
  localText = localText.slice(0, position) + char + localText.slice(position);
  displayText(localText);

  // Send to server
  socket.emit('operation', {
    type: 'insert',
    position,
    char,
    version: localVersion
  });
}

// Receive operations from other users
socket.on('operation', (op) => {
  // Apply operation
  const operation = new Operation(op.type, op.position, op.char);
  localText = operation.apply(localText);
  localVersion = op.version;
  displayText(localText);
});
```

---

3. Live Streaming

Architecture

```
┌─────────────────────────────────────────────────────┐
│     Live Streaming Architecture         │
├─────────────────────────────────────────────────────┤
│                    │
│  Streamer                │
│  └─────┬─────────────────┐          │
│     │ Upload stream │        │
│      ↓           ↓                 │
│  ┌───────────────┐   ┌───────────────┐            │
│  │ Ingest  │    │ Ingest  │ (Redundant)    │
│  │ Server  │    │ Server  │               │
│  └───────┬───────┘   └───────┬───────┘            │
│      └──────────┬──────────┘             │
│             ↓                 │
│      ┌─────────────────────┐             │
│      │ Transcoding   │              │
│      │ (Multiple     │              │
│      │  qualities)   │              │
│      └──────────┬──────────┘             │
│             │                 │
│    ┌──────────┼──────────┬──────────┐      │
│    ↓      ↓      ↓      ↓      │
│  1080p   720p   480p   360p    │
│    │      │      │      │      │
│    └──────────┼──────────┴──────────┘      │
│             ↓                 │
│      ┌─────────────────────┐             │
│      │    CDN     │              │
│      │ (Global edge) │              │
│      └──────────┬──────────┘             │
│             │                 │
│    ┌──────────┼──────────┬──────────┐      │
│   ↓      ↓      ↓     ↓      │
│ Viewer 1  Viewer 2  Viewer 3  Viewer 4  │
│ (1080p)  (720p)   (480p)  (360p)    │
└──────────────────────────────────────────────────────────┘

Protocols:
- Ingest: RTMP (Real-Time Messaging Protocol)
- Delivery: HLS (HTTP Live Streaming) or DASH
- Latency: 2-30 seconds (acceptable for most use cases)
```

Low-Latency Streaming (< 1 second)

```
WebRTC (Web Real-Time Communication):

Peer-to-Peer:
┌──────────┐          ┌──────────┐
│Streamer │◄─────────────────→│ Viewer  │
└──────────┘ Direct connection └──────────┘
        (< 500ms latency)
```

```
For multiple viewers:

┌──────────┐      ┌──────────────┐
│Streamer  │──────→│SFU (Selective│
└──────────┘      │  Forwarding  │
           │    Unit)     │
           └──────┬───────┘
                  │
       ┌──────────┼──────────┐
       ↓          ↓          ↓
    Viewer 1   Viewer 2   Viewer 3

Latency: 200-500ms (real-time!)
Use cases: Video calls, gaming, live auctions
```

---

4. Real-Time Notifications

Push Notification System:

```
┌─────────────────────────────────────────────────────────┐
│   Real-Time Notification System          │
├─────────────────────────────────────────────────────────┤
│                          │
│ Event Sources                    │
│  ├─ New message                  │
│  ├─ New follower                 │
│  ├─ Like on post                 │
│  └─ System alerts                │
│     ↓                        │
│  ┌────────────────────┐                │
│  │ Event Stream   │           │
│  │  (Kafka)     │           │
│  └──────────┬─────────┘                │
│      │               │
│  ┌─────────▼──────────┐                │
│  │ Notification   │           │
│  │  Service     │           │
│  └──────────┬─────────┘                │
│      │               │
│   ┌──────┴───────┴───────────────┐        │
│   ↓     ↓     ↓      │
│  ┌──────────┐ ┌──────────┐ ┌────────────┐    │
│  │WebSocket │ │ FCM  │ │ Email │   │
│  │(Browser)│ │(Mobile)│ │     │   │
│  └──────────┘ └──────────┘ └────────────┘    │
└─────────────────────────────────────────────────────────┘
```

Implementation:

```javascript
```

```
class NotificationService {
  constructor() {
    this.kafka = new Kafka({ brokers: ['kafka:9092'] });
    this.wsConnections = new Map();  // userId -> socket
    this.fcmClient = new FCMClient();  // Firebase Cloud Messaging
  }

  async start() {
    const consumer = this.kafka.consumer({ groupId: 'notifications' });
    await consumer.connect();
    await consumer.subscribe({ topic: 'user-events' });

    await consumer.run({
      eachMessage: async ({ message }) => {
        const event = JSON.parse(message.value);

        await this.processEvent(event);
      }
    });
  }

  async processEvent(event) {
    // Determine notification type and recipients
    let notification;

    switch (event.type) {
      case 'NEW_MESSAGE':
        notification = {
          userId: event.recipientId,
          title: 'New message',
          body: `${event.senderName}: ${event.preview}`,
          data: { messageId: event.messageId, chatId: event.chatId }
        };
        break;

      case 'NEW_FOLLOWER':
        notification = {
          userId: event.followedUserId,
          title: 'New follower',
          body: `${event.followerName} started following you`,
          data: { userId: event.followerId }
        };
        break;

      case 'POST_LIKED':
        notification = {
```

```
        userId: event.postAuthorId,
        title: 'New like',
        body: `${event.likerName} liked your post`,
        data: { postId: event.postId }
      };
      break;
  }

  if (notification) {
    await this.sendNotification(notification);
  }
}

async sendNotification(notification) {
  const { userId, title, body, data } = notification;

  // Check user preferences
  const prefs = await db.getUserNotificationPrefs(userId);

  // 1. WebSocket (if user is online)
  if (this.wsConnections.has(userId)) {
    const socket = this.wsConnections.get(userId);
    socket.emit('notification', { title, body, data });
    console.log(`Sent WebSocket notification to user ${userId}`);
  }

  // 2. Push notification (mobile)
  if (prefs.pushEnabled) {
    const devices = await db.getUserDevices(userId);

    for (const device of devices) {
      await this.fcmClient.send({
        token: device.fcmToken,
        notification: { title, body },
        data
      });
    }

    console.log(`Sent push notification to ${devices.length} devices`);
  }

  // 3. Email (if enabled and not online)
  if (prefs.emailEnabled && !this.wsConnections.has(userId)) {
    await emailService.send({
      to: prefs.email,
      subject: title,
      body: body
      });

      console.log(`Sent email notification to ${prefs.email}`);
    }

    // 4. Store in notification inbox (always)
    await db.saveNotification({
      userId,
      title,
      body,
      data,
      createdAt: new Date(),
      read: false
    });
  }

  registerConnection(userId, socket) {
    this.wsConnections.set(userId, socket);

    socket.on('disconnect', () => {
      this.wsConnections.delete(userId);
    });
  }
}

const notificationService = new NotificationService();
notificationService.start();

// Register WebSocket connections
io.on('connection', (socket) => {
  socket.on('authenticate', (userId) => {
    notificationService.registerConnection(userId, socket);
  });
});
```

Key Takeaways

1. **Real-Time Communication:**
   - WebSocket: Full-duplex, bidirectional
   - SSE: Server to client only
   - Long polling: Fallback for old browsers
2. **Scaling Real-Time:**
   - Redis Pub/Sub for multi-server

- Sticky sessions or message broker
- Presence tracking

3. **Chat Systems:**
   - WebSocket for delivery
   - Kafka for persistence
   - Cassandra for history
   - Redis for presence
4. **Collaborative Editing:**
   - Operational Transformation
   - CRDTs (alternative)
   - Conflict resolution
5. **Live Streaming:**
   - RTMP for ingest
   - HLS for delivery
   - WebRTC for low-latency
   - CDN for scale
6. **Notifications:**
   - Multiple channels (WS, push, email)
   - User preferences
   - Delivery guarantees

Practice Problems

1. Design WhatsApp (1B users, real-time messaging, presence)
2. Design Google Docs (collaborative editing, real-time sync)
3. Design Twitch (live streaming, chat, millions of viewers)

Ready for Chapter 22: Location-Based Services?