Java Technologies

Lecture 7

Asynchronous Communication & Messaging

Fall, 2025

Agenda

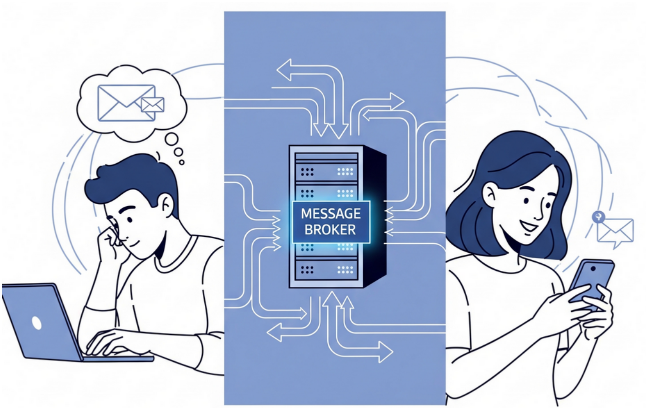
- Patterns of Communication, Asynchronous Communication
- Method-Level Async Processing, `@Async`
- Future vs. CompletableFuture Responses
- Messaging, The Point-to-Point and Pub/Sub Models
- Java Message Service (JMS)
- Acknowledgements & Transactions, Advanced JMS Features
- Event Streaming
- Apache Kafka, Partitions, Replication, Fault Tolerance
- The WebSocket Protocol
- Java EE WebSocket API
- Spring Boot STOMP over WebSocket

Patterns of Communication

- Does the client need an immediate answer?
  - **Synchronous**: Request/Response, Blocking
  - **Asynchronous**: Non-blocking
- How many consumers should process a message?
  - **Point-to-Point**: One-to-One, using a Queue
  - **Publish-Subscribe**: One-to-Many, using a Topic
- How coupled should the services be?
  - **Tight Coupling**: Agents know about each other
  - **Loose Coupling**: Agents communicate using a third-party
- What are the latency and throughput needs?
  - **Fire-and-Forget**: High throughput, can tolerate data loss
  - **Async with Persistence**: High throughput, must not lose data
  - **Real-Time**: Low latency, Bidirectional

Asynchronous Communication

A form of interaction that doesn't require a real-time, simultaneous exchange between agents, allowing for a delayed response between sending and receiving messages.



Method-Level Asynchronous Processing

- The "agents" are methods, one invoking another.
- The "broker" is the application framework/server.
- It allows you to execute tasks
  $\rightarrow$ **without blocking the main thread.**
- What exactly means "the main thread" in our context? ☹
- The **Thread-per-Request** Model
  - The application server maintains a **pool of worker threads**.
  - Each HTTP request is exectued by a thread from the pool.
  - Once the response is sent, the worker returns to the pool.
  - **The request thread is blocked** until the response is sent. This is the default behavior for standard synchronous code.
- A method invoked asynchronously is executed in a **background thread**, allowing the request thread to proceed without waiting for the result $\rightarrow$ improving responsiveness. Always?

When to Use Method-Level Async

- Why not increase the number of request-threads?
  The request-handling threads from the server pool are "more expensive" than simple background threads (increased context switching, memory usage, scheduling overhead).
- Creating separate pool executors for the background threads is flexible, but may also be expensive. 💡
- So, when to use method-level async, and when not?
  - ✔ Fire-and-Forget operations
  - ✔ I/O-bound tasks
  - ✔ Tasks that can be parallelized

  - ✘ CPU-bound tasks
  - ✘ Very short-lived tasks
  - ✘ Tightly coupled calls

Using @Async in Spring Boot

- Enable async support in a @Configuration class.

```
@SpringBootApplication
@EnableAsync
```

- Annotate the invoked method with @Async.

```
@Service
public class AsyncService {

    // Fire-and-Forget
    @Async
    public void sendEmail(String user) {
        System.out.println("Email sent to " + user);
    }

    // Expecting a response in the future
    @Async
    public CompletableFuture<String> generateReport() {
        return CompletableFuture.completedFuture("Done.");
    }
```

Calling an Asyc Method

- The `@Async` annotation on a method means executing it on a separate thread from a thread pool, in a **non-blocking** manner.

```
System.out.println("Sending email...");
asyncService.sendEmail(user); // non-blocking
```

- An async method returns a **"promise"** that a result will be available in the future. This promise is represented by the interface `Future`, or `CompletableFuture` (extending it).

```
System.out.println("Requesting report...");
CompletableFuture<String> result = asyncService.generateReport();
// Doing other work while report is generated...
System.out.println(result.get()); // blocking
```

- Methods are provided to check if the computation is complete, to wait for its completion, to get the result, or to cancel it.

Future vs. CompletableFuture

- Both represent the result of an asynchronous computation.
- `CompletableFuture` is an extension of `Future`, designed for **reactive, non-blocking** programming.

```
public class CompletableFuture<T>
    implements Future<T>, CompletionStage<T>
```

- `CompletionStage` allow you to **chain (combine)** asynchronous tasks, running them conditionally or in parallel.

```
CompletableFuture<String> future1 = service1.fetch();
CompletableFuture<String> future2 = service2.fetch();

var result = CompletableFuture.allOf(future1, future2)
     .thenApply(v -> { // combine results after both are done
            String result1 = future1.join();
            String result2 = future2.join();
            return result1 + " & " + result2;
}); // This is especially useful in microservices architectures
```

Fine Tunning Async Execution

- Creating a custom ThreadPoolTaskExecutor

```
@Configuration
public class AsyncConfig {

  @Bean(name = "myAsyncExecutor")
  public Executor asyncExecutor() {
    var executor = new ThreadPoolTaskExecutor();
    executor.setCorePoolSize(5);   // Min number of threads
    executor.setMaxPoolSize(10);   // Max number of threads
    executor.setQueueCapacity(25); // Before spawning new threads
    executor.initialize();
    return executor;
  }
} // By default: no queueing, unbounded thread creation.
```

- Use it, specifying the bean qualifier.

```
@Async("myAsyncExecutor")
public CompletableFuture<String> doWork() {
    return CompletableFuture.completedFuture("Done.);
}
```

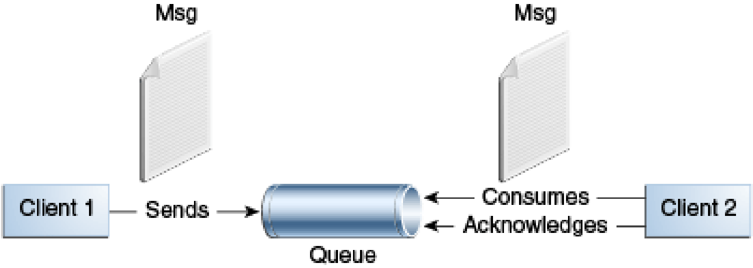
Messaging

- Messaging is a method of **asynchronous** communication between services or applications (agents).
- Instead of calling each other directly, services send messages to a **broker**, which routes and delivers them to consumers.
- It enables **decoupled, reliable, and scalable** communication.
- It is paramount for microservices architectures.

Key Concepts

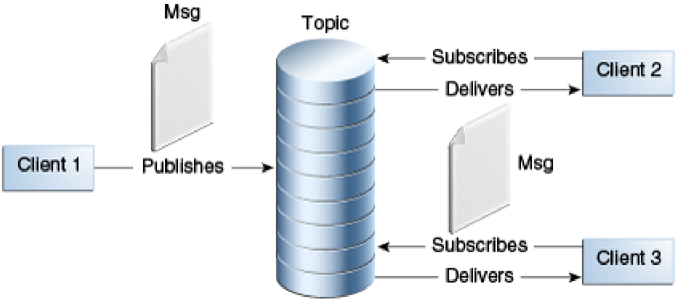
- **Message**: Payload + metadata/headers
- **Producer / Publisher**: Creates and sends messages.
- **Consumer / Subscriber**: Receives and processes messages.
- **Broker**: An intermediary that stores, routes, and delivers messages (e.g., Kafka, RabbitMQ, ActiveMQ, Apache Pulsar).
- **Channels**: The virtual pipes connecting senders and receivers.
  - **Point-to-point** (Queue): One producer, one consumer
  - **Publish-Subscribe** (Topic): One producer, multiple consumers.
- **Delivery Guarantees**
  - At most once (possible loss, no duplicates).
  - At least once (no loss, possible duplicates).
  - Exactly once (ideal, harder to achieve).

The Point-to-Point Model



- Each message is addressed to a specific queue by a **sender**.
- **Receivers** extract messages from the queues.
- **Queues** retain messages until they are consumed or expire.
- Each message has only one consumer.
  $\rightarrow$ Competing Consumer Pattern
- The receiver can fetch the message whether or not it was running when the producer sent the message.

The Publish/Subscribe Model

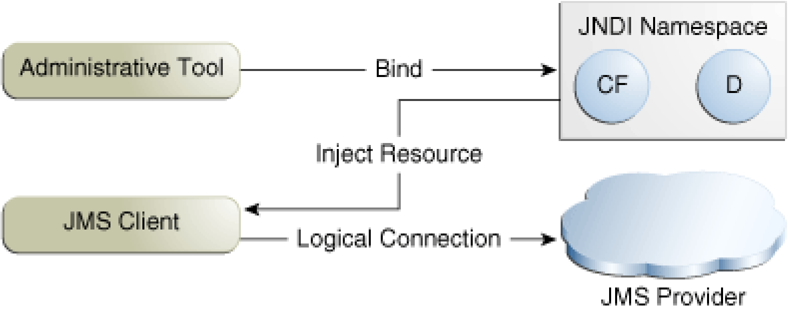


- **Producers** publish messages to a **topic**.
- **Consumers** receive messages by **subscribing** to that topic.
- A topic can have many consumers, but a subscription has only one consumer. Subscriptions may be **durable** or not.
  → The Fan-Out Pattern

Java Message Service (JMS)

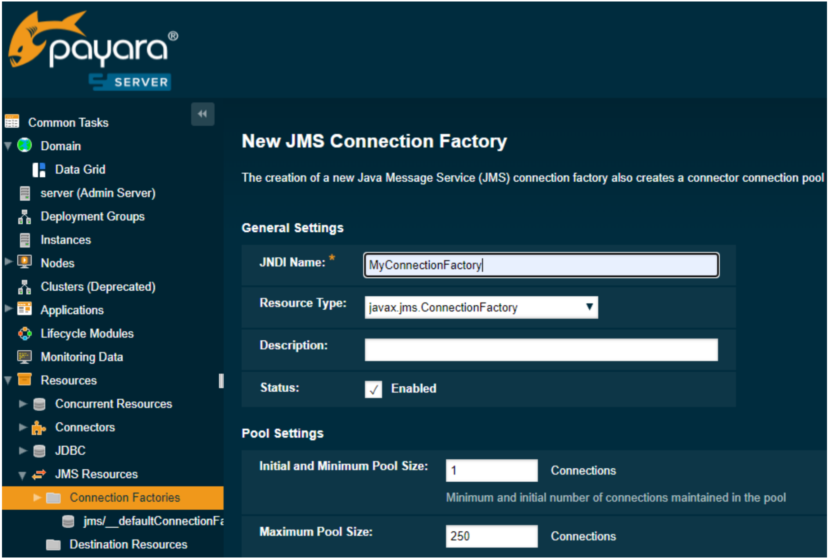
- JMS is a Java/Jakarta EE specification that defines a **standard API** for how Java programs can interact with message-oriented middleware (MOM), **regardless of the protocol**.
- Offers **portability** of JMS applications across JMS providers.
- Implementations: OpenMQ (Oracle), ActiveMQ (Apache), RabbitMQ (Pivotal), JBoss Messaging, etc.
- Application servers, like GlassFish, Payara or WildFly, have a JMS implementation included, by default.
- **Advantages**: Standardized API, reliability, scalable – integrates with EJB, widespread adoption in enterprise applications.
- **Disdvantages**: Not suitable for distributed event streaming, where replication and very high throughput is required.

JMS Architecture



- A **JMS provider** is a messaging system that implements the JMS interfaces and provides administrative and control features.
- **JMS clients** are the programs or components, that produce and consume messages, using the P2P or Pub/Sub models.
- **Administered objects** are JMS objects configured for the use of clients: Connection Factories (CF), Destinations.

Administered Objects Configuration



Sending a Message Using JMS (Java EE)

```
@Stateless // EJB (Enterprise Java Beans)
public class SimpleMessageSender {

  @Resource // JNDI (Java Naming and Directory Interface)
  private ConnectionFactory connectionFactory;

  @Resource(lookup = "jms/MyQueue")
  private Queue queue;

  public void sendMessage(String text) {
    try (JMSContext context = connectionFactory.createContext()) {

        TextMessage message = context.createTextMessage(text);

        JMSProducer producer = context.createProducer();
        producer.send(queue, message);
        System.out.println("Sent message: " + text);
    } catch (JMSException e) {
        e.printStackTrace();
    }
  }
} // EJBs are the Java EE components for business logic.
```

Receiving a Message Using JMS (Java EE)

```
@Stateless
public class SimpleMessageReceiver {

  @Resource
  private ConnectionFactory connectionFactory;

  @Resource(lookup = "jms/MyQueue")
  private Queue queue;

  public void receiveMessage() {
    try (JMSContext context = connectionFactory.createContext()) {
        // Context =  Connection + Session
        JMSConsumer consumer = context.createConsumer(queue);

        //Receive one message synchronously (blocking)
        TextMessage message = (TextMessage) consumer.receive();
        System.out.println("Received message: "
            + message.getText());

     } catch (Exception e) { e.printStackTrace(); }
  }
}
```

Receiving a Message Using a MDB

Message consumption can be implemented **asynchronously / non-blocking**, using EJB Message-Driven Beans.

```
@MessageDriven(mappedName="jms/myQueue") // EJB
public class SimpleMessageReceiver implements MessageListener {

    @Override
    public void onMessage(Message message) {
        try {
            if (message instanceof TextMessage) {
                String text = ((TextMessage) message).getText();
                System.out.println("Received message: " + text);
            }
        } catch (Exception e) {
           System.err.println(e);
        }
    }
}
```

Acknowledgement & Transactions

- **Acknowledgment** is the mechanism by which a consumer signals to the JMS provider (the broker) that a message has been successfully received and processed.
    - `CLIENT_ACKNOWLEDGE`: Manual acknowledgment by client. Overhead: High. Usage: Critical messages.
    - `AUTO_ACKNOWLEDGE`: Automatic acknowledgment after receive. Overhead: Medium. Usage: General Purpose.
    - `DUPS_OK_ACKNOWLEDGE`: Lazy batch acknowledgment. Overhead: Low. Usage: High throughput, Idempotent ops.
- **Transacted Sessions**
    - Groups a series of operations into an atomic unit of work.
    - Messages are not considered processed until the transaction is completed → `session.commit()`.
    - The acknowledgment of messages is part of the transaction.
    - If the transaction fails → `session.rollback()`.
    - Highest overhead.

Advanced JMS Features

- Message **Selection and Filtering**
  $\rightarrow$ SQL-based Message Selectors
- Message **Persistence**
  $\rightarrow$ `DeliveryMode.PERSISTENT, NON-PERSISTENT`
- Message **Priority Levels**
  $\rightarrow$ from 0 (lowest) to 9 (highest), the default level is 4
- Allowing Messages to **Expire**
  $\rightarrow$ Messages can have a limited time-to-live (TTL).
- **Delivery Delay**
  $\rightarrow$ Scheduled messaging. Messages are held by the broker until the specified delivery time is reached. Why?
- Creating **Temporary Destinations**
  $\rightarrow$ Last only for the duration of the connection in which they are created. Request-Reply Pattern.

Using JMS in Spring Boot

- Spring Boot (Tomcat) does not have a JMS provider included.
- The most common broker supporting JMS is ActiveMQ.

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-activemq</artifactId>
</dependency>
```

- Works with multiple providers (Artemis, RabbitMQ, IBM MQ).
- Download and start ActiveMQ broker.
- Configure `application.properties`

```
spring.activemq.broker-url=tcp://localhost:61616
spring.activemq.user=admin
spring.activemq.password=admin
spring.jms.pub-sub-domain=false   # false=Queue, true=Topic
```

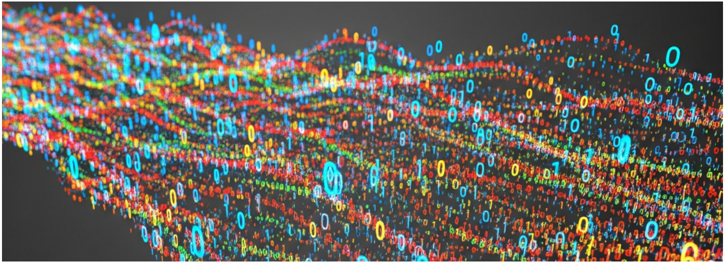
- Auto-configuration removes most of the boilerplate.

Sending/Receiving Messages Using JMS (Spring)

```java
@Service
public class MessageSender {

    @Autowired private JmsTemplate jmsTemplate;

    public void send(String message) {
        jmsTemplate.convertAndSend("MyQueue", message);
        System.out.println("Sent message: " + message);
    }
}
```

```java
@Component
public class MessageReceiver {

    @JmsListener(destination = "MyQueue")
    public void receiveMessage(String message) {
        System.out.println("Received message: " + message);
    }
}
```

Event Streaming



- Event streaming is a data processing paradigm where data is treated as a **continuous, unbounded stream of events**.
- An **event** is any record of something that happened in a system – for example, a user clicking a button, a financial transaction, a sensor reading, or a system log entry.
- **It deals with data as it arrives** → crucial for data platforms, event-driven architectures, and microservices.

Key Concepts of Event Streaming

- **Event**: A record describing something that happened – it has a timestamp, metadata, and payload. Similar to a message.
- **Event Stream**: A continuous, ordered sequence of events, often unbounded. Similar to a never-ending log.
- **Producer**: Systems that generate events and publish them to a stream. Mobile/web apps, IoT devices, backend services.
- **Consumers**: Applications or services that subscribe to event streams to process or react to the data in real time. Fraud detection apps, real-time dashboards, alerting systems.
- **Brokers / Streaming Platform**: Middleware that transports, stores, and distributes events.

  → Apache Kafka, Amazon Kinesis, Apache Pulsar.

Key Features of Event Streaming Systems

- **Event-Driven Architecture**: Respond automatically to events.
- **Real-time**: Act on data instantly.
- **Decoupled**: Producers and consumers are independent.
- **Immutable and Ordered**: Events are typically immutable records that are stored in a log in the order they occurred.
- **Replay and Persistence**: Event logs can replayed later.
- **Scalable**: Handle millions of events per second.
- **Resilient**: If a consumer fails, it can pick up where it left off, thanks to the stored event log.
- Event Streaming vs. Messaging vs. Batch Processing
  - **Event streaming**: Real-time data processing, data pipelines.
  - **Traditional Messaging**: Task distribution, command queues.
  - **Batch Processing**: Nightly jobs, processing data in chunks.

Apache Kafka

kafka

More than 80% of all Fortune 100 companies trust, and use Kafka.

Apache Kafka is an open-source distributed event streaming platform used by thousands of companies for high-performance data pipelines, streaming analytics, data integration, and mission-critical applications.

10 OUT OF 10
MANUFACTURING

7 OUT OF 10
BANKS

10 OUT OF 10
INSURANCE

8 OUT OF 10
TELECOM

Key Concepts of Apache Kafka

- **Event**: A record of type (key, value, timestamp, headers)
- **Topic**: A named stream of events.
- **Producer**: Publishes (writes) events into topics.
- **Consumer**: Subscribes to (reads) events from topics. Can be grouped into **consumer groups** for load balancing.
- **Broker**: A Kafka server that stores topics and serves clients.
- **Partition**: Topics are split into partitions for scalability and parallelism. Each partition is an ordered log of events.
- **ZooKeeper / KRaft**: Manages metadata, cluster coordination, and leader election.
- **Replication**: Kafka replicates partitions across brokers for fault tolerance. One broker acts as the **leader**, others as **followers**.

Apache Kafka Use Cases

- **Messaging & log aggregation**
  Netflix uses Kafka to aggregate logs from thousands of microservices.
- **Real-time analytics & monitoring**
  Monitoring website traffic and alerting if traffic suddenly spikes.
- **Event-driven microservices**
  Central "nervous system" for microservices in an e-commerce app.
- **Data pipelines & integration**
  Streaming data from a website to a data warehouse for analytics.
- **IoT & sensor data streaming**
  Smart factories → telemetry → anomaly detection systems.
- **Fraud detection & security**
- **Machine learning pipelines**
- **Stream processing apps**

Using Kafka in Spring Boot

- Add the Spring Kafka starter.

```
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter</artifactId>
    </dependency>
```

- Configure application.yml

```
spring:
  kafka:
    bootstrap-servers: localhost:9092
    consumer:
      group-id: my-group
      auto-offset-reset: earliest
      key-deserializer: StringDeserializer
      value-deserializer: StringDeserializer
    producer:
      key-serializer: StringSerializer
      value-serializer: StringSerializer
# org.apache.kafka.common.serialization.StringSerializer
```

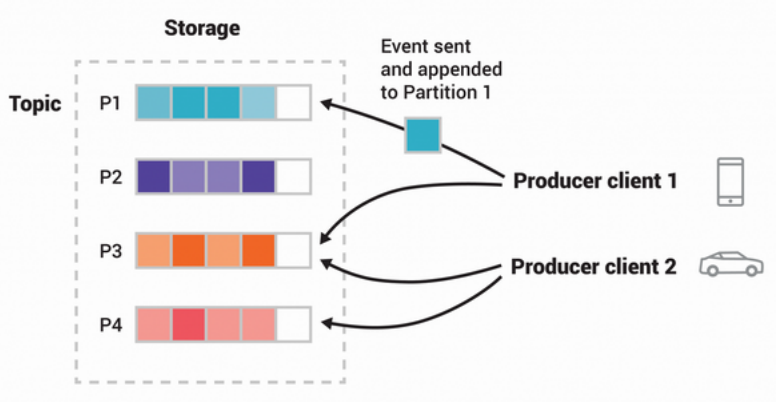
A Kafka Simple Producer & Consumer

```
@Service
public class MessageProducer {
    @Autowired
    private KafkaTemplate<String, String> kafkaTemplate;

    public void sendMessage(String message) {
        kafkaTemplate.send("my-topic", message); // What about the key?
        System.out.println("Sent: " + message);
    }
}
```

```
@Service
public class MessageConsumer {

    @KafkaListener(topics = "my-topic", groupId = "my-group")
    public void listen(String message) {
        System.out.println("Received: " + message);
    }
}
```

Kafka Partitions

- A partition is a **sub-division of a topic**.
- Each partition is an ordered, immutable sequence of records that is append-only (new records are always added to the end).
- Records in a partition are identified by an **offset** (index).



Kafka Records & Offsets

- A **record (message, event)** is the basic unit of data written to and read from Kafka topic. It consists of:
  - **Key** (optional): Used for partitioning.
  - **Value**: The actual data (payload - JSON, string, Avro, etc.).
  - **Timestamp**: When the record was produced.
  - **Headers** (optional): Tracing info or content type.
- Each record in a partition has a unique, sequential **offset**.
- A **consumer offset** is the number that marks the last record the consumer has successfully processed in a partition.
- Consumers **commit their offsets**, meaning they are stored them in a special internal topic, so it can resume from there later.
  - **Automatic** (Default): Periodically in the background.
  - **Manual** (Acknowledgment): After processing is successful.

Using Acknowledgments

```
spring.kafka.consumer.enable-auto-commit=false
```

```java
@Service
public class MyKafkaConsumer {

  @KafkaListener(topics = "my-topic", groupId = "my-group")
  public void consume(ConsumerRecord<String, String> record,
                      Acknowledgment ack) {
    try {
      System.out.println("Processing message: " + record.value());
      // Do some business logic...
      // Commit offset manually AFTER successful processing
      ack.acknowledge();
      // Kafka won't advance the offset until acknowledge is called.
    } catch (Exception e) {
      System.err.println("Processing failed: " + record.value());
      // no ack → offset not committed → message will be retried
    }
  }
}
```

Partition Assignment with Keys

- Every record can optionally include a **key of any type**.
- The key **determines the partition** the record goes into.

```
partition = hash(key) % number_of_partitions
hash(key) → typically uses Murmur2 (fast, evenly distributed).
```

  → All records with the same key go to the same partition.
  → Ensures the correct ordering of events.

- **Skewed data** can lead to "hot partitions". ⚠
- If no key is given, Kafka uses a **round-robin strategy** to spread records evenly across partitions (usually, stateless events).
  → Even load distribution across partitions.
  → Maximizes throughput, no ordering guarantee.
- **Compacted Topics**: Only the latest value for a key is stored (older records with the same key are deleted).

Replication & Fault Tolerance

- Each partition is **replicated across multiple brokers**.
  - One copy is the **leader**, it handles all reads and writes.
  - The others are **followers**, copy the leader's data to stay in sync.
- **Replication factor** = how many copies exist for each partition.
- **Fault Tolerance**: Data remains durable and accessible even if individual brokers (servers) in the cluster fail.
  - **Leader Failure**: One of the in-sync followers (ISRs) is elected as the new leader. How is the new leader elected? 💡
  - **Follower Failure**: Kafka continues with the leader + remaining followers. When the failed broker recovers, it catches up.
  - **Cluster Failure**: At least one replica/partition must be alive.
- Acknowledgement settings for ISRs (configurable):

```
spring.kafka.producer.
  acks=0   # fire and forget (no fault tolerance).
  acks=1   # leader confirms write, followers may lag
  acks=all # leader waits for all ISRs to confirm (minimal risk)
```

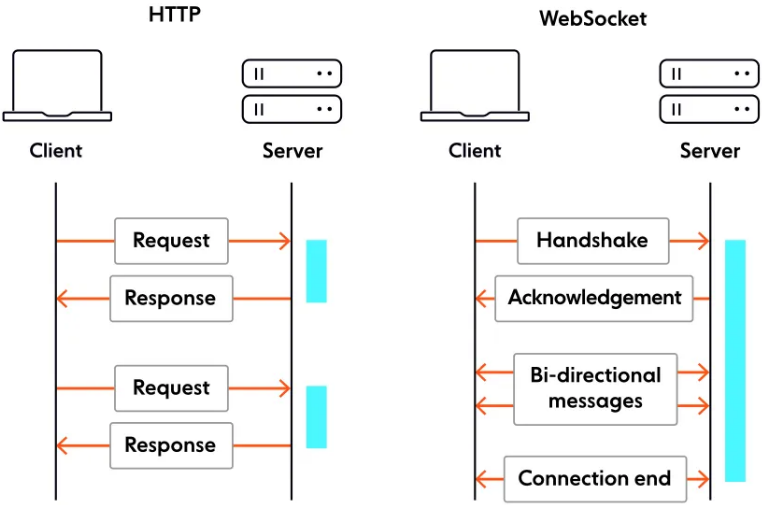
Key Mechanisms for Scalability

- **Partitioning:**
  - Topics are split into multiple partitions.
  - Enables parallel reads and writes.
- **Distributed Brokers:**
  - Partitions spread across multiple brokers.
  - Adding brokers balances load and increases throughput.
- **Replication:**
  - Partitions replicated across brokers for fault tolerance.
  - Optional read scaling from followers.
- **Producer Scalability:**
  - Multiple producers write to same topic/partition.
  - Append-only logs for fast sequential disk writes.
- **Consumer Scalability:**
  - Consumer groups read partitions in parallel.
  - More partitions $\rightarrow$ more consumers can be added.

The WebSocket Protocol

- WebSocket is a **bidirectional, full-duplex** communication protocol, over a single TCP connection.
- The connection is **persistent**, it remains open until it is explicitly closed by either the client or the server.
- It has a **lower overhead** than HTTP with respect to the data frames exchanged between a client and a web server.
- Suitable for **real-time** communication.
- It has dedicated **URL schemes**: `ws://`, `wss://`
- **Use Cases**: Chats, live notifications, dashboards, online games.
- **Implementations**:
    - Java/Jakarta EE WebSocket API
    - Spring Boot WebSocket with STOMP (over WebSocket API)
      STOMP = Simple (or Streaming) Text Oriented Messaging Protocol.

HTTP vs. WebSocket



Using Java EE WebSocket API

```java
@ServerEndpoint("/chat")
public class ChatEndpoint {

    @OnOpen
    public void onOpen(Session session) {
        System.out.println("Connected: " + session.getId());
    }

    @OnMessage
    public void onMessage(String message, Session session) {
        // message is a raw data frame (String, byte[], etc.)
        System.out.println("Message received: " + message);
        // You need to store the session in a collection
        // if additional operations are required (broadcast)
    }

    @OnClose
    public void onClose(Session session) {
        System.out.println("Closed: " + session.getId());
    }
}
```

STOMP over WebSocket

- It provides a **messaging** protocol layer on top of WebSocket.
  $\rightarrow$ Higher-level semantics and infrastrucure support.
- **Topic-based Pub/Sub**: Broadcast a message to multiple clients subscribed to a topic.
- **Easy integration with Spring Messaging**: `@MessageMapping` similar to `@RequestMapping` for HTTP.
- **Supports Fallback with SockJS**: Fallback to HTTP streaming or polling if the browser does not support WebSockets
- **Broker support for scaling**: STOMP can work with external message brokers like Kafka or RabbitMQ.
- **Metadata and Headers**: STOMP messages carry headers for authorization, session IDs, content type.

Using STOMP over WebSocket in Spring Boot

- Add the starter `spring-boot-starter-websocket`.
- Configure WebSocket with STOMP

```
@Configuration
@EnableWebSocketMessageBroker
public class WebSocketConfig
        implements WebSocketMessageBrokerConfigurer {
  @Override
  public void registerStompEndpoints( // for clients, to connect
          StompEndpointRegistry registry) {
      registry.addEndpoint("/ws")
          .setAllowedOriginPatterns("*").withSockJS();
  }
  @Override
  public void configureMessageBroker(
          MessageBrokerRegistry registry) {
      // Prefix for messages sent from server to clients
      registry.enableSimpleBroker("/topic");
      // Prefix for messages sent from clients to server
      registry.setApplicationDestinationPrefixes("/app");
  }
}
```

Create a Message Model and a Controller

- Define a simple message payload class.

```
public record ChatMessage(String sender, String content) {}
```

- Create a controller to handle messages.

```
@Controller
public class ChatController {

    @MessageMapping("/chat")   // listens to /app/chat
    @SendTo("/topic/messages") // broadcasts to /topic/messages
    public ChatMessage sendMessage(ChatMessage message)
            throws Exception {
        System.out.println("Received: " + message);
        return message;
    }
}
```

- Create an HTML/JS Client.