Session - 12

**📘 Notes: Synchronous vs. Asynchronous Communication in Computing**

**🔁 1. What is Synchronous Communication?**

**📌 Definition:**

**Synchronous communication** is when two systems interact **in real-time** — the sender sends a request and **waits for a response** before continuing.

In synchronous systems, the sender is **blocked** (paused) until the receiver processes the request and replies.

**💡 Example in Daily Life:**

* A phone call: You talk and wait for the other person to reply before continuing.
* ATM transaction: You enter your PIN and wait for the bank to verify before proceeding.

**💻 Programming Example:**

String result = httpClient.get("https://api.example.com/data");

System.out.println("Response: " + result);

* The program pauses on the get() line until the data is received.

**✅ Advantages:**

* Simple to implement and understand
* Immediate feedback — you get the result instantly
* Easier debugging (since execution is linear)

**❌ Disadvantages:**

* Slower performance (waiting time)
* Not resilient: if the receiver is slow or down, the sender is stuck
* Poor scalability under heavy load

**🔄 2. What is Asynchronous Communication?**

**📌 Definition:**

**Asynchronous communication** is when the sender sends a request and **continues without waiting** for the response. The response (if needed) is handled **later**, often via callbacks, events, or polling.

The sender and receiver **work independently** of each other.

**💡 Example in Daily Life:**

* Sending an email: You send it and continue your work. The recipient replies later.
* Ordering food: You place the order and wait; meanwhile, you do other things.

**💻 Programming Example:**

CompletableFuture<String> future = httpClient.getAsync("https://api.example.com/data");

future.thenAccept(response -> {

System.out.println("Received: " + response);

});

System.out.println("Request sent, not waiting for response.");

* The program doesn't wait; it prints and does other tasks while the request is processed in the background.

**✅ Advantages:**

* Non-blocking: Better performance and responsiveness
* Scalable: Can handle many tasks simultaneously
* Useful for slow or background tasks (e.g., email, report generation)

**❌ Disadvantages:**

* More complex to write and manage
* Error handling can be tricky
* Harder to debug due to non-linear flow

**🔬 3. Key Differences**

| **Feature** | **Synchronous** | **Asynchronous** |
| --- | --- | --- |
| **Waits for response** | Yes | No |
| **Blocking** | Yes – caller waits | No – caller continues |
| **Complexity** | Simpler | More complex |
| **Performance** | Slower under high load | Faster and more scalable |
| **Use Cases** | Login, payment, data retrieval | Email, notifications, background jobs |
| **Real-world analog** | Phone call | Email |

**🧰 4. When to Use What?**

**✅ Use Synchronous:**

* When the result is needed **immediately** to proceed
* For **critical or real-time** operations (e.g., authentication, form submission)

**✅ Use Asynchronous:**

* For **long-running or background tasks** (e.g., video processing, report generation)
* When you want better **system responsiveness** and scalability
* In **event-driven** or **distributed** systems (like microservices)

**📦 5. In the Context of Messaging Systems (e.g., RabbitMQ)**

* **Synchronous**:
  + Client sends a message and waits for acknowledgment or result
  + Used in tightly coupled systems
* **Asynchronous**:
  + Producer sends a message to a queue and moves on
  + Consumer later processes the message independently
  + Used in **event-driven architecture**

**🧠 Summary**

* **Synchronous = Wait**  
  Direct and immediate communication. Slower, simpler.
* **Asynchronous = Don’t Wait**  
  Send and forget. Faster, more scalable, but more complex.

**📘 Types of Asynchronous Communication**

Asynchronous communication can be broadly classified into:

1. **Fire-and-Forget**
2. **Callback-Based (or Response-Based)**

**🔥 1. Fire-and-Forget**

**📌 Definition:**

In **fire-and-forget**, the sender **sends a message** and **does not care** whether it was received or processed. No response is expected.

Think of it like dropping a letter in a mailbox and walking away — you don’t wait for confirmation.

**💡 Real-world Example:**

* Sending an email
* Logging an event (e.g., user clicked a button)

**💻 Programming Example (Java + RabbitMQ):**

rabbitTemplate.convertAndSend("email\_queue", "Send welcome email");

System.out.println("Email message sent.");

* The sender just sends the message and moves on — no confirmation or reply expected.

**✅ Use Cases:**

* Logging
* Sending metrics or analytics data
* Notifications (where delivery is not critical)

**🔁 2. Callback-Based (or Response-Based)**

**📌 Definition:**

In this type, the sender sends a request **asynchronously**, but it **expects a response**, which is handled via a **callback function**, **event**, or **future/promise**.

Like ordering food for delivery — you do other things, but you expect it to arrive.

**💡 Real-world Example:**

* Requesting data from an API but continuing with other tasks until the response arrives.

**💻 Programming Example (Java using CompletableFuture):**

CompletableFuture<String> future = apiService.getDataAsync();

future.thenAccept(response -> {

System.out.println("Received response: " + response);

});

System.out.println("Request sent, doing other work...");

* The response will be handled later, asynchronously.

**✅ Use Cases:**

* APIs that return data
* Microservices communicating with one another
* UI applications waiting for data (e.g., autocomplete suggestions)

**🧠 Summary Table**

| **Feature** | **Fire-and-Forget** | **Callback-Based** |
| --- | --- | --- |
| **Response expected?** | ❌ No | ✅ Yes |
| **Complexity** | Simple | More complex |
| **Reliability needs** | Low | Medium to high |
| **Example** | Logging, push notifications | Async APIs, message acknowledgments |
| **Programming concept** | Just send | Promise / Future / Event Listener |

**🧰 Conclusion**

* Both types are **asynchronous** (non-blocking).
* Use **fire-and-forget** when you don’t need confirmation.
* Use **callback-based** when you expect a result or need to act on the response.

In **asynchronous messaging systems**, two important messaging models are:

1. **Point-to-Point (Queue-based)**
2. **Publish/Subscribe (Topic-based)**

These models define **how messages are delivered** from **producers (senders)** to **consumers (receivers)**.

Let’s go through each in detail:

**🎯 1. Point-to-Point (P2P) Model**

**📌 Definition:**

In the **point-to-point** model, messages are sent to a **queue**, and **only one consumer** receives each message.

Think of it like a **task queue** — once a worker picks up a task, no one else can see it.

**💡 Characteristics:**

* Only **one consumer** processes each message.
* Messages are stored in a **queue**.
* Commonly used for **load distribution** (e.g., background job processing).

**💻 Example (with RabbitMQ):**

1. Producer sends a message to a **queue**.
2. One of the available consumers picks it up.

Producer --> Queue --> One Consumer

**🧠 Use Cases:**

* Order processing systems
* Email sending services
* Background job queues (like worker threads)

**📢 2. Publish/Subscribe (Pub/Sub) Model**

**📌 Definition:**

In the **publish/subscribe** model, messages are sent to a **topic (or exchange)**, and **all subscribed consumers** receive a copy of the message.

Think of it like **broadcasting a message** to multiple radio listeners.

**💡 Characteristics:**

* One message is delivered to **multiple subscribers**.
* Subscribers register to a **topic** or **pattern**.
* Used for **event broadcasting**.

**💻 Example (with RabbitMQ - fanout or topic exchange):**

1. Producer sends a message to an **exchange**.
2. The exchange broadcasts it to all **bound queues**.
3. Each queue has a **separate consumer**.

--> Queue1 --> Consumer1

Producer --|

--> Queue2 --> Consumer2

**🧠 Use Cases:**

* Real-time notifications (e.g., news feed updates)
* Logging and monitoring systems
* Chat systems, stock price updates

**🧠 Key Differences**

| **Feature** | **Point-to-Point** | **Publish/Subscribe** |
| --- | --- | --- |
| Delivery | One consumer per message | All subscribers get a copy |
| Target | Queue | Topic / Exchange |
| Consumer count | One | Many |
| Use Case | Task distribution | Event broadcasting |
| RabbitMQ Exchange Type | direct (or default queue) | fanout, topic, or headers |

**📌 Visual Summary**

**Point-to-Point:**

Producer --> [Queue] --> Consumer

**Publish/Subscribe:**

Producer --> [Topic/Exchange]

↙ ↘

Consumer1 Consumer2

**🧰 Summary**

| **Model** | **Suitable For** | **Message Goes To** | **Received By** |
| --- | --- | --- | --- |
| Point-to-Point | One-to-one tasks, job processing | Queue | One consumer |
| Pub/Sub | Broadcasting, notifications | Topic/Exchange | All subscribed consumers |

**🐰 What is RabbitMQ?**

**RabbitMQ** is a **message broker** — it helps applications (or services) communicate with each other by sending messages via a **queue**.

In simple terms:

It’s like a post office. It receives messages from one system, queues them, and delivers them to another system safely, asynchronously, and reliably.

**📦 Why Use RabbitMQ?**

* **Decouples systems**: The sender and receiver don't have to be active at the same time.
* **Asynchronous processing**: Offload heavy or slow tasks to be handled in the background.
* **Reliable delivery**: Messages are not lost even if the receiver is temporarily down.
* **Scalability**: Handle high message loads by distributing across consumers.

**🔧 Key Concepts in RabbitMQ**

**1. Producer**

* The sender of the message.
* It publishes messages to an **exchange**.

**2. Consumer**

* The receiver of the message.
* It listens to a **queue** and processes messages.

**3. Queue**

* A buffer that stores messages until they are consumed.
* Messages are delivered in FIFO (First In, First Out) order.

**4. Exchange**

* Determines how messages are routed to queues.
* Receives messages from the producer and routes them to queues based on rules.

**5. Binding**

* A link between a queue and an exchange.
* It defines the rules for message routing.

**6. Routing Key**

* A string used by exchanges to decide how to route messages to queues.

**🔁 RabbitMQ Message Flow (Simple)**

1. **Producer** sends a message to an **Exchange**.
2. The **Exchange** uses **bindings** and a **routing key** to determine which **queue(s)** the message should go to.
3. **Queue** holds the message.
4. **Consumer** picks up the message from the **Queue** and processes it.

**🔄 Types of Exchanges**

| **Type** | **Description** |
| --- | --- |
| **Direct** | Routes messages to queues based on exact **routing key match**. |
| **Topic** | Routes based on **pattern matching** in routing key (logs.error, user.\*). |
| **Fanout** | Broadcasts messages to **all queues** bound to the exchange. |
| **Headers** | Routes based on **headers** instead of routing key (less common). |

**📘 Real-World Analogy**

Imagine you're ordering food from a restaurant:

* You (Producer) place an order (Message) at the counter (Exchange).
* The counter staff routes the order to the correct kitchen section (Queue) based on the type of food.
* The chef (Consumer) picks up the order from the queue and prepares it.

You don’t care *how* it gets there — you just know it’ll be made and delivered when it’s ready.

**✅ Benefits of Using RabbitMQ**

* **Reliability**: Message acknowledgments, persistence.
* **Durability**: Survives server crashes if configured properly.
* **Clustering**: Can be scaled horizontally across machines.
* **Monitoring**: Management UI and CLI tools.

**⚙️ Basic Code Example (Using Java & Spring Boot)**

@Component

public class Producer {

@Autowired

private RabbitTemplate rabbitTemplate;

public void sendMessage(String message) {

rabbitTemplate.convertAndSend("my\_exchange", "my\_routing\_key", message);

}

}

@Component

public class Consumer {

@RabbitListener(queues = "my\_queue")

public void receiveMessage(String message) {

System.out.println("Received: " + message);

}

}

**📊 Visualization**

Producer --> Exchange --> Queue --> Consumer

↘ Routing Key / Binding ↙

**💡 Summary**

* RabbitMQ acts as a **middleman** for messages between services.
* It allows **asynchronous, reliable communication**.
* Core components: **Producer, Exchange, Queue, Consumer**.
* Supports various exchange types for flexible routing.

Sure, Koushik! Let's go step by step and explain clearly what's happening in the **RabbitMQ setup** for your RoboShop application:

**🐇 What is RabbitMQ?**

* RabbitMQ is a **message broker** or **queue system**.
* It allows different services (like payment, shipping, order) to **communicate asynchronously** via messages.
* Example: Payment service sends an event “Order Paid”, Shipping picks it up and processes delivery.

**🔧 Step-by-Step Explanation**

**✅ Step 1: Create RabbitMQ Repo File**

vim /etc/yum.repos.d/rabbitmq.repo

* vim opens the file where we’ll paste repository details.
* This repo provides access to **RabbitMQ and Erlang packages**.
* Erlang is the language on which RabbitMQ is built, so it's required.

**✨ Inside the File:**

[modern-erlang]

name=modern-erlang-el9

baseurl=... (URLs)

enabled=1

gpgcheck=0

* These sections tell the system where to download RabbitMQ and Erlang packages from.
* $basearch → Dynamically chooses your system's architecture (like x86\_64).

**✅ Step 2: Install RabbitMQ**

dnf install rabbitmq-server -y

* Uses DNF package manager to install RabbitMQ.
* -y auto-confirms the installation.
* This installs:
  + RabbitMQ core service
  + Dependencies like Erlang

**✅ Step 3: Start and Enable RabbitMQ Service**

systemctl enable rabbitmq-server

systemctl start rabbitmq-server

* enable → Starts RabbitMQ automatically on every reboot.
* start → Immediately starts the RabbitMQ service.

You can verify with:

systemctl status rabbitmq-server

**✅ Step 4: Create Application User in RabbitMQ**

By default, RabbitMQ has a guest/guest user, but **this user can only connect from localhost**.

So, we create a custom user:

rabbitmqctl add\_user roboshop roboshop123

* Adds a new user called roboshop with password roboshop123.

rabbitmqctl set\_permissions -p / roboshop ".\*" ".\*" ".\*"

* Grants **full permissions** (read/write/configure) to the user roboshop on the default / virtual host.
  + "."\* means: access all resources.

**📦 Summary Table:**

| **Step** | **Purpose** |
| --- | --- |
| Create repo file | To download latest RabbitMQ & Erlang packages |
| Install RabbitMQ | Installs the message broker service |
| Start & enable service | Ensures it runs now and after reboot |
| Create user | Allows external services (like Payment) to connect |
| Set permissions | Grants access to queues/exchanges in RabbitMQ |

**🧠 Real Example: Where RabbitMQ is used in RoboShop?**

Let’s say:

* A **Payment** service completes a transaction.
* It **sends a message** to RabbitMQ: "Payment for Order 123 completed."
* The **Shipping** service is listening to RabbitMQ.
* It picks the message → Starts packaging and dispatch.

So, RabbitMQ helps these two services **communicate without calling each other directly** (loose coupling).