**Publisher-Subscriber pattern**

To enable an application to announce events to multiple interested consumers asynchronously without coupling the senders to the receivers, we can use a messaging system or a publish-subscribe pattern.

1. **\*\*Messaging System:\*\*** Think of a messaging system as a central hub where events are sent by the application (the sender) and then delivered to interested consumers (the receivers). The key here is that the sender doesn't need to know anything about the receivers. It just sends the event to the messaging system, and the system takes care of delivering it to the right consumers. It's like dropping a letter into a mailbox without worrying about who will pick it up.

2. **\*\*Publish-Subscribe Pattern:\*\*** In this pattern, the application acts as a publisher of events, and multiple interested consumers subscribe to receive those events. Again, the publisher doesn't need to know who the subscribers are; it just publishes the events to a central place. Any consumer interested in receiving specific types of events can subscribe to those topics. When the publisher sends an event, all the subscribers interested in that type of event will automatically receive it, without the need for direct connections.

**Working:**

1. **\*\*Publishing an Event:\*\*** When an event occurs, the application publishes it to the messaging system or the central hub. The event contains information about what happened or what data is available for consumption.

2. **\*\*Event Topics:\*\*** Each event belongs to a specific topic or category. For example, events related to "new product releases" could belong to the "product" topic, while events about "user registrations" could belong to the "user" topic.

3. **\*\*Subscribing to Topics:\*\*** Consumers interested in certain types of events can subscribe to specific topics. For instance, if a consumer is interested in new product releases, they would subscribe to the "product" topic.

4. **\*\*Asynchronous Delivery:\*\*** The messaging system takes care of delivering the events to all the subscribed consumers. The process is asynchronous, meaning the events are sent and received independently of each other, allowing the consumers to handle the events at their own pace.

By using a messaging system or a publish-subscribe pattern, the sender and receivers are decoupled. The sender only needs to publish the events without being aware of who will receive them, and the receivers can subscribe to the events they care about without directly interacting with the sender. This decoupling allows for better scalability, flexibility, and maintainability of the application, as changes in one part of the system won't affect the other parts. It's like sending messages to a group without worrying about who will read them, and each person can choose to read the messages that interest them.

**Context and problem:**

To announce events to all interested consumers without knowing their identities, we can use a concept called "topic-based messaging" in combination with a message broker.

**\*\*1. Message Broker:\*\***

Imagine a message broker as a central hub or middleman that handles all the messages in the system. It's like a smart post office that receives messages from senders and then delivers them to the right recipients. This message broker is responsible for managing and distributing the events efficiently to all interested consumers.

**\*\*2. Topic-Based Messaging:\*\***

In topic-based messaging, events are categorized into different "topics" based on their type or content. It's like sorting messages into different labeled boxes based on their subject. For example, all events related to "product updates" would go into the "product" box, and events about "user registrations" would go into the "user" box.

**\*\*3. Subscribing to Topics:\*\***

Each consumer can choose which topics they are interested in. They "subscribe" to one or more topics they want to receive events for. It's like picking the boxes they want to receive letters from.

**\*\*4. Event Publishing:\*\***

When an event happens, the sender (a component in the application) publishes the event to the message broker along with the topic it belongs to. The sender doesn't need to know anything about the consumers; it simply sends the event with its corresponding topic to the message broker.

**\*\*5. Event Delivery:\*\***

The message broker, being the smart post office, takes care of delivering the events to all the consumers who have subscribed to the corresponding topics. It knows which consumers are interested in which topics based on their subscriptions. The consumers receive the events asynchronously, meaning they can process the events at their own pace, without blocking the sender.

**\*\*6. Scaling and Subset Interests:\*\***

Using this approach, the sender doesn't need to create dedicated message queues for each consumer, which could become inefficient and difficult to manage at scale. Instead, the message broker handles the distribution efficiently. Moreover, consumers can choose to subscribe only to the topics that interest them, receiving only the events they care about.

**Summary:** Topic-based messaging with a message broker allows the sender to publish events to the central message broker without knowing anything about the consumers. Consumers subscribe to topics of their interest, and the message broker ensures that events are delivered to all relevant subscribers. This decouples the senders from the consumers, allowing for scalability and flexibility in cloud-based and distributed applications, while still ensuring that information reaches all interested parties. It's like sending messages to different labeled boxes at the post office, and people can choose which boxes to check for the letters they want to read.

**Solution:**

**The asynchronous messaging subsystem**

**1. \*\*Sender (Publisher):\*\*** Packages events into messages using a known format and sends them via an input messaging channel.

**2. \*\*Consumers (Subscribers):\*\*** Each consumer has its own output messaging channel.

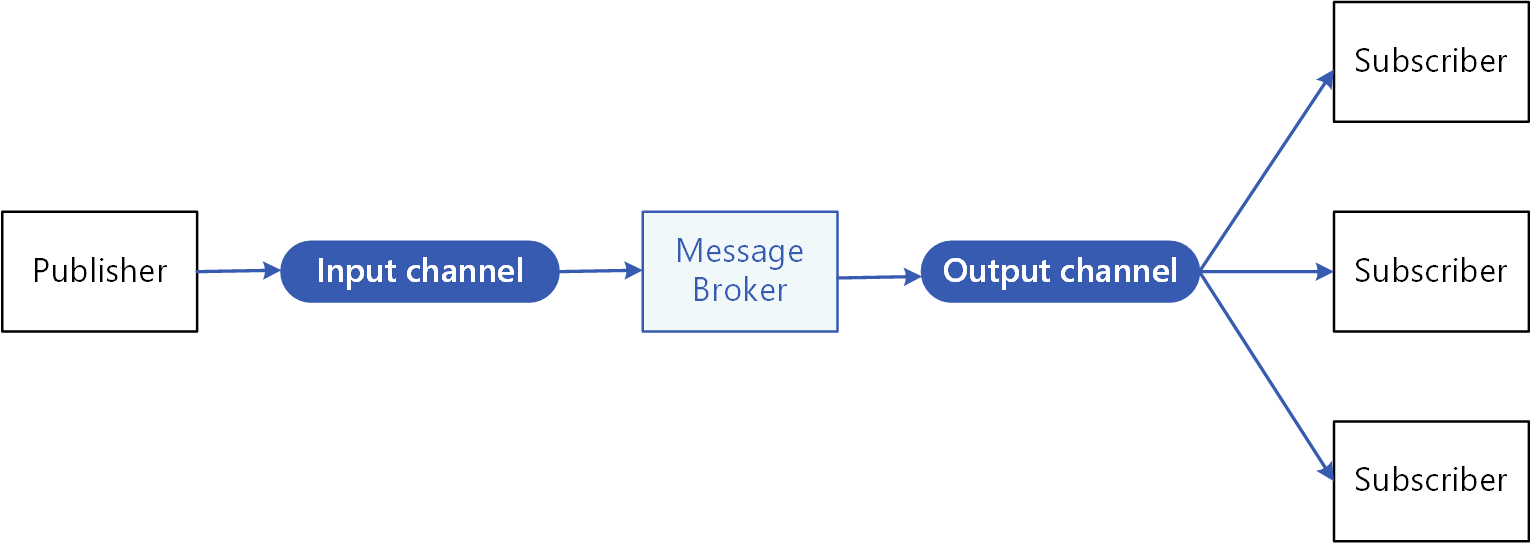
**3. \*\*Intermediary (Message Broker):\*\*** Copies messages from the input channel to the output channels of all interested consumers.

**4. \*\*Message Flow:\*\*** Sender sends messages to the input channel. The intermediary ensures messages are delivered to relevant consumer output channels.

**5. \*\*Benefits:\*\*** Decouples sender from consumers, allows multiple subscribers, and supports asynchronous message delivery.

1. **\*\*Use Cases:\*\*** Ideal for cloud-based systems, event-driven architectures, and scalable applications.
2. A message is a packet of data. An event is a message that notifies other components about a change or an action that has taken place.

**The following diagram shows the logical components of this pattern:**



**Pub/sub messaging has the following benefits:**

1. **\*\*Decoupling:\*\*** Pub/sub messaging allows subsystems to communicate without being directly connected. This decoupling means that each subsystem can function independently, making it easier to manage and maintain the overall system. Even if one or more receivers (subscribers) are offline, the messaging system can properly manage messages and deliver them once the subscribers are available.

2. **\*\*Scalability & Responsiveness:\*\*** With pub/sub messaging, the sender (publisher) can quickly send a single message to the input channel and then focus on its core processing responsibilities. The messaging infrastructure takes care of delivering messages to all interested subscribers. This separation of responsibilities improves the sender's responsiveness and makes the system more scalable, as the messaging system can handle the distribution of messages efficiently.

3. **\*\*Reliability:\*\*** Asynchronous messaging enhances the reliability of the application. It allows the system to continue running smoothly even under increased loads or during intermittent failures. Messages are buffered, ensuring they are delivered to the intended consumers, reducing the risk of message loss.

4. **\*\*Deferred/Scheduled Processing:\*\*** Pub/sub messaging allows subscribers to defer picking up messages until off-peak hours or process messages according to specific schedules. This flexibility is useful for managing workloads during peak times and optimizing resource utilization.

5. **\*\*Simpler Integration:\*\*** Pub/sub messaging simplifies integration between systems using different platforms, programming languages, or communication protocols. It also enables seamless communication between on-premises systems and applications running in the cloud, making it easier to build distributed systems.

6. **\*\*Asynchronous Workflows:\*\*** Pub/sub messaging is well-suited for implementing asynchronous workflows across an enterprise. It allows components to communicate without blocking each other, making it easier to build complex systems with asynchronous interactions.

7. **\*\*Testability:\*\*** Pub/sub messaging improves testability as channels can be monitored and messages can be inspected or logged. This allows for more comprehensive integration testing, ensuring that messages are delivered correctly and reliably.

1. **\*\*Separation of Concerns:\*\*** Each application can focus on its core capabilities without worrying about the details of message delivery. The messaging infrastructure handles everything required to reliably route messages to multiple consumers, promoting a cleaner and more modular design in applications.

**Issues and considerations:**

| **Consideration** | **Explanation** |
| --- | --- |
| **Existing technologies** | Use existing messaging tools like Azure Service Bus, Event Hubs, Redis, RabbitMQ, or Apache Kafka instead of creating your own. These tools support the pub/sub model, making messaging easier. |
| **Subscription handling** | Make sure there are ways for consumers to join or leave the channels they are interested in. Think of it like subscribing to newsletters – you can sign up or unsubscribe as needed. |
| **Security** | Keep the messaging channels secure so only authorized users or applications can access them. This prevents unauthorized access, just like a locked door keeps unwanted people out. |
| **Subsets of messages** | Allow consumers to receive only the messages they care about. It's like getting specific emails in a group inbox. They can filter messages based on topics or content, or use wildcard rules to select certain types of messages. |
| **Bi-directional communication** | In some cases, subscribers may need to talk back to the sender. To handle this, set up a separate communication channel for replies. It's like using a phone to call customer support after receiving an email. |
| **Message ordering** | Understand that messages may arrive in different orders. Design your system so that it doesn't rely on the exact order of messages. It's like reading emails – you don't always read them in the order they were sent, but you handle each one properly. |
| **Message priority** | Sometimes, certain messages need special attention. Use a priority queue to ensure important messages get delivered first. Think of it like marking some emails as high priority to deal with them first. |
| **Poison messages** | Handle messages that are broken or need resources that are not available. Instead of keeping them in the inbox, put them aside to avoid causing problems. It's like removing damaged packages from a delivery service's workflow. |
| **Repeated messages** | Sometimes the same message might be sent more than once. Handle this by identifying duplicates and removing them. It's like receiving multiple copies of a letter, but you only need to read it once. |
| **Message expiration** | Some messages have a limited lifespan and become irrelevant after a certain time. Make sure to check if a message has expired before acting on it. It's like checking the expiration date on food items before deciding whether to eat them. |
| **Message scheduling** | Occasionally, messages need to be sent or processed at specific times. Use scheduling to delay the delivery or processing until the right moment. It's like setting an alarm to remind you to do something at a particular time. |

**When to use this pattern:**

**Use the pub/sub messaging pattern in the following scenarios:**

1. \*\*Broadcasting to Many Consumers:\*\* When you need to send information to a large number of consumers, like broadcasting announcements or updates, pub/sub messaging efficiently distributes the messages without the need for direct connections.

2. \*\*Interoperability between Applications:\*\* When your application needs to communicate with independently-developed applications or services that might be built on different technologies or languages, pub/sub messaging allows seamless integration.

3. \*\*Non-Real-Time Communication:\*\* If your application can provide information to consumers without requiring immediate responses, pub/sub messaging is a good fit. Consumers can receive and process messages at their own pace.

4. \*\*Eventual Consistency Model:\*\* Pub/sub messaging works well with systems that embrace eventual consistency, where data might not be immediately synchronized across all components but eventually becomes consistent.

5. \*\*Different Availability Requirements:\*\* When the consumers have varying availability requirements or operate on different uptime schedules than the sender, pub/sub messaging allows them to receive messages when they are online and ready to process them.

In summary, the pub/sub messaging pattern is ideal when you have multiple consumers to inform, want to integrate diverse applications, don't need real-time responses, deal with eventual consistency, or have consumers with varying availability. It simplifies communication, promotes scalability, and enhances the flexibility of your distributed applications.

**This pattern might not be the most suitable choice in the following situations:**

1. \*\*Few Consumers with Different Information Needs:\*\* If your application only has a few consumers, and each of them requires significantly different information from the producing application, using a pub/sub messaging pattern might introduce unnecessary complexity. In such cases, a more direct and tailored approach, like point-to-point messaging or request-reply communication, could be a better fit.

2. \*\*Near Real-Time Interaction:\*\* Pub/sub messaging is inherently asynchronous, which means that the producing application doesn't receive immediate responses from consumers. If your application requires near real-time interaction and depends on immediate feedback or responses from consumers, pub/sub messaging might not be the ideal choice. Instead, consider using synchronous communication patterns that allow for real-time interactions, like request-reply or synchronous message processing.

In summary, while the pub/sub messaging pattern offers numerous benefits for certain scenarios, it might not be the most appropriate solution when you have only a few consumers with diverse information needs or when your application relies on near real-time interactions with consumers. In those cases, consider other communication patterns that better suit the specific requirements of your application.

**Example:**

The following diagram shows an enterprise integration architecture that uses Service Bus to coordinate workflows, and Event Grid to notify subsystems of events that occur.

