Publisher-Subscriber (An Architectural Design Pattern)

What is Publisher-Subscriber Pattern?

The Publish/Subscribe pattern, sometimes known as pub/sub, is an architectural design pattern that enables publishers and subscribers to communicate with one another. In this arrangement, the publisher and subscriber rely on a message broker to send messages from the publisher to the subscribers. Messages (events) are sent out by the host (publisher) to a channel, which subscribers can join.

Compared to older design patterns like message queuing and event brokers, Pub/Sub is more versatile and scalable. The key to this is that Pub/Sub allows messages to flow between different system components without the components knowing one other’s identities.

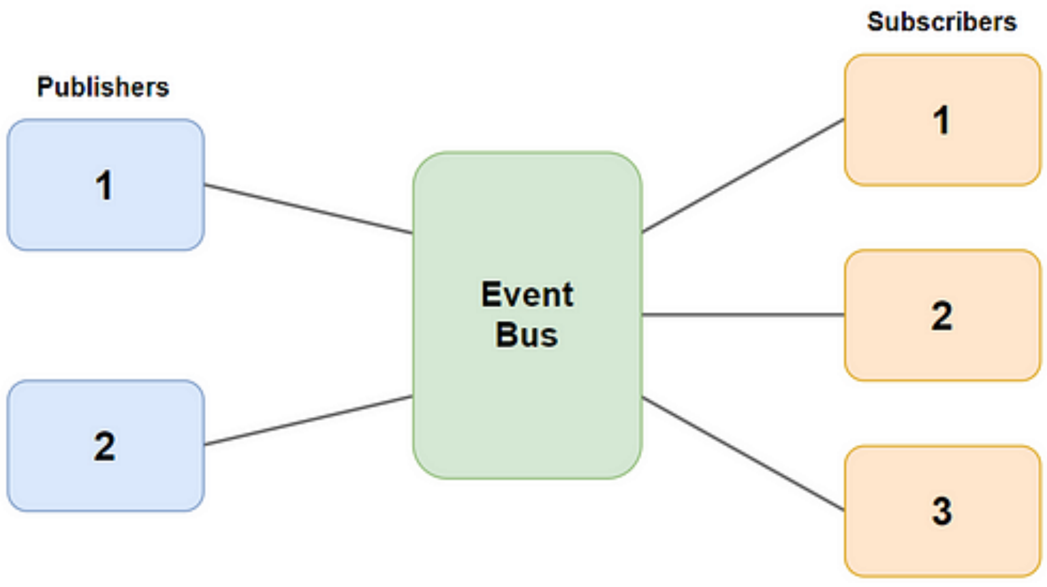
A Real-life Example of Pub-Sub Pattern

The Publish/Subscribe (Pub/Sub) model is commonly used in social networks through features such as "following." To better understand how this works, let's consider an example of a social network for sharing recipes.

In this network, users can share their own recipes and follow the recipes of other users. When sharing a recipe, the user can categorize it by topic, such as meal or ingredient seasonality. When a user follows another user, they are subscribing to the recipes that the friend publishes.

Subscribers, or followers, have the option to view all of the recipes published by the user they are following or only those that match their interests. They can also create filters to exclude certain types of recipes, such as those containing certain ingredients.

Users can follow as many other users as they like, so their timeline will be filled with recipes from a variety of sources. However, each recipe is only published once, by the original user.



Here Event Bus is responsible for routing messages to the appropriate subscribers. It does this by keeping track of which subjects each subscriber is subscribed to.

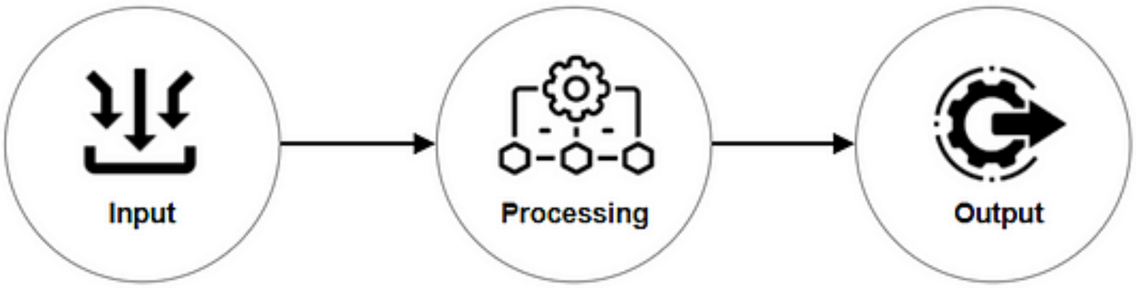
Publishers decide what topics their messages will belong to, and the Event Bus filters the messages by topic before delivering them to the relevant subscribers. For example, if a publisher sends a message with Topic A, it will be forwarded to any subscribers who have subscribed to Topic A. Similarly, a message with Topic B will be delivered to subscribers of Topic B.

It's important to note that if a publisher sends a message with an incorrect topic, it will only be delivered to subscribers of that incorrect topic. In the diagram above, if a publisher sends a message with Topic A but mistakenly labels it as Topic B, it will only be sent to Subscriber 2 and Subscriber 3.

How Does Publish-Subscribe Pattern Work?

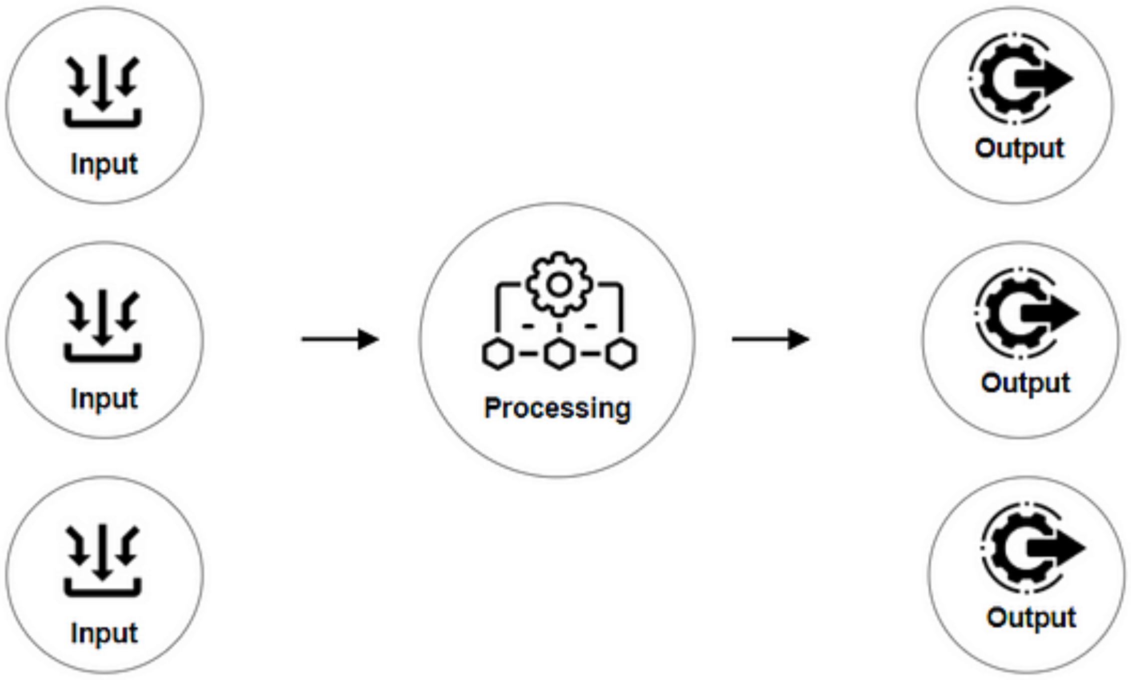
Building reusable arrangements of modules and their interconnections is the foundation of software design patterns. In a UML design diagram, these modules are often classes or objects. Modern architectural patterns, on the other hand, see modules as larger, self-executing processes scattered over distributed systems.

To fully comprehend the benefits of the Pub/Sub design, you must first understand the fundamental pattern upon which an information system is formed and then trace its progression into a distributed system. An information system is often made up of a generic set of software modules that are organized in this simple sequential structure.

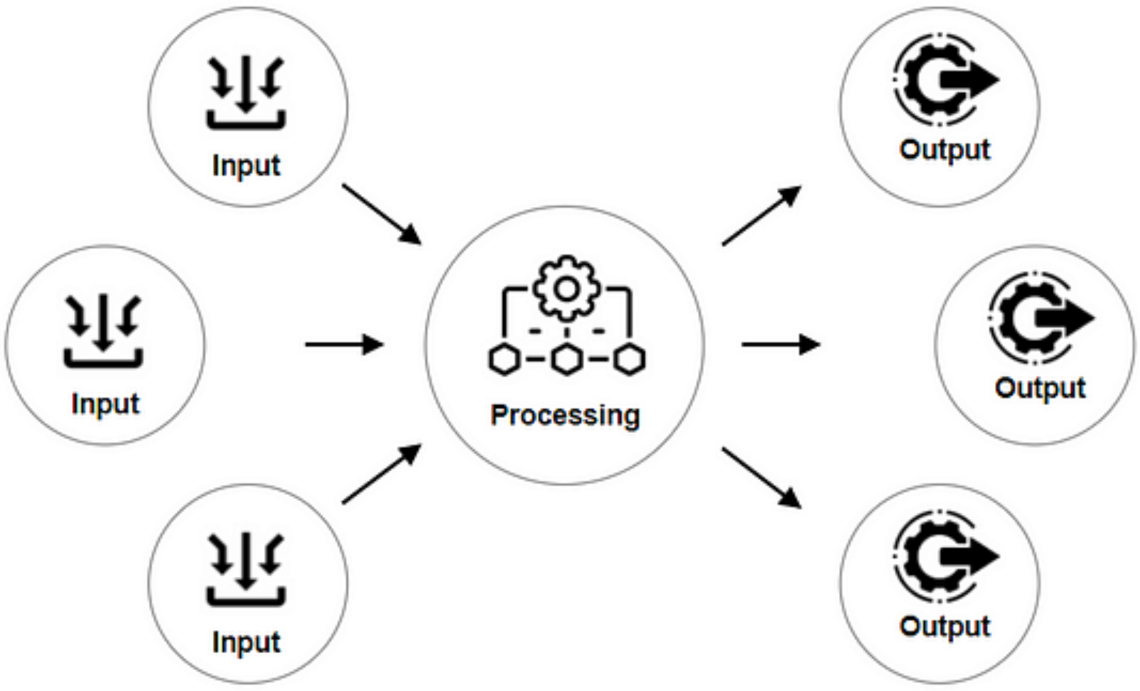


Consider the diagram above as simple software made up of three parts. The input module receives user input and converts it into a message that is sent to the processing module. The data is processed by the processing module and sent to the output module as a new message. The data is shown on the user’s screen using the output module.

The real world, on the other hand, is seldom so simple. For handling concurrent queries at a suitable size, the system will require several inputs and output modules.



The system faces the difficulty of routing messages from input modules to their appropriate output modules at this scale. The input and output modules will require an addressing mechanism to handle this challenge. The messages will be processed and routed to the relevant recipient based on an address by the processing module. In order to solve the routing problem, all three modules work together.



The system will be able to manage thousands of concurrent connections at an Internet-scale. Users from all over the world will send and receive messages using the system. It must also be able to handle a large number of users from all over the world.

The system modules, on the other hand, will not work as planned at such a vast scale.

- The load is too much for the processing module to handle. The load must be divided across numerous processing modules due to the high volume and geographical spread.
- The dynamics of input and output change at this scale. The use of pre-defined addressing between modules adds a significant amount of overhead.

The first issue can be solved by using numerous processing units. This has the effect of horizontally separating the system. This, however, adds to the routing complexity. The messages must now be routed to the appropriate processing module by the input modules.

At the internet scale, attaching module-specific routing metadata to messages becomes a bottleneck. Under these conditions, the design of message transfer from one module to the next necessitates a complete overhaul.

Advantages of Publisher-Subscriber Pattern

- **Low coupling on the publisher's end:** Publishers do not need to know the number, identity, or types of messages that subscribers are interested in. They simply output data in response to the correct events through their API. This allows for flexibility and scalability, as new subscribers can be easily added to the system without affecting the publisher.
- **Reduced cognitive load for subscribers:** Subscribers do not need to worry about the inner workings of the publisher or have access to the publisher's source code. They can only communicate with the publisher using the publisher's public API, which simplifies their understanding of the system.
- **Separation of concerns:** The simplicity of the Pub/Sub architecture

(data flows one way from publishers to subscribers) allows developers to practice fine-grained separation of concerns. This means that different message types can be split into distinct categories that each fulfill a single, straightforward purpose. For example, data with the topic "/cats" could contain information about cats, while data with the topic "/dogs" could contain information about dogs.

- **Improved testability:** The fine-grained control over topics makes it easy to confirm that the various event buses are transmitting the necessary messages.
- **Improved security:** The Pub/Sub architecture is well-suited for the security principle of assigning minimal privileges or information. Developers can easily create modules that are subscribed to only the minimum number of message types needed to function.

Disadvantages of Publisher-Subscriber Pattern

The inflexibility of data sent by the publisher

The publish/subscribe approach can introduce a high level of semantic coupling in the messages that are sent between publishers and subscribers. This means that it can be difficult to modify the data structure of these messages once it has been established. To change the format of the messages, all of the subscribers must be updated to accept the new format, which can be challenging or impossible if the subscribers are external. This is a common problem with versioned APIs.

One solution to this issue is to use a versioned message format that allows subscribers to validate the format they are receiving. However, this assumes that subscribers are correctly consuming the versioning information.

Another option is to use versioned endpoints, such as "/APIV0/" and "/APIV1/," to maintain backward compatibility. This approach has the disadvantage of requiring developers to support multiple versions, which can be time-consuming.

Instability of Delivery

One disadvantage of the publish/subscribe model is that it can be difficult to determine the health of subscribers. The publisher does not have complete information about the systems that are listening to the messages, which can lead to problems.

For example, logging systems often use the publish/subscribe model. If a logger subscribed to the "Critical" message type crashes or becomes stuck in an error state, it may miss important messages. Any services that rely on error warnings will then be unaware of issues with the publisher.

This problem is not unique to the publish/subscribe model, and can occur in any client/server system. However, one advantage of pub/sub is that it allows for high levels of redundancy by allowing multiple instances of a logger to run concurrently with minimal additional system effort.

To mitigate this risk, changes to the design could be implemented, such as requiring receipts of received messages. This would allow the publisher to receive feedback on the status of the subscribers.

Additional Reading: https://docs.microsoft.com/en-us/azure/architecture/patterns/publisher-subscriber

Thanks to Navtosh for his contribution in creating the first version of this content. If you have any queries/doubts/feedback, please write us at contact@enjoyalgorithms.com. Enjoy learning, Enjoy system design, Enjoy algorithms!