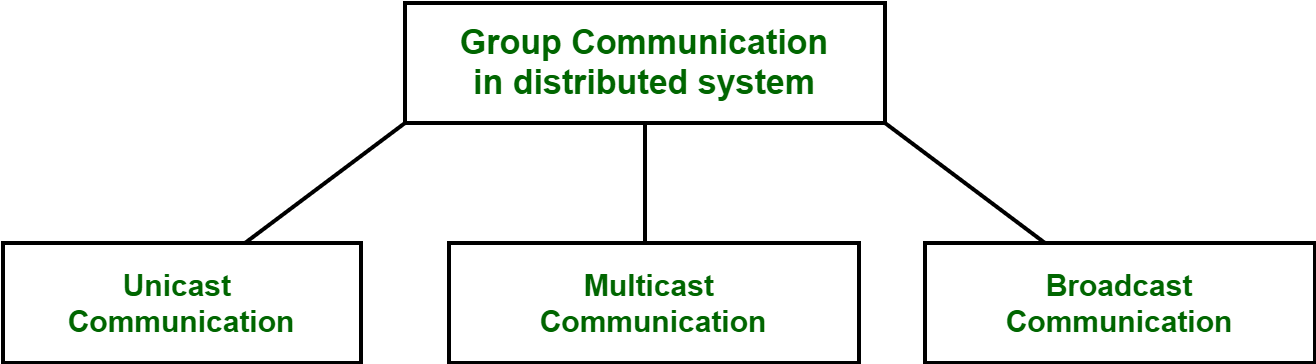
**Week 2: Communication and Middleware**

* **Lesson 3: Communication in Distributed Systems**
  + Communication models and protocols.
  + Sockets, message-passing, and remote procedure calls (RPC).
* **Lesson 4: Middleware and Message Brokers**
  + Introduction to middleware.
  + Message queuing systems (e.g., RabbitMQ, Apache Kafka).
* **Class Activity**: Hands-on lab on socket programming.

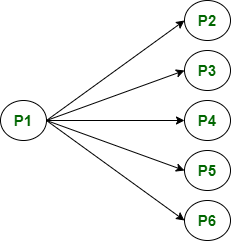
**Group Communication in Distributed Systems**

**Communication** between two processes in a distributed system is required to exchange various data, such as code or a file, between the processes. When one source process tries to communicate with multiple processes at once, it is called **Group Communication**. A group is a collection of interconnected processes with abstraction. This abstraction is to hide the message passing so that the communication looks like a normal procedure call. Group communication also helps the processes from different hosts to work together and perform operations in a synchronized manner, therefore increasing the overall performance of the system.



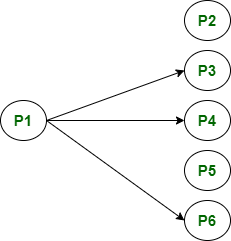
**Types of Group Communication in a Distributed System:**

* **Broadcast Communication:** When the host process tries to communicate with every process in a distributed system at the same time. Broadcast communication comes in handy when a common stream of information is to be delivered to every process in the most efficient manner possible. Since it does not require any processing whatsoever, communication is very fast in comparison to other modes of communication. However, it does not support a large number of processes and cannot treat a specific process individually.



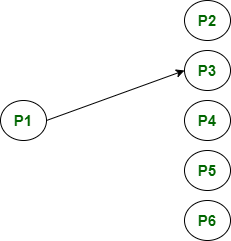
*A broadcast Communication: P1 process communicating with every process in the system*

* **Multicast Communication:** When the host process tries to communicate with a designated group of processes in a distributed system at the same time. This technique is mainly used to find a way to address the problem of a high workload on the host system and redundant information from the process in the system. Multitasking can significantly decrease the time taken for message handling.



*A multicast Communication: P1 process communicating with only a group of the process in the system*

* **Unicast Communication:** When the host process tries to communicate with a single process in a distributed system at the same time. Although, the same information may be passed to multiple processes. This works best for two processes communicating as it has to treat a specific process only. However, it leads to overheads as it has to find the exact process and then exchange information/data.



*A unicast Communication: P1 process communicating with only the P3 process*

**Role of Middleware in Distributed System**

Middleware is an intermediate layer of software that sits between the application and the network. It is used in distributed systems to provide common services, such as authentication, authorization, compilation for best performance on particular architectures, input/output translation, and error handling.

In distributed systems, middleware is a software component that provides services between two or more applications and can be used by them. Middleware can be thought of as an application that sits between two separate applications and provides service to both. In this article, we will see the role of middleware in distributed systems.

With all the miscommunication going around these days, enterprises need to start using software solutions that streamline communications across departments. One such product that fits this description, is known as Middleware, which allows organizations to implement their processes seamlessly by integrating all components of the enterprise. In this article, we will examine the role of middleware in distributed systems and how it can help businesses to evolve.

However, there are numerous middlewares around, each having its own set of responsibilities. This can be attributed to the fact that different types of middleware are designed for different scenarios and each type has its own set of requirements and goals.

**Example:**

Message-oriented middleware is designed to transport messages between two or more applications and is best suited for distributed applications that require transaction-oriented messaging. It could be used to monitor network traffic flows or to monitor the health of a distributed system.

**Key Points:**

* Distributed systems are becoming increasingly complex as they are involved in activities spanning an ecosystem of partners, suppliers, and customers.
* One way to achieve this integration is through the use of middleware. In recent years, there has been a rise in the number of middleware solutions available in the market.
* However, due to this array of choices available it has become vital for enterprises to first identify what type of solution they require before making a purchasing decision.

**Advantages of Middleware in Distributed Systems:**

* Middleware is an intermediate layer of software that sits between the application and the network. It is used in distributed systems to provide common services, such as authentication, authorization, compilation for best performance on particular architectures, input/output translation, and error handling.
* Middleware offers several advantages to distributed systems. Middleware can be modularized from the application so it has better potential for reuse with other applications running on different platforms.
* Application developers can design Middleware so it’s sufficiently high-level that it becomes independent of specific hardware environments or operating system platforms which simplifies porting applications developed on one type of platform onto another without rewriting code or without resorting to inefficient and expensive binary compatibility toolsets such as cross-compilers.

**Middleware Key Challenges of Developing Distributed Systems:**

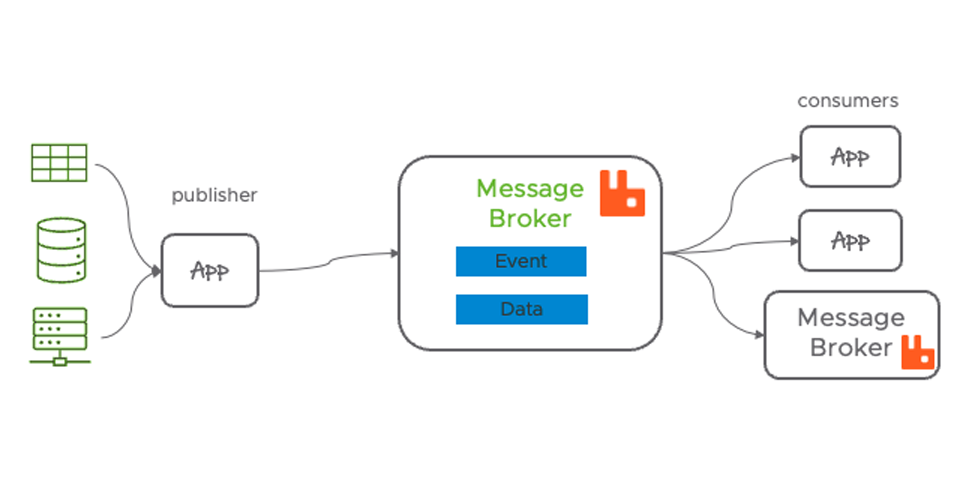
* Developing distributed systems can be not just challenging but also very time-consuming, especially in building and maintaining large-scale infrastructures. This is because it requires considering the software architecture of both the client and server as well as the underlying network topology.
* Middleware technologies help developers by handling tough tasks such as cross-platform application development, network management, and security enforcement so that they are manageable by software teams.
* The developers of Middleware Technologies are focusing on designing new technologies to address both current and emerging challenges to enable their users to build scalable distributed applications quickly, easily, and cost-efficiently.

Middleware supports a wide range of distributed applications, by providing services that span the application and network tiers. Therefore, it is more than likely that middleware will continue to evolve and be adapted for use in more application scenarios. Its role in distributed systems is expected to increase as enterprises move toward becoming globally integrated enterprises.

**What is a Message Broker?**

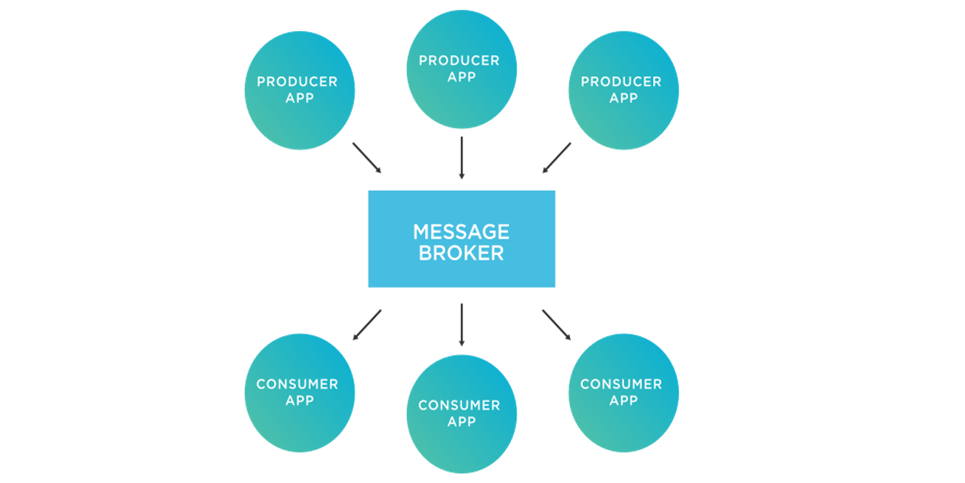
Modern computing relies on a distributed architecture in which separate software components—often in different environments—coordinate with each other. Messaging and event streaming are essential to support microservices, serverless, and other cloud-native application architectures.

* One of the programming challenges created by microservices and other distributed architectures is communication between application components. Message brokers are designed to provide a common communication mechanism that scales with application needs. In addition to messaging, message brokers typically provide services such as translation, routing, persistence, and delivery, streamlining communications while reducing the burden on software developers to handle these capabilities in application code.
* An enterprise-grade message broker helps to ensure fast reliable communication between servers, applications, and devices in multi-cloud environments, with reliable asynchronous messaging for passing events, commands, data, and responses among producing and consuming applications or services. A distributed message broker simplifies communication between microservices while providing security, automated management, and high performance.
* Distributed microservices applications communicate with each other by exchanging data in the form of messages. Message brokers can help integrate disparate applications and services across a wide variety of languages and platforms. Asynchronous messaging allows producers and consumers to send and receive messages independently and at different times without blocking.



**Message broker architecture**

A message broker provides the necessary plumbing to facilitate the exchange of messages (discrete, structured data). A producer or publisher sends the message, and a consumer or subscriber receives the message. A message broker accepts and if necessary, translates messages sent by a producer, then organizes and holds messages in a message queue until received by the consumer(s).



**Message Broker vs Message Queue**

Once a message is received, a message broker typically maintains the message in a message queue until a consumer/subscriber can retrieve it. Message queues are data structures that provide reliable temporary storage for messages until they are read. They store messages in the same order in which they were sent, so they can be processed in sequence. Producers and consumers can share the same queue, and messages can be routed to one or many message queues. Message queues help ensure information doesn’t get lost or destroyed due to incompatible systems, network latency, or application failure. Because the message queue is such an important component of a message broker, the designation “MQ,” (message queue) appears in the name of message brokers such as RabbitMQ and ActiveMQ.

**Message Broker Vs Event Streaming Platform**

* While many use message broker and event streaming somewhat interchangeably, they are not the same. The confusion arises because of overlap in the use cases for the two technologies.
* A message broker stores a message in a queue until it is consumed and then deletes the message. Message brokers offer flexible routing and reliability. Event streaming platforms append events to the end of a log and don’t delete events as they are consumed.
* Streaming facilitates raw throughput. Consumers can start reading the stream at any point and see the entire history. An “event” could be the same as a message, but with streaming applications, it is often the stream of events that is of interest. Multiple events are often processed together.
* For example, a low-fidelity temperature sensor might report the temperature once per second. Rather than use the temperature reported in a single event, it may be more accurate to average the last ten temperature readings and use the average as a measure of the temperature. Similarly, a stock trading application might analyze changes in a stock price over time, processing a series of events (multiple ticks in the price) in multiple ways before taking an action or generating a notification.

**Message Broker vs Message Bus**

* A message bus—or enterprise service bus (ESB)—is a type of middleware built on a central system that enables application-to-application communication. Primarily used in old-style Service Oriented Architecture (SOA), ESBs formalize data formats and protocols for use by multiple applications.
* SOA and ESB technologies remain in wide use but are being replaced by messaging and streaming in modern microservices applications. More and more organizations are migrating to open-source message brokers and embracing microservices architectures and cloud-native frameworks.

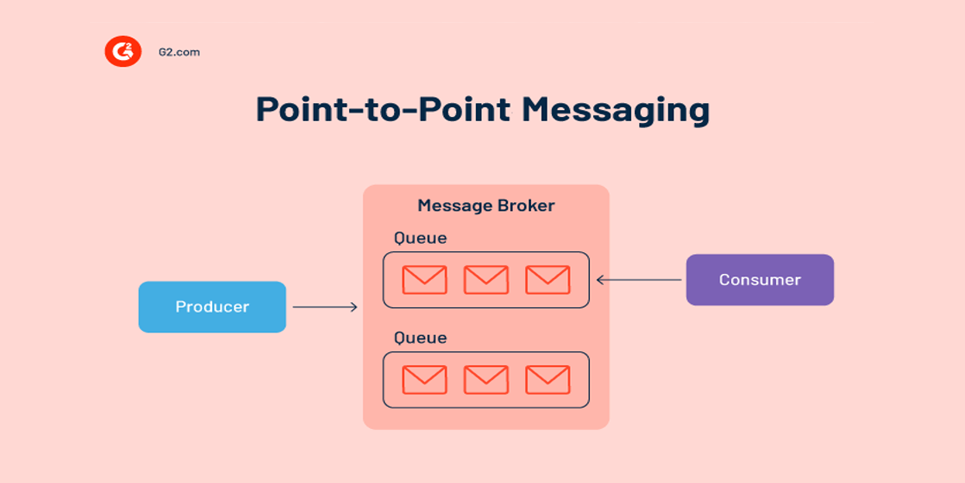
**Common Message Broker Models**

The following terms may be employed depending on the messaging model being employed:

* **The producer or publisher**is the message sender
* **The consumer or subscriber** is the message recipient
* **Message queue or topic** is where a message is routed to

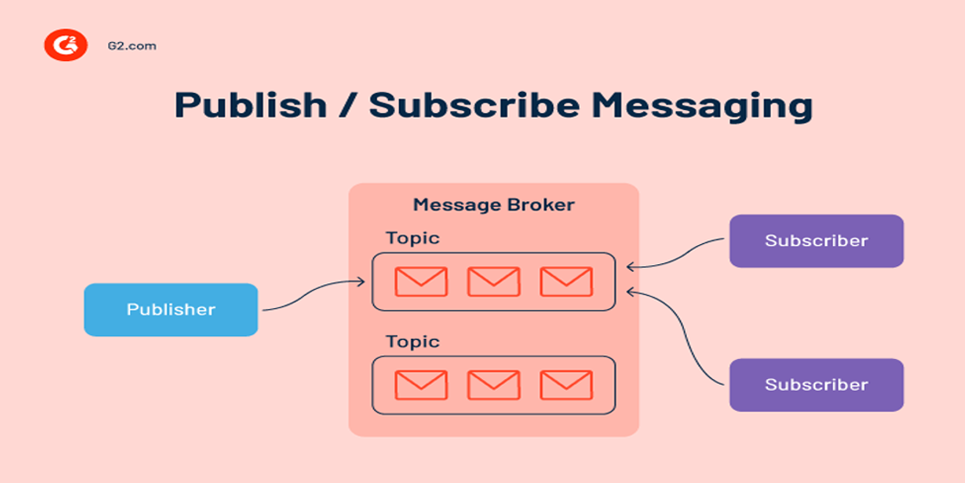
**Point-to-point messaging model**

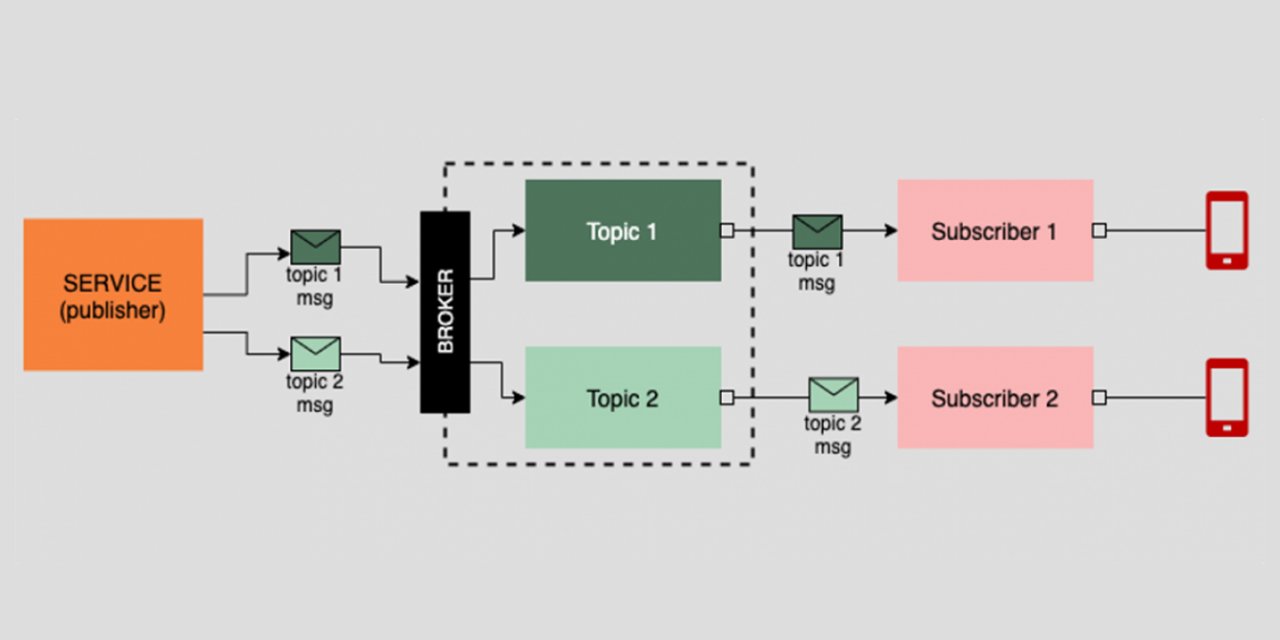
A point-to-point messaging model is used when a producer sends a single message directly (point-to-point) to a consumer. Messages could include data to be stored, a command to be carried out, or any other information that needs to be shared between the producer and the consumer.



**Publish-subscribe messaging model**

When a message producer needs to send the same message to many recipients simultaneously, then a publish-subscribe model is used. With this messaging pattern, the message broker publishes a message to a topic, where it is organized, cached, processed, and delivered to subscribers. One or many publishers can publish on the same topic, and messages are available to anyone who subscribes to that topic. Publish-subscribe provides a flexible way of communicating between multiple microservices and applications.  





**What to keep in mind when using a message broker**

Applications often have trouble communicating with other applications or services deployed in different environments. In an increasingly distributed environment, a message broker helps communicate and coordinate with services running on multiple platforms. The message broker you choose depends on your use case and operations. 

**When should I implement a message broker?**

Many microservices applications use APIs to communicate directly with each other, so you might wonder why a message broker is needed. One of the main reasons to choose a message broker is scale. Once application components begin exchanging thousands of messages, the performance and resiliency of a message broker begin to justify the complexity that comes with the extra software.

Several common use cases lend themselves to message broker use:

* **Microservices:** As the number of microservices that make up an application increases, a message broker can rationalize communications between services. For example, suppose a service needs to send an identical message to ten other services. Rather than make 10 API calls, with a message broker you simply send one message to the broker.
* **Long-running tasks:** Sometimes a message triggers a long-running task. For instance, when a video is uploaded, a message may be sent to trigger a series of tasks such as transcoding, compression, distribution, etc. These tasks may take many hours to run in the background.
* **Mobile apps:** Mobile devices and apps may not always be running or reachable. A message broker can ensure that messages (notifications) are available and can be received in order when the mobile device or application regains service.
* **Transactions:** With transactional systems, you often need to ensure that a message is only processed once and not duplicated. Because message brokers delete messages after they are consumed, they can be used to ensure this happens.

**Common message brokers**

There are many open-source and cloud-based message brokers available.

* **RabbitMQ** is an open-source distributed message broker that facilitates the efficient delivery of messages in complex routing scenarios. It delivers flexible routing and command and response between applications. RabbitMQ has recently added event-streaming capabilities that model a Kafka-like append-only log on top of a traditional RabbitMQ message queue.
* **VMware RabbitMQ** is an enterprise-grade message broker with event streaming capabilities based on open-source RabbitMQ.
* **Apache ActiveMQ** is an open-source message broker that facilitates communication between microservices and supports protocols across a range of languages and platforms.
* **Apache Kafka** describes itself as a distributed event streaming platform rather than a message broker. However, many use Kafka to satisfy message broker use cases. Kafka’s scalability enables applications to publish and subscribe to multiple message streams simultaneously. Kafka uses a distributed append-only log; messages are written to the end of a log that’s persisted to disk.
* **Apache Pulsar** provides cloud-native, multi-tenant, high-performance messaging and queuing for microservices and serverless functions.