4. Microservice Architecture

ASP.NET & Microservice Messaging

Agenda

- Microservice Messaging

Issues with Synchronous Communication

- why messaging is a viable option
- Imagine this scenario
  - you are a coach of a sports team for kids. An issue comes up that causes the rescheduling of a game
  - Get the list of people and call them one by one. You call the other coaches, plus the umpires/judges and all of the parents
  - Now imagine that call one of the parents that oversee the venue kiosk.
  - That parent then calls the owner of the venue kiosks and discovers they cannot attend on the proposed date.
  - Each step in the communication chain has transitive latency.
  - This transitive latency is problematic for you, who never intended to spend so much time trying to communicate with others

Issues with Synchronous Communication

- Other issues exist for you and the other coaches
- The list of people to contact varies
- Just because a coach is contacted does not mean all the parents must be contacted as well.
- And it is never known who has the correct list of contact information
- What would happen if anyone were not notified of the schedule change?
- What if the chain of communication involved other downstream pieces?
- What if there is a failure to notify them?
- This allows for cascading failures

Issues with Synchronous Communication

- You agreed to be the coach to help the kids, not worry about communication issues.
- Each call has the potential of not being answered and information failing to be sent and understood.
- Also, each call takes the coaches' time because they are synchronous.
- Each caller must wait for someone to answer, receive the message, and acknowledge what needs to happen next.

Limits of RPC

- The preceding scenario is how RPC works
- This shows the biggest drawback of RPC with microservices; processes must wait for a response, that is, synchronous communication
- Another issue with this type of communication is handling the number of simultaneous calls. For example, if more calls are coming in than the microservice can handle, the caller will see either severe latency or no contact at all. This forces the caller to use retry policies or the circuit breaker pattern.

Limits of RPC

- Scaling microservices horizontally requires a load balancer
- And just because another instance of a microservice exists does not mean it is instantly registered in the load balancer (+ latency) (+ effort to register)
- adding other microservices to a business process
- If the caller is only aware of one microservice, then it must be altered to know about the other microservices.
- This means more code changes to each caller that must be aware of the others.
- And how is versioning to be handled? There are even more code changes to handle adapting to a different version.

Messaging

- what if there was a way to broadcast the information once?
- Perhaps group text or an email would suffice. Then everyone receives that information independently, and each recipient reacts accordingly. That is the purpose of messaging
- Using messaging in the microservices architecture allows independent pieces to communicate without knowing the location of each other.
- Messaging solves many of the problems identified in the preceding example story.

Messaging

- If the coach had a messaging system in place, contacting some people or everyone would be easier.
- Also, adding, removing, and updating contact information would be simpler.
- So, where does messaging fit in an architecture?

Architecture

- simple messaging architecture to reach a microservice in a disconnected way
- using messaging with microservices allows for a dynamic and reactive architecture
- With an event-driven architecture, the microservices react to messages sent as a response to an event.
- Perhaps a user has committed to the creation of an order.
- Or maybe there is a change to the inventory of an item

Reasons to Use Messaging

- The list of reasons includes
  - Loosely coupled
  - Buffering
  - Scaling
  - Independent processing

Loosely Coupled

- By using messaging, the sender and the microservices (message consumers) are loosely coupled.
- The sender of a message does not need to know anything about the microservices receiving the message.
- This means that microservices do not need to know the endpoints of others.
- It allows for the swapping and scaling of microservices without any code changes.
- autonomy for the microservices
- microservices to only be tied together where they fit to fulfill business processes and not at the network layer

Buffering

- there are opportunities for downtime or intermittent network connection issues
- Message brokers utilize queues that provide a buffer of the messages
- During times of issues, the undeliverable messages are stored.
- Once the consumers are available again, the messages are delivered.
- This buffering also retains sequencing.
- The messages are delivered in the order they were sent.
- Maintaining the message order is known as First In, First Out (FIFO).

Scaling

- any production applications or distributed processing, you will have multiple instances of your microservices.
- This is not just for high availability but allows messages to be processed with higher throughput.

Independent Processing

- change over time
- You can have a solution to generate orders and manage the shipping of the products.
- Later, you can add other microservices to perform additional business processes by having them become message consumers.
- The ability to add microservices to business processes as consumers allows them to be independent information processors.

Message Types

- There are multiple types of messages, depending on the need for communication.
- These message types play a role in fulfilling various responsibilities in business processes

Query

- Consider the scenario of contacting your favorite store to see if they have the latest game in stock. (Pretend they do not yet have a site to sell online.)
- You want to purchase three copies quickly, but the store is not open.
- You send them an email or perhaps filling out an inquiry form on their website for that information.
- This is using a query message type in an asynchronous communication style.

Query

- Although the query could be done synchronously, like a phone call, it does not have to be done that way.
- When the store opens, they reply that they do have the game in stock.
- The reply also includes how many are currently in stock and suggestions on other similar games.
- A reply to a query is the Document message type. The document contains the information requested and any supporting information.

Command

- You reply confirming you want to secure those three copies, and you are on your way to the store.
- The message to the store letting them know that they are to hold the items aside for you is a command.
- The command message type instructs what the consumer is to do without explicitly requiring a reply.

Event

- You arrive at the store and purchase the games.
- Later, you receive an email about the purchase.
- The store thanks you for the purchase and includes a digital receipt.
- That email was sent due to an event in their system.
- An order was created for you, and as a result, you received the email.

Event

- When you purchased the games, the system notified other areas of the system about the order.
- The event kicked off processes in other parts of the store's system.
- Each independent process received that event and reacted.
- For example, the system processed the payment, took the item out of inventory, and notified the shipment department of where the games were to be sent.

Event

- The architecture of having multiple consumers listening to the same event is called Publish/Subscribe (Pub/Sub).
- The publisher of the event sends out the message for others to process.
- The consumers subscribe to the event and process the message differently.
- For example, the payment processor consumer will act on the message differently than an inventory management consumer

Message Routing

- For messages to go from a publisher to the consumers, there must be a system to handle the messages and the routing.
- This system provides message routing to the consumers that have subscribed to the different messages.
- There are two types of these systems:
  - brokered and
  - broker-less.

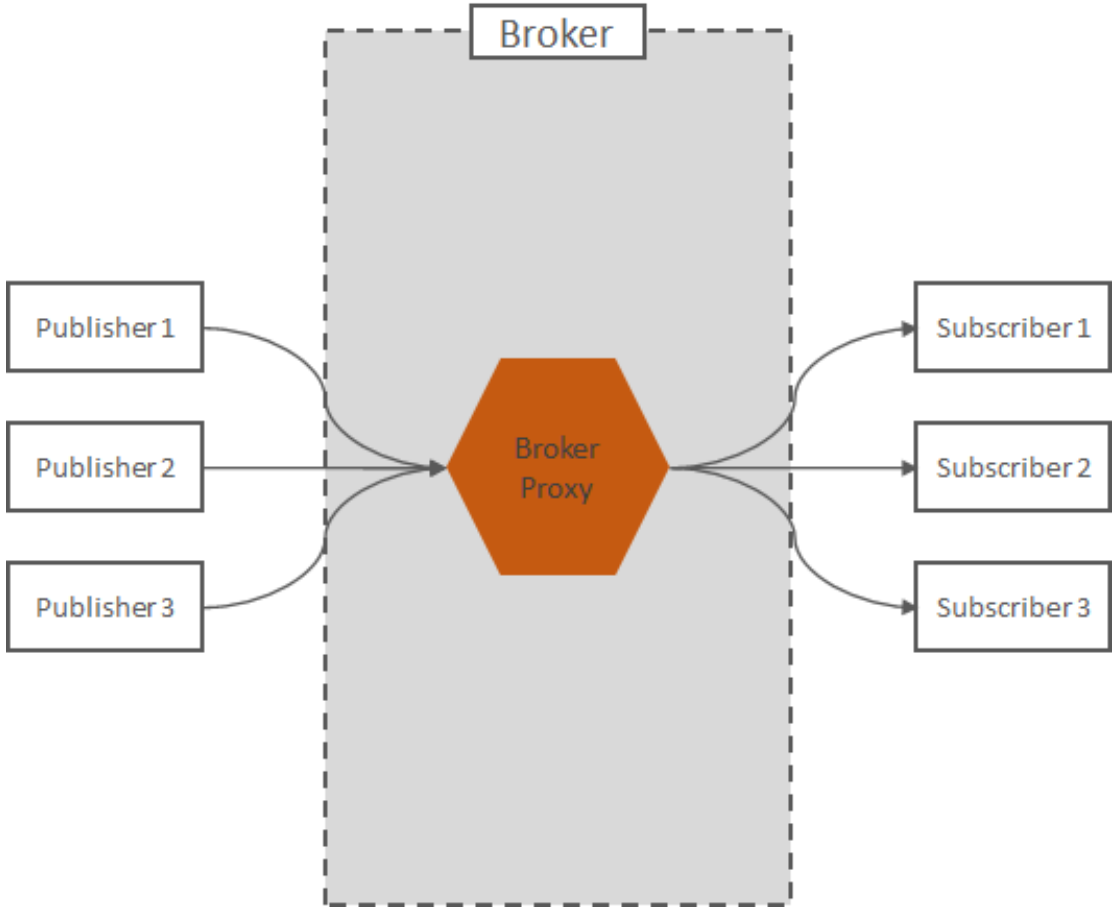
Broker-less

- A broker-less system, like ZeroMQ (also known as ØMQ, 0MQ, or zmq), sends messages directly from the publisher to the consumer.
- This requires each microservice to have the broker-less engine installed.
- It also requires each endpoint to know how to reach others.
- As you scale your microservices, it quickly becomes harder to manage the endpoint list.

Broker-less

- Because there is no central message system, it can have lower latency than a brokered system.
- This also causes a temporal coupling of the publisher to the consumer.
- This means that the consumer must be live and ready to handle the traffic.
- One way to handle the chance of a consumer not being available is to use a distributor.
- A distributor is a load balancer to share the load when you have multiple instances of a microservice.
- The distributor also handles when a consumer is unavailable, sending a message to another instance of your microservice

0MQ


Broker
Publisher 1
Publisher 2
Publisher 3
Broker
Proxy
Subscriber 1
Subscriber 2
Subscriber 3

Brokered

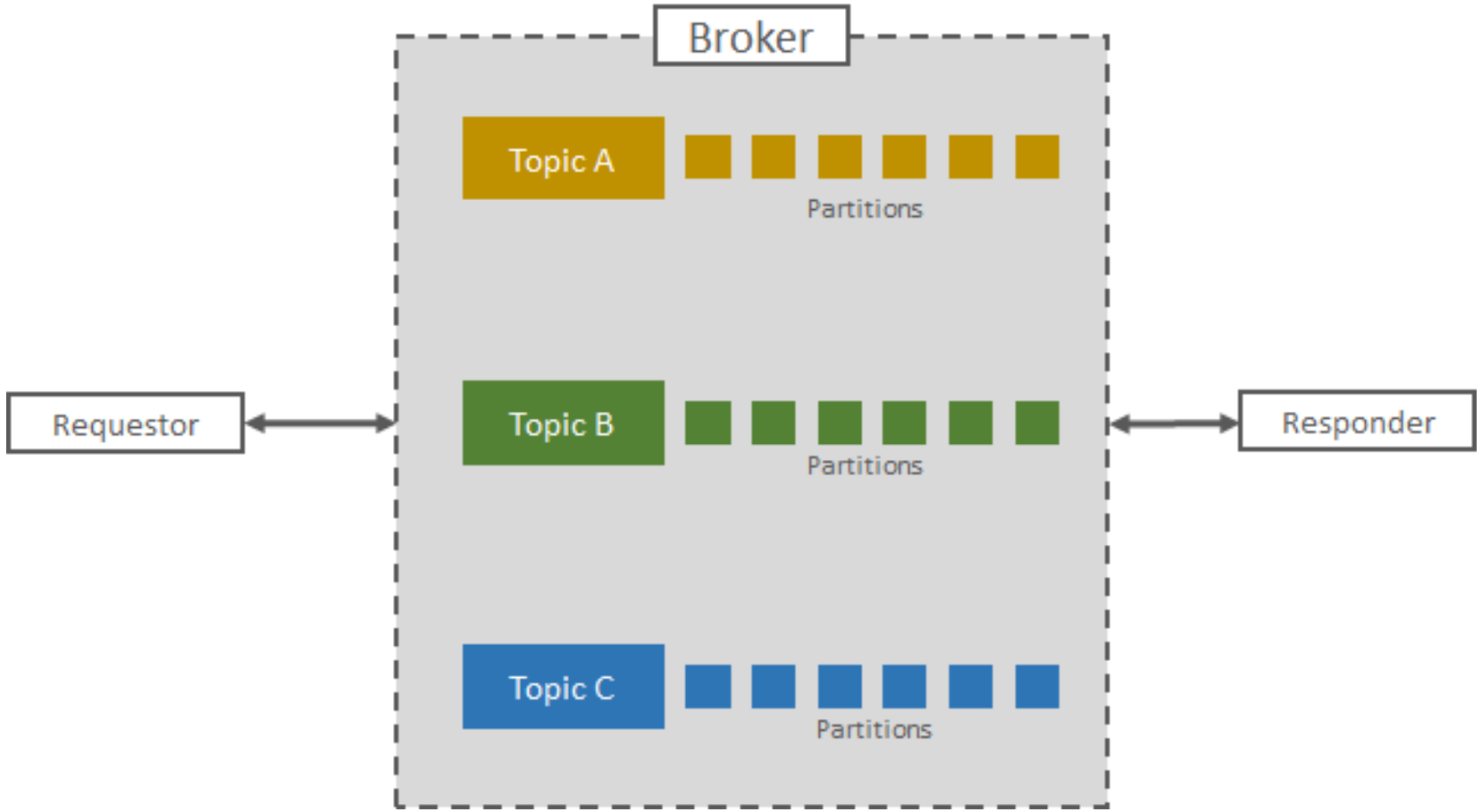
- A brokered system like ActiveMQ, Kafka, and RabbitMQ provides a centralized set of queues that hold messages until they are consumed
- Because the messages are stored and then sent to consumers, it provides a loosely coupled architecture.
- The storing of messages until they are consumed is not a high latency task.
- It simply means the publisher does not have to store the messages but can rely on the broker.

Brokered

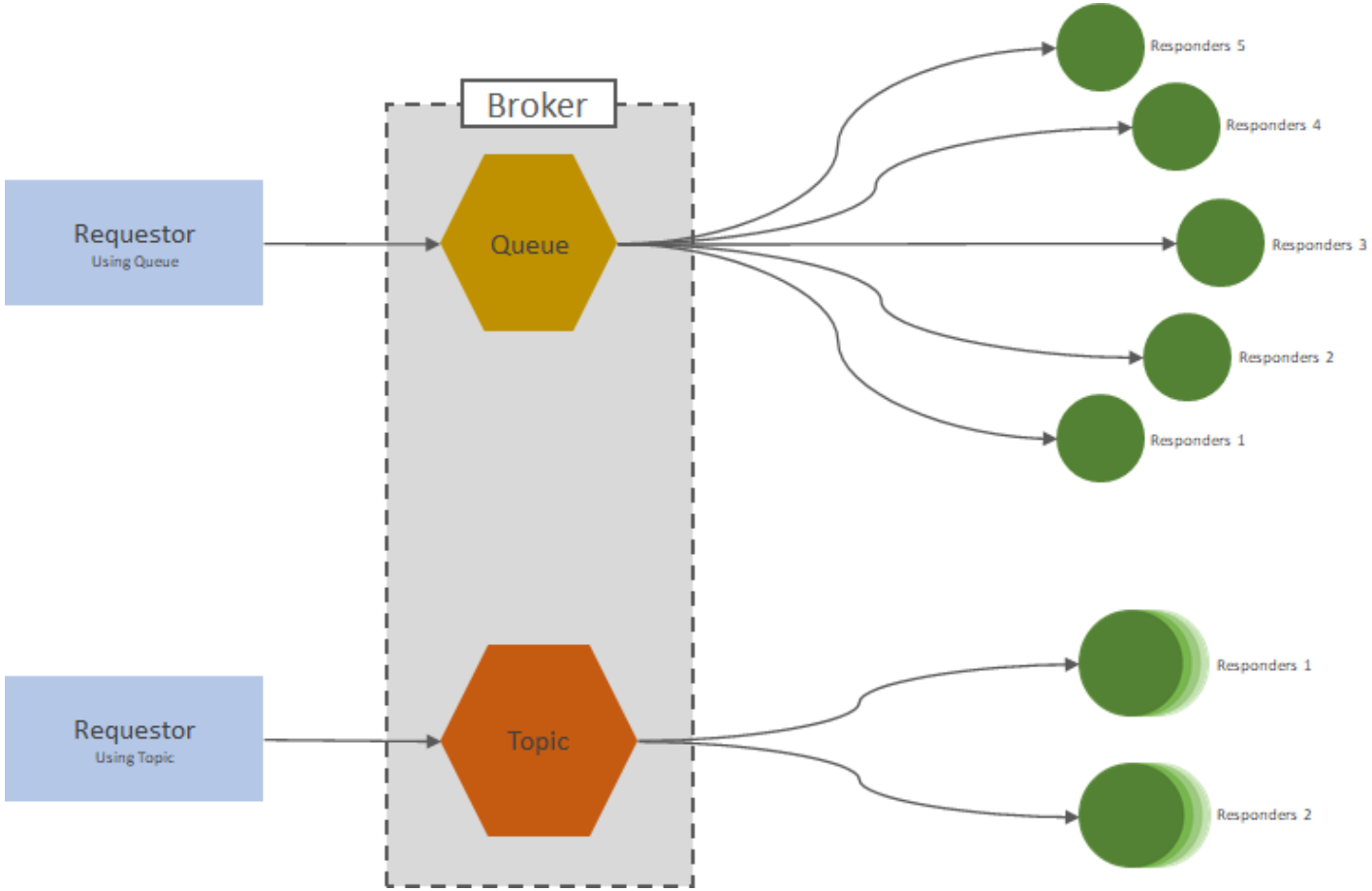
- Designing the use of a broker is about first understanding the business needs of each application and microservice.
- This topology will consist of various producers and consumers, each with their own messaging needs.
- Some producers only need to send messages and are not subscribing to other messages.
- Other producers will send messages for one business while also subscribing to other messages.
- As you will see in the code examples, our test client sends commands regarding invoices.
- Subscribes to the message that the invoice was created

	Kafka	Active MQ	Rapid MQ	Amazon SNS	Amazon SQS	Google Pub/Sub
Open Source	Yes	Yes	Yes	No	No	No
Written In	Scala	Java	Erlang	-	-	-
Languages Supported	Go, Haskell, PHP, Python, C#, Ruby, Node.js, OCaml	Java, C, C++, Python, Perl, Ruby, PHP, etc.	Java, Python, PHP, Ruby, JavaScript, C, C++, etc.	Java, Python, Ruby, PHP, Node.js, etc.	Java, Python, Ruby, PHP, Node.js, etc.	C++, C#, Go, Java, Node.js, PHP, Python, Ruby, etc.
Protocols	Binary protocol over TCP	AMQP, AUTO, MQTT, REST, OpenWire, etc.	AMQP, STOMP, MQTT, HTTP, and WebSockets	HTTP, HTTPS, SMTP, SMS, SQS, application, lambda and firehouse		HTTP, gRPC
Synchronous/ Asynchronous	Asynchronous	Supports both	Supports both	Asynchronous	Supports both	Asynchronous
Customers	LinkedIn, Spotify, Pinterest, Uber and others	Dopplr, FuseSource, RomTrac, STG and others	JP Morgan, NASA, Google, Adhar Project	Staples Inc., Bed Bath & Beyond Inc, Amazon, Stack, etc.	Pinterest, medium.com, Amazon, Stack, etc.	Client Platform, 9GAG, PLAID, Samba Tech, and others

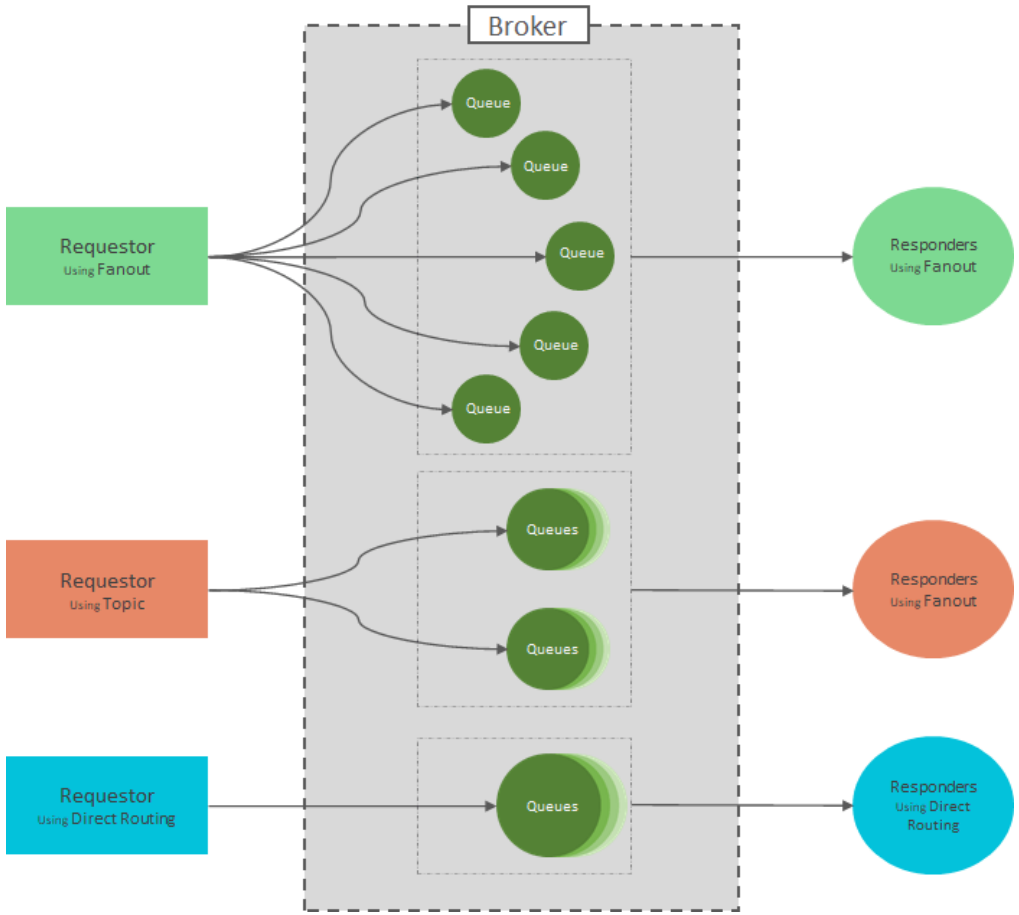
Kafka


Broker
Topic A
Partitions
Topic B
Partitions
Topic C
Partitions
Requestor
Responder

ActiveMQ


Broker
Requestor
Using Queue
Queue
Responders 5
Responders 4
Responders 3
Responders 2
Responders 1
Requestor
Using Topic
Topic
Responders 1
Responders 2

RabbitMQ


Broker
Queue
Queue
Queue
Queue
Queue
Requestor
Using Fanout
Responders
Using Fanout
Queues
Queues
Requestor
Using Topic
Responders
Using Fanout
Queues
Requestor
Using Direct Routing
Responders
Using Direct
Routing

Masstransit

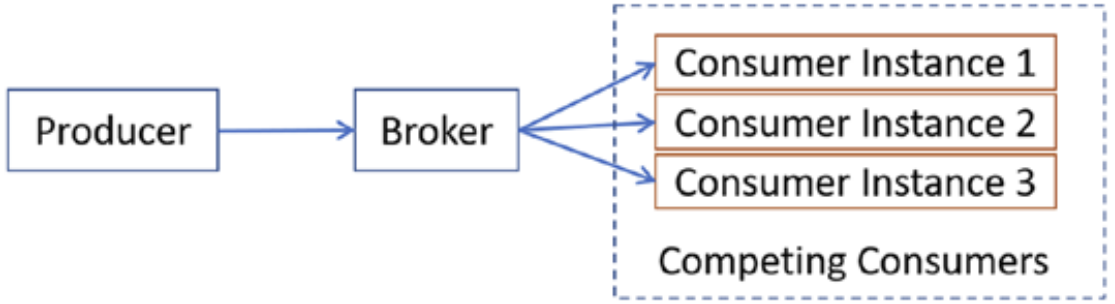
- An advantage of using a broker is that if a message fails in the processing by a consumer, the message stays in queue until it can pick it up again or another process is ready to process it.
- We will use the MassTransit library.
- It provides some abstractions that allow us to get running quickly.
- MassTransit can use RabbitMQ, among others, to provide the message transport we need.

Consumption Models

- There are multiple models for receiving messages you will see with multiple consumers.
- You will use one or both models.

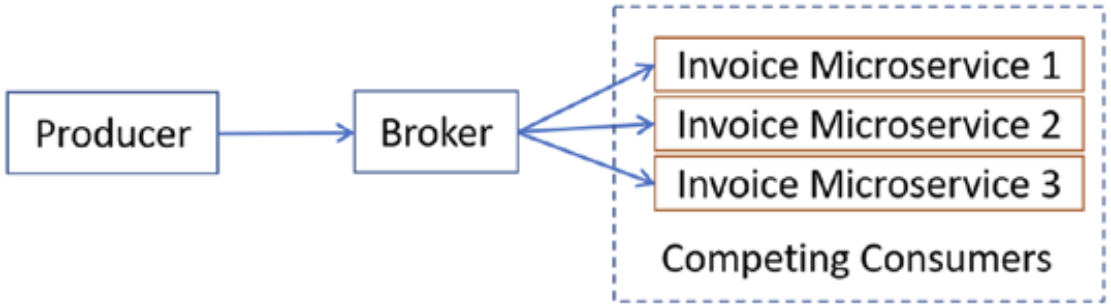
Competing Consumers

- Various business processes need to have a message processed by only one consumer.
- For example, a create invoice message should only be processed once.
- If multiple instances of a consumer processed it, duplicate information could be stored and sent out to other microservices.



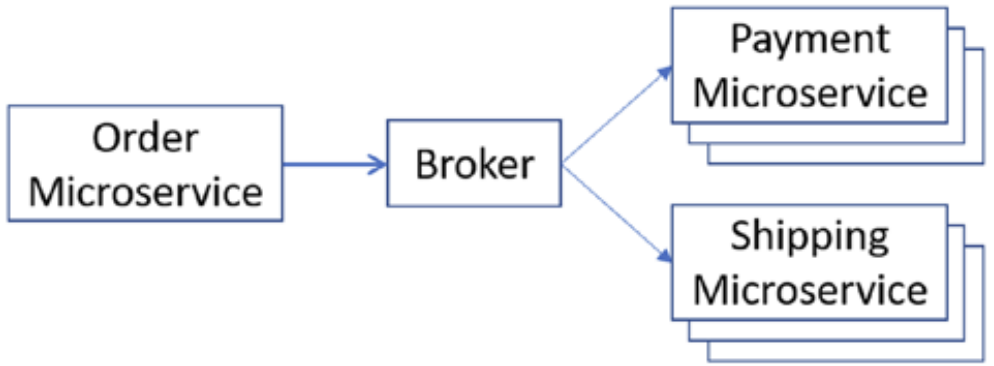
Competing Consumers

- An example of a competing consumer, is the Invoice Microservice.
- When you create multiple instances of the Invoice Microservice, they each become a competing consumer
- You should have multiple instances of a microservice for scaling, availability, and distribution of load.
- A microservice is designed to subscribe to a specific queue and message type.
- But when there are multiple instances of that microservice, you have a competing consumer scenario.
- When a message is sent, only one of the microservice instances receives the message.
- If that instance of the microservice that has the message fails to process it, the broker will attempt to send it to another instance for processing



Independent Consumers

- other microservices must also consume the message.
- these microservices do not compete with other consumers.
- receive a copy of the message no matter which competing consumer processes the message

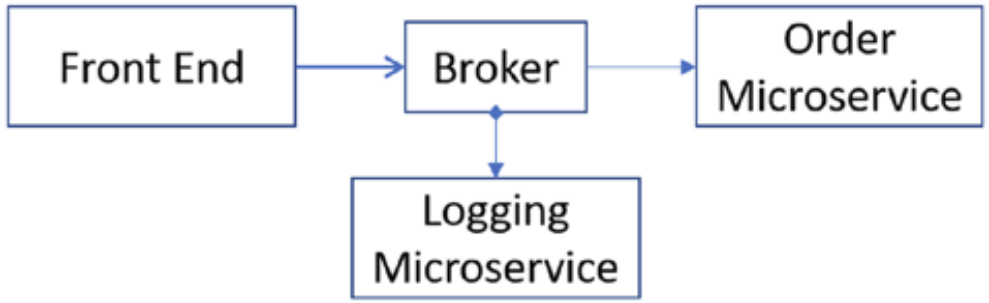


Independent Consumers

- These independent consumers process the messages for their specific business processing needs.
- a message is sent from the Order Microservice. The message needs to be received by one instance of the Payment Microservice and one instance of the Shipping Microservice.
- the Payment and Shipping microservices are not competing with each other. Instead, they are independent consumers.
- Also, note that each instance of a Payment Microservice is a competing consumer to itself. The same is true for the Shipping Microservice instances

Wire Tap pattern

- An example of this has a front-end component that sends a command to the Order Microservice to revise an order.
- In RabbitMQ, you can have an exchange that is bound to another exchange.
- This allows copies of messages to be sent to another consumer, even though the exchange is set up with direct topology



Delivery Guarantees

- As we evaluate our business needs and design a messaging topology solution, we must also consider that there will be times when messages
- either cannot be delivered or
- are delivered again!

- Message delivery disruption can allow for the possibility of causing duplication of data or other side effects

Delivery Guarantees

- When issues occur, there is a chance of duplicate messages being sent, so our consumers must handle the messages in an **idempotent** manner.
- **Treat every message with the possibility that it has already been processed before**

At Most Once

- This delivery guarantee is for when your system can tolerate the cases when messages fail to be processed.
- An example is when receiving temperature readings every second.
- If the broker delivers the message, but the consumer fails to process it, the message is not resent.
- The broker considers it delivered regardless of the consumer's ability to complete any processing of the message.
- “Fire-and-Forget.”

At Least Once (idempotent)

- The “at least once” delivery guarantee is where you will spend the most effort designing the consumers to handle messages in an idempotent manner

- If a message fails to be given to a consumer, it is retried.
- If a consumer receives a message but fails during the process, the broker will resend that message to another consumer instance.

At least once

- An example scenario: you have a consumer that creates a database entry for a new order, then that instance dies. It did not send an acknowledgment to the broker that it was done with the message.
- So, the broker sends that message to another consumer instance.
- There the message is processed and inadvertently creates a duplicate database entry.
- Although most broker systems have a flag that is set when it sends a message again, you are safer to assume the message is a duplicate anyway and process accordingly.

Further Reading

- There is also much to understand with distributed transactions which is outside the scope of this book.
- The Art of Immutable Architecture by Michael Perry, published by Apress.
- In Perry's book, there is a specific section on Idempotence and Commutativity applicable to handling messages

Once and Only Once

- Exactly Once
- Many argue it does not exist
- The point here is that guaranteeing a message was only delivered and processed once requires state management involving the consumers and the broker system.
- This addition of state management may be far more trouble than designing for the other delivery guarantees.

Message Ordering

- If the entire message system were single threaded, then there would be a guarantee of the order of messages
- tolerate the reception of out-of-order messages
- Example:

1. a message sent to an Order Microservice to create a new order
2. Another message to a different instance of the same microservice is sent with an update to the order
3. During the processing of the first message, an error occurs, and the message is not acknowledged.
4. The broker then sends that message to the next microservice instance that is available.
5. Meanwhile, the message with the update to the order is being processed.
6. The microservice code must decide how to update an order that does not exist yet.

Message Ordering

- One suggestion is for each instance of the microservice to share the same data store.
- Then messages received out of order could be stored temporarily and then reassembled when the order's creation message is processed
- We cannot force the order of the messages, but we can design for handling messages out of order

- Based on Elsevier Book .NET 6 Pro Microservices 2022