Asynchronous Communication

Wednesday, June 2, 2021 6:51 PM

Ref: https://docs.microsoft.com/en-us/azure/architecture/guide/technology-choices/messaging

Event Vs Message Communication

Messages can be classified into two main categories.

Message : If the producer expects an action from the consumer, that message is a command.

Event : If the message informs the consumer that an action has taken place, then the message is an event.

Events

Messages

Wednesday, June 2, 2021 7:11 PM

Commands
The producer sends a command with the intent that the consumer(s) will perform an operation within the scope of a business transaction.

A command is a high-value message and must be delivered at least once. If a command is lost, the entire business transaction might fail. Also, a command shouldn't be processed more than once. Doing so might cause an erroneous transaction. A customer might get duplicate orders or billed twice.

Commands are often used to manage the workflow of a multistep business transaction. Depending on the business logic, the producer may expect the consumer to acknowledge the message and report the results of the operation. Based on that result, the producer may choose an appropriate course of action.

Technology choices for a message broker

Azure Service Bus
Azure Event Grid
Azure Event Hubs

Service Bus

Wednesday, June 2, 2021 7:14 PM

Azure Service Bus queues are well suited for transferring commands from producers to consumers. Here are some considerations.

--Queues
-- Topic & subscription

Pull model
A consumer of a Service Bus queue constantly polls Service Bus to check if new messages are available. The client SDKs and Azure Functions trigger for Service Bus abstract that model. When a new message is available, the consumer's callback is invoked, and the message is sent to the consumer.

Guaranteed delivery
Service Bus allows a consumer to peek the queue and lock a message from other consumers.

It's the responsibility of the consumer to report the processing status of the message. Only when the consumer marks the message as consumed, Service Bus removes the message from the queue. If a failure, timeout, or crash occurs, Service Bus unlocks the message so that other consumers can retrieve it. This way messages aren't lost in transfer.

A producer might accidentally send the same message twice. For instance, a producer instance fails after sending a message. Another producer replaces the original instance and sends the message again. Azure Service Bus queues provide a built-in de-duping capability that detects and removes duplicate messages. There's still a chance that a message is delivered twice. For example, if a consumer fails while processing, the message is returned to the queue and is retrieved by the same or another consumer. The message processing logic in the consumer should be idempotent so that even if the work is repeated, the state of the system isn't changed.

Message ordering
If you want consumers to get the messages in the order they are sent, Service Bus queues guarantee first-in-first-out (FIFO) ordered delivery by using sessions. A session can have one or more messages. The messages are correlated with the SessionId property. Messages that are part of a session, never expire. A session can be locked to a consumer to prevent its messages from being handled by a different consumer.

For more information, see Message Sessions.

Message persistence
Service bus queues support temporal decoupling. Even when a consumer isn't available or unable to process the message, it remains in the queue.

Checkpoint long-running transactions
Business transactions can run for a long time. Each operation in the transaction can have multiple messages. Use checkpointing to coordinate the workflow and provide resiliency in case a transaction fails.

Service Bus queues allow checkpointing through the session state capability. State information is incrementally recorded in the queue (SetState) for messages that belong to a session. For example, a consumer can track progress by checking the state (GetState) every now and then. If a consumer fails, another consumer can use state information to determine the last known checkpoint to resume the session.

Dead-letter queue (DLQ)
A Service Bus queue has a default subqueue, called the dead-letter queue (DLQ) to hold messages that **couldn't be delivered or processed**. Service Bus or the message processing logic in the consumer can add messages to the DLQ. The DLQ keeps the messages until they are retrieved from the queue.

Here are examples when a message can end up being in the DLQ:

A poison message is a message that cannot be handled because it's malformed or contains unexpected information. In Service Bus queues, you can detect poison messages by setting the MaxDeliveryCount property of the queue. If number of times the same message is received exceeds that property value, Service Bus moves the message to the DLQ.

A message might no longer be relevant if it isn't processed within a period. Service Bus queues allow the producer to post messages with a time-to-live attribute. If this period expires before the message is received, the message is placed in the DLQ.

Examine messages in the DLQ to determine the reason for failure.

Hybrid solution
Service Bus bridges on-premises systems and cloud solutions. On-premises systems are often difficult to reach because of firewall restrictions. Both the producer and consumer (either can be on -premises or the cloud) can use the Service Bus queue endpoint as the pickup and drop off location for messages.

Topics and subscriptions
Service Bus supports the Publisher-Subscriber pattern through Service Bus topics and subscriptions.

This feature provides a way for the producer to broadcast messages to multiple consumers. When a topic receives a message, it's forwarded to all the subscribed consumers. Optionally, a subscription can have filter criteria that allows the consumer to get a subset of messages. Each consumer retrieves messages from a subscription in a similar way to a queue.

For more information, see Azure Service Bus topics.

What is Azure Service Bus?

Tuesday, July 6, 2021 2:50 PM

https://docs.microsoft.com/en-us/azure/service-bus-messaging/service-bus-messaging-overview#topics

Microsoft Azure Service Bus is a fully managed enterprise message broker with message queues and publish-subscribe topics. Service Bus is used to decouple applications and services from each other, providing the following benefits:

Load-balancing work across competing workers
Safely routing and transferring data and control across service and application boundaries
Coordinating transactional work that requires a high-degree of reliability

Overview

Data is transferred between different applications and services using messages. A message is a container decorated with metadata, and contains data. The data can be any kind of information, including structured data encoded with the common formats such as the following ones: JSON, XML, Apache Avro, Plain Text.
Some common messaging scenarios are:

- Messaging. Transfer business data, such as sales or purchase orders, journals, or inventory movements.
- Decouple applications. Improve reliability and scalability of applications and services. Producer and consumer don't have to be online or readily available at the same time. The load is leveled such that traffic spikes don't overtax a service.
- Load Balancing. Allow for multiple competing consumers to read from a queue at the same time, each safely obtaining exclusive ownership to specific messages.
- Topics and subscriptions. Enable 1:*n* relationships between publishers and subscribers, allowing subscribers to select particular messages from a published message stream.
- Transactions. Allows you to do several operations, all in the scope of an atomic transaction. For example, the following operations can be done in the scope of a transaction.
    1. Obtain a message from one queue.
    2. Post results of processing to one or more different queues.
    3. Move the input message from the original queue.The results become visible to downstream consumers only upon success, including the successful settlement of input message, allowing for once-only processing semantics. This transaction model is a robust foundation for the

compensating transactions pattern in the greater solution context.

- Message sessions. Implement high-scale coordination of workflows and multiplexed transfers that require strict message ordering or message deferral.

If you're familiar with other message brokers like Apache ActiveMQ, Service Bus concepts are similar to what you know. As Service Bus is a platform-as-a-service (PaaS) offering, a key difference is that you don't need to worry about the following actions. Azure takes care of those chores for you.

- **Worrying about hardware failures**
- **Keeping the operating systems or the products patched**
- **Placing logs and managing disk space**
- **Handling backups**
- **Failing over to a reserve machine**

From <https://docs.microsoft.com/en-us/azure/service-bus-messaging/service-bus-messaging-overview#topics>

https://docs.microsoft.com/en-us/azure/service-bus-messaging/service-bus-queues-topics-subscriptions

Azure Queue VS Azure Service Bus Queues

Tuesday, July 6, 2021 3:06 PM

Consider using Storage queues

As a solution architect/developer, you should consider using Storage queues when:

- Your application must store over 80 gigabytes of messages in a queue.
- Your application wants to track progress for processing a message in the queue. It's useful if the worker processing a message crashes. Another worker can then use that information to continue from where the prior worker left off.
- You require server side logs of all of the transactions executed against your queues.

Consider using Service Bus queues

As a solution architect/developer, you should consider using Service Bus queues when:

- Your solution needs to receive messages without having to poll the queue. With Service Bus, you can achieve it by using a long-polling receive operation using the TCP-based protocols that Service Bus supports.
- Your solution requires the queue to provide a guaranteed first-in-first-out (FIFO) ordered delivery.
- Your solution needs to support automatic duplicate detection.
- You want your application to process messages as parallel long-running streams (messages are associated with a stream using the session ID property on the message). In this model, each node in the consuming application competes for streams, as opposed to messages. When a stream is given to a consuming node, the node can examine the state of the application stream state using transactions.
- Your solution requires transactional behavior and atomicity when sending or receiving multiple messages from a queue.
- Your application handles messages that can exceed 64 KB but won't likely approach the 256-KB limit.
- You deal with a requirement to provide a role-based access model to the queues, and different rights/permissions for senders and receivers. For more information, see the following articles:
- Authenticate with managed identities
- Authenticate from an application
- Your queue size won't grow larger than 80 GB.
- You want to use the AMQP 1.0 standards-based messaging protocol. For more information about AMQP, see Service Bus AMQP Overview.
- You envision an eventual migration from queue-based point-to-point communication to a publish-subscribe messaging pattern. This pattern enables integration of additional receivers (subscribers). Each receiver receives

independent copies of either some or all messages sent to the queue.
- Your messaging solution needs to support the "At-Most-Once" delivery guarantee without the need for you to build the additional infrastructure components.
- Your solution needs to publish and consume batches of messages.

From <https://docs.microsoft.com/en-us/azure/service-bus-messaging/service-bus-azure-and-service-bus-queues-compared-contrasted>

https://docs.microsoft.com/en-us/azure/service-bus-messaging/service-bus-azure-and-service-bus-queues-compared-contrasted

Events

Wednesday, June 2, 2021 7:12 PM

An event is a type of message that a producer raises to announce facts.

The producer (known as the publisher in this context) has no expectations that the events will result in any action.

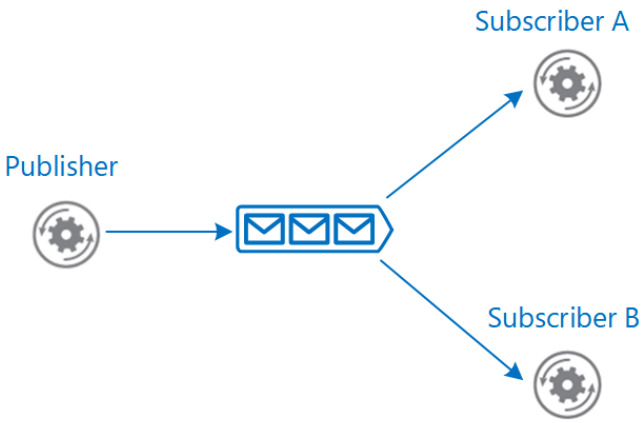
Interested consumer(s), can subscribe, listen for events, and take actions depending on their consumption scenario. Events can have multiple subscribers or no subscribers at all. Two different subscribers can react to an event with different actions and not be aware of one another.

The producer and consumer are loosely coupled and managed independently. The consumer isn't expected to acknowledge the event back to the producer. A consumer that is no longer interested in the events, can unsubscribe. The consumer is removed from the pipeline without affecting the producer or the overall functionality of the system.

There are two categories of events:

The producer raises events to announce discrete facts. A common use case is event notification. For example, Azure Resource Manager raises events when it creates, modifies, or deletes resources. A subscriber of those events could be a Logic App that sends alert emails.

The producer raises related events in a sequence, or a stream of events, over a period of time. Typically, a stream is consumed for statistical evaluation. The evaluation can be done within a temporal window or as events arrive. Telemetry is a common use case, for example, health and load monitoring of a system. Another case is event streaming from IoT devices.



Event-driven architecture style

Tuesday, July 6, 2021 3:47 PM

An event-driven architecture consists of event producers that generate a stream of events, and event consumers that listen for the events.
Events are delivered in near real time, so consumers can respond immediately to events as they occur. Producers are decoupled from consumers — a producer doesn't know which consumers are listening. Consumers are also decoupled from each other, and every consumer sees all of the events. This differs from a Competing Consumers pattern, where consumers pull messages from a queue and a message is processed just once (assuming no errors). In some systems, such as IoT, events must be ingested at very high volumes.
An event driven architecture can use a pub/sub model or an event stream model.

- Pub/sub: The messaging infrastructure keeps track of subscriptions. When an event is published, it sends the event to each subscriber. After an event is received, it cannot be replayed, and new subscribers do not see the event.
- Event streaming: Events are written to a log. Events are strictly ordered (within a partition) and durable. Clients don't subscribe to the stream, instead a client can read from any part of the stream. The client is responsible for advancing its position in the stream. That means a client can join at any time, and can replay events.

On the consumer side, there are some common variations:

- Simple event processing. An event immediately triggers an action in the consumer. For example, you could use Azure Functions with a Service Bus trigger, so that a function executes whenever a message is published to a Service Bus topic.
- Complex event processing. A consumer processes a series of events, looking for patterns in the event data, using a technology such as Azure Stream Analytics or Apache Storm. For example, you could aggregate readings from an embedded device over a time window, and generate a notification if the moving average crosses a certain threshold.
- Event stream processing. Use a data streaming platform, such as Azure IoT Hub or Apache Kafka, as a pipeline to ingest events and feed them to stream processors. The stream processors act to process or transform the stream. There may be multiple stream processors for different subsystems of the application. This approach is a good fit for IoT workloads.

The source of the events may be external to the system, such as physical devices in an IoT solution. In that case, the system must be able to ingest the data at the volume and throughput that is required by the data source.
In the logical diagram above, each type of consumer is shown as a single box. In practice, it's common to have multiple instances of a consumer, to avoid having the consumer become a single point of failure in system. Multiple instances might also

be necessary to handle the volume and frequency of events. Also, a single consumer might process events on multiple threads. This can create challenges if events must be processed in order or require exactly-once semantics. See Minimize Coordination.

When to use this architecture

- Multiple subsystems must process the same events.
- Real-time processing with minimum time lag.
- Complex event processing, such as pattern matching or aggregation over time windows.
- High volume and high velocity of data, such as IoT.

Benefits

- Producers and consumers are decoupled.
- No point-to-point integrations. It's easy to add new consumers to the system.
- Consumers can respond to events immediately as they arrive.
- Highly scalable and distributed.
- Subsystems have independent views of the event stream.

Challenges

- Guaranteed delivery. In some systems, especially in IoT scenarios, it's crucial to guarantee that events are delivered.
- Processing events in order or exactly once. Each consumer type typically runs in multiple instances, for resiliency and scalability. This can create a challenge if the events must be processed in order (within a consumer type), or if the processing logic is not idempotent.

Additional considerations

- The amount of data to include in an event can be a significant consideration that affects both performance and cost. Putting all the relevant information needed for processing in the event itself can simplify the processing code and save additional lookups. Putting the minimal amount of information in an event, like just a couple of identifiers, will reduce transport time and cost, but requires the processing code to look up any additional information it needs. For more information on this, take a look at this blog post.

From <https://docs.microsoft.com/en-us/azure/architecture/guide/architecture-styles/event-driven>

Azure Event Grid

Wednesday, June 2, 2021 9:14 PM

Azure Event Hubs

Wednesday, June 2, 2021 9:16 PM