The Event Mesh: A Primer

RTInsights.com

Red Hat

Copyright © 2020 RTInsights. All rights reserved. All other trademarks are the property of their respective companies. The information contained in this publication has been obtained from sources believed to be reliable. NACG LLC and RTInsights disclaim all warranties as to the accuracy, completeness, or adequacy of such information and shall have no liability for errors, omissions or inadequacies in such information. The information expressed herein is subject to change without notice.

Chapter 1: What Are Events?

An event is a change, action, or observation in a system that produces a notification. Events are triggered by changes such as master data entity updates like a customer billing address change, or an action like a new order or a trade approval. Other triggers are more basic, like a battery charge level threshold being met or a periodic temperature reading from a sensor. It is common to refer to the notification as the "event." Rather than use the term trigger, software architects often say "send an event" when a specified condition occurs.

Event notifications produced programmatically by the system of record (SOR), can contain a snapshot of the affected entity, a delta (i.e., changed attributes only), or a reference (e.g., a PO number). Snapshot and delta events usually carry business meaning, have a complex structure, and are commonly represented in JSON or XML. When a reference is contained in the event notification, a call back to the SOR is required to get a more meaningful data set. For example, if an event contains just the PO number, then a call back to the Order Management Application is required to request more information about the PO entity.

In an event-based system, a component produces an event without necessarily expecting or controlling the consumption of that event. This decoupling of event production and consumption enables a developer to create an event without needing to know the specifics of the environment in which it will be consumed—greatly simplifying the process.

The database (DB) change log also can be a source of events. The advantage of the DB change log is that it is maintained by the DBMS automatically and does not require the application developer to make modifications to produce the event notification.

Regardless of the type of event, time ordering is an important characteristic. Time ordering, also referred to as sequence preservation, ensures that the stream of events retains the integrity of the business state. For example, a snapshot event represents the state of an entity at a point in time. If processed out of sequence, an invalid representation of the entity would result. An event stream is a time-ordered set of events that can't be modified once produced. Such a property is described as immutable.

In a system based on events, a component produces an event without necessarily expecting or controlling the immediate consumption of that event. There is a loose coupling between the event producer and consumer. From a development standpoint, the decoupling of event production and consumption greatly simplifies matters. A developer can create an event without needing to know the specifics of the environment in which it will be consumed.

That contrasts with a system relying on service invocation using a request-reply interaction pattern. The roles in service-oriented architectures (SOA) and more modern REST architecture style rely on synchronous tight coupling between the client and the server. The tight coupling of components leads to some limitations that can be overcome in an event-driven architecture.

Copyright (©) 2020 RTInsights

Event-Enabling Technologies

Over the past 20 years, messaging systems have implemented the core capabilities for point-to-point and pub/sub communication. Recently, cloud computing has surfaced limits of the current technologies in areas such as interoperability and scalability. A number of open source projects and standards have gone beyond the core features and enabled large-scale event-driven application development, including Apache Kafka, Advanced Message Queueing Protocol, and Knative.

1. Apache Kafka is a publish/subscribe messaging system that uses a distributed commit log as the durable record of all messages. Multiple producers can add messages to a topic, and multiple consumers can read messages managing their position in the sequence. Other features include the ability to replay messages, high throughput, and horizontal scalability. Apache Kafka is often deployed when there is a high volume of messages with low latency requirements.

2. Advanced Message Queuing Protocol (AMQP) is a standard messaging protocol with advanced capabilities such as flow-control, message delivery guarantees, and security. As a wire-level protocol, it enables the integration between different messaging platforms and products. AMQP is a good choice for the protocol of the event mesh in a heterogeneous cloud/on-premise environment. The Apache Software foundation's AMQP implementation—Qpid provides a number of features like queuing, transaction, management, and clustering and APIs for C++ and Java (JMS).

3. Knative is a Kubernetes-based platform for serverless computing. Developers can focus on the development of specific functions, while the platform is responsible for allocating computing resources based on the requests. Using Knative means your serverless applications can be deployed and run on any Kubernetes platform, preventing vendor lock-in.

Knative Eventing is a component of Knative that enables event-driven applications on the platform that react to real-time information via event notifications. It is consistent with the CloudEvents specification, which ensures interoperability with various messaging protocols, including Apache Kafka, MQTT, AMQP. Knative Eventing enables cloud-native, cloud-agnostic event-driven development.

These three technologies are emerging as keys to event-driven architecture for microservices.

Copyright (©) 2020 RTInsights

Chapter 2: The Event-Driven Enterprise

Enterprises are compelled to improve the speed and responsiveness of their internal and customer-facing processes to keep pace with dynamic competitors. In almost all industries, there is an increasing volume of event producers and consumers in their IT ecosystems. Making use of the volume of events and making use of the information in event streams requires an event-driven architecture (EDA).

Event-driven architecture (EDA) is a software architecture paradigm promoting the production, detection, consumption of, and reaction to events.[1] But EDA entails more than just the production and consumption of events. It involves the planning for how events are interpreted, transformed, published to subscribers, propagated across distributed networks, and persisted. These design patterns illustrate the scope of an EDA.

Key design patterns and capabilities of an EDA include:

- Eventual consistency between the system of record and systems that maintain copies. Rather than two-phase commit or distributed transactions, events are used to allow systems to become consistent after some delay. The delay can be very short (e.g., within seconds) or much longer e.g., within hours) as long as the business requirements are met.

- Pub-sub and message-oriented middleware to provide transport to reliably move event messages from point-to-point or publisher-to-subscriber.

- Protocol transformation and routing using enterprise integration patterns. EDA systems typically use brokers and a service bus to provide a hub-and-spoke topology and capabilities to bridge different middleware and message formats.

- Streaming analytics, which is a type of event stream processing that applies machine learning or business rules against a time delimited window of events to make decisions or take actions.

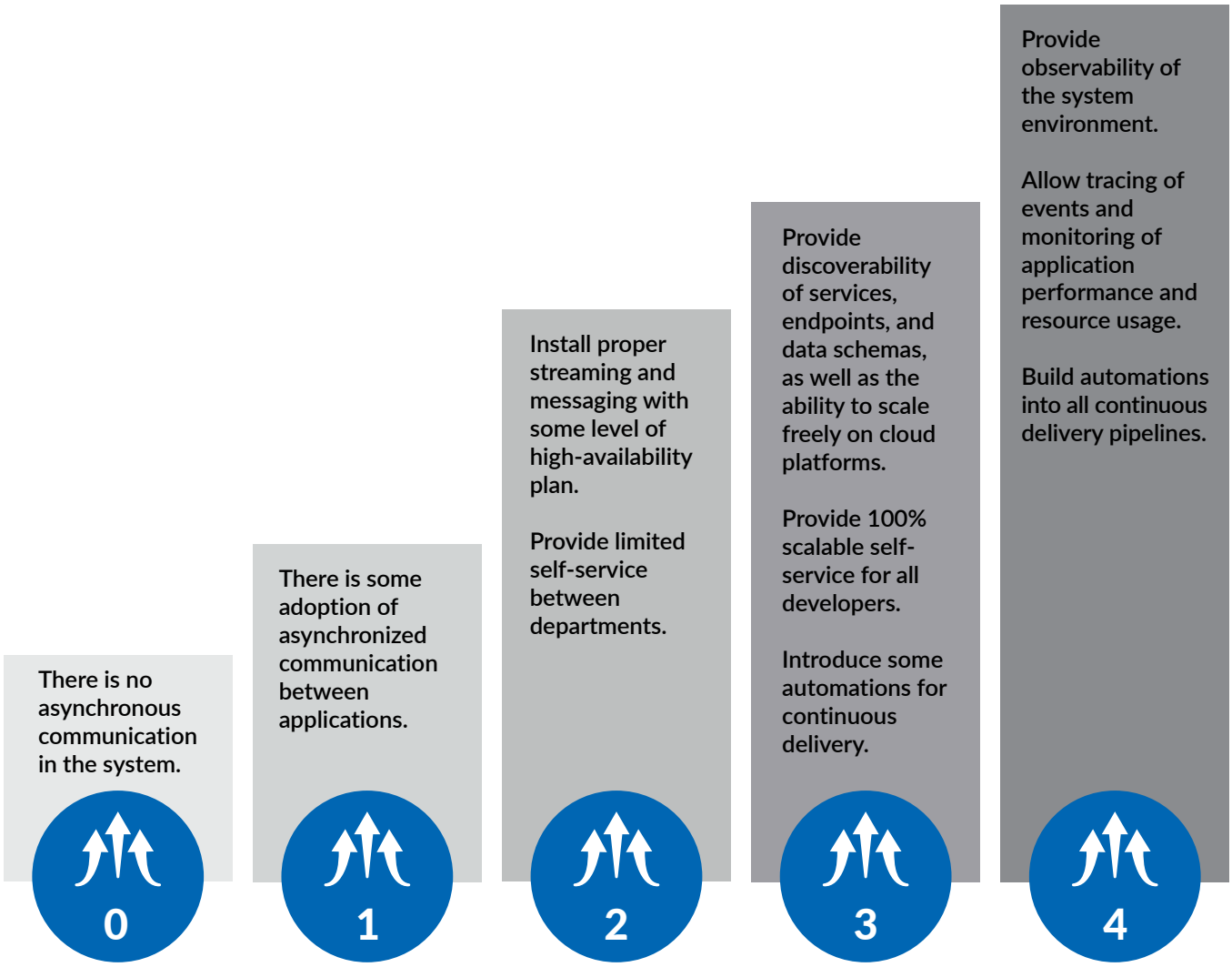
- Event stream as the first-class system of record, where in a microservices domain, the event stream is an authoritative source of data. This pattern is referred to as Event Sourcing.

- Idempotent message processing, which ensures that systems are designed so that the same event can be processed multiple times without changing the result beyond the initial processing.

- Event mesh, which is a dynamic infrastructure that allows messages to be delivered across a distributed enterprise.

As EDA concepts become more broadly adopted, enterprises progress through increasing levels of maturity.

[1] Source: Wikipedia, "Event-Driven Architecture"



Event-driven architecture maturity level

At Level 0, there is no asynchronous communication in the enterprise. All integration is synchronous, for example, Synchronous request-reply Web services called from Web Browser to back-end applications.

At Level 1, there is some adoption of async communication between Applications. For example, point-to-point message exchange between related systems using one messaging platform like IBM MQ, JMS, or Apache Kafka.

At Level 2, streaming and messaging are installed with some level of high-availability and limited self-service. The messaging infrastructure has the capability to handle failures in the infrastructure. It can provide guarantees about the delivery of the messages (e.g., at-least-once, at-most-once) and can participate in distributed transactions. More applications, based on diverse technology stacks in other parts of the enterprise, can connect to the event streams to take advantage of the available information.

Copyright (©) 2020 RTInsights

At Level 3, discoverability and scalability are enabled. Applications register the end-points and the data schema in a directory that can be accessed by other applications. The applications producing events become less aware of all the clients connected, and the same events can be used for new purposes such as logging and business monitoring. The messaging infrastructure can handle higher and variable load using auto-scaling and allows event producers and consumers to be unaware of the physical network topology. Some automation is implemented for the release process, referred to as Continuous Delivery, so the event-driven applications can be deployed more easily.

Finally, at Level 4, observability of the entire system is enabled, and fully automated processes for releases provide Continuous Delivery capabilities. The events are pervasive in the organization with new sources and destinations added routinely. The focus changes to maintaining a robust messaging infrastructure that can scale quickly as the volume of messages increases. Enterprise-wide observability is available so that administrators can monitor resources utilization real-time and trace messages end-to-end across multiple nodes in the event mesh. Full automation is implemented for enabling Continuous Delivery so the event-driven applications can be deployed at a click of the button.

Coinciding with the move to an EDA, another fundamental architectural shift is taking place. Namely, there is an explosion in the use of serverless technology, which lets developers focus on the code without concern for managing and provisioning the back-end infrastructure.

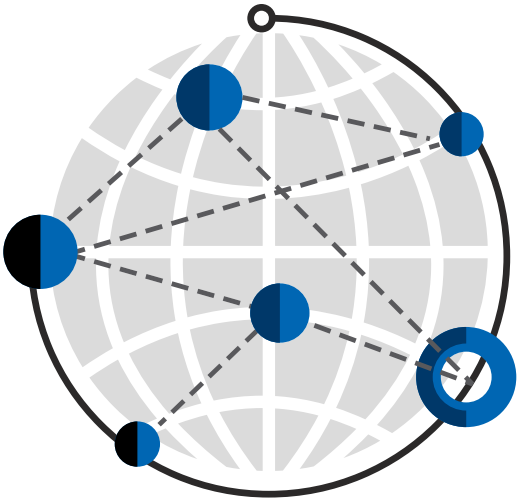
Given these capabilities, organizations are embracing EDA to support a variety of business initiatives. Coinciding with the move to an EDA, another fundamental architectural shift is taking place. Namely, there is an explosion in the use of serverless technology, which lets developers focus on the code without concern for managing and provisioning the back-end infrastructure.

Serverless introduces new challenges to EDA. Things like Java events, event handling loops, and older EDA structures were not designed to work in today's application environment. These older technologies were suitable in traditional client-server and data center deployment scenarios. They do not easily scale to modern application scenarios that are based on loosely coupled elements working together over a distributed cloud architecture.

Technology such as Knative eventing—which addresses a common need for cloud-native development and provides composable primitives to enable late-binding event sources and event consumers—helps serverless and EDA complement one another.

Copyright (©) 2020 RTInsights

Chapter 3: What Is an Event Mesh?



Large enterprises have widely distributed network and application topologies. They need to reliably deliver a high volume of events across global networks, traversing hybrid clouds and on-prem applications while avoiding bottlenecks. The event mesh is a key enabler for the event-driven enterprise.

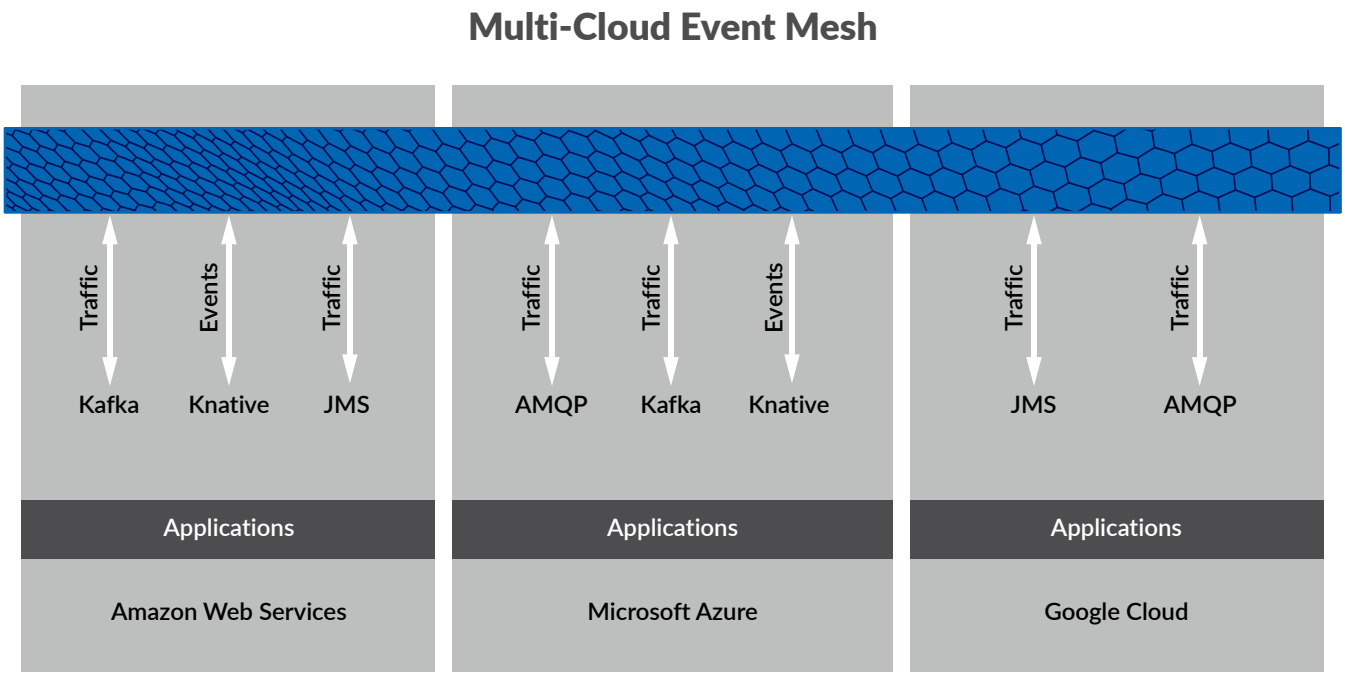
An event mesh is a dynamic infrastructure that propagates events across disparate cloud platforms and performs protocol translation. Critical capabilities include:

- Support for ingress and egress of events in various transports, such as Kafka, Knative, HTTP, AMQP, and others
- Fault tolerance for high-reliability delivery of messages, including automated recovery from network failures and fallback destinations for undeliverable messages
- Support for multi-protocol bridges between disparate events, applications, and messaging platforms
- Support for on-premises and multi-cloud deployment to provide a uniform infrastructure across the enterprise
- Multiple client APIs for a wide range of programming languages and environments
- Support for Multicast (all subscribers receive a copy of each message) or anycast (one subscriber receives a copy of each produced message)
- Management console for the administration of the mesh and monitoring of activity
- Secure transmission of event messages

These capabilities are needed to support modern application development. Microservices and cloud architectures often leverage a service mesh, which is a networking infrastructure that outboards the network logic, allowing the microservice to focus on business logic. A service mesh supports synchronous request-reply processing. Similarly, an event mesh supports application developers by alleviating concerns about the location of consumers across local, regional, and global distributed topologies to support loosely coupled event-driven use cases.

Copyright (©) 2020 RTInsights

The figure below illustrates the elements of an event mesh. Events can flow bidirectionally across the multi-cloud topology. As more applications produce and consume events, a key feature of the event mesh is that events published by any application in any programming environment (e.g. a Java JMS e event) in one cloud can be consumed by an application on another cloud using a different API (e.g., a Kafka event subscription in a Python application). This frees application developers from the complexity of designing and managing complex event distribution networks and lets application development teams choose the development environment of their choice without any constraints of the messaging platform. And serverless application deployment model reduces the infrastructure requirements for the disparate application environments.



Chapter 4: Deploying the Event Mesh in the Real World

An event-driven architecture that leverages the event mesh supports a wide range of use cases that encompass complex multi-cloud, widely distributed topologies using diverse application stacks.

The examples in this section show how the event mesh is leveraged in real-world scenarios.

Car Rental Recommendation System

A car rental company has a reservation and loyalty application with a large user base, accessible through web and mobile apps, with many pages and dynamic content. The apps generate a large volume of interaction events that need to be processed in real-time to create a personalized experience for the user and to update the customer behavior model. The design for this scenario is for the user interface (UI) components to push events via the event mesh pub-sub component, such as Apache Kafka.

A big-data processor, like Apache Spark, will subscribe to these streams for analytics. At the same time, stream analytics analyze the real-time events via Kafka topics and generate new events that are sent back to the UI as dynamic content or to other applications, like fraud detection. Such a design has been proven to be highly scalable, handling a large number of events per day while providing real-time recommendations and personalization.

Multinational Financial Organization

A financial organization with branches and offices in multiple countries and trading hub cities needs to keep latency and response time low for the synchronization of real-time trading data. Distributed apps connect to the edges of the event mesh and send messages to data streams identified by virtual names that represent the desired destinations. Routers, like Apache Qpid, ensure that messages go to the proper destination using naming services and routing rules. Messages are routed without clients being aware of the physical topology of the network and messaging system.

This use case highlights a multi-protocol, standards-based messaging system. A telecommunications company has a customer service mainframe application built using CICS/COBOL in the 1980s. It sends repair orders to a Java dispatch app running on a Linux server and finally sends notifications to a mobile application. To support this use case, the mainframe communicates with the messaging system through an AMQP channel between IBM MQ and the message broker. The Java application uses the JMS standard API to receive/send messages. Finally, the communication with the mobile application is done using a lightweight AMQP API supported by Apache Qpid.

Interoperable Train Control Messaging

The Federal Railroad Administration funded the development of a messaging solution that allows applications to exchange messages regardless of their physical locations and type of connectivity. The apps included onboard systems, office systems, and wayside signals. High availability of all message exchange was required to ensure the safe operations of the trains. Trains travel through high and low bandwidth networks and through tunnels with no network connectivity. All messages need to have guaranteed delivery, and the entire network is highly available and fault-tolerant. A multi-transport high availability clustered Qpid AMQP implementation was selected to support the exchange of messages.

Chapter 5: Conclusions and More Resources

Advanced enterprises consider an EDA of equal importance and complementary to a service-oriented architecture. The reason: In many modern enterprises, there are more event producers and consumers in their IT ecosystems, and the volume of events is growing. Harnessing these events enables enterprises to be more agile in their internal business processes and their customer interactions.

Enterprises typically follow an EDA journey that can be viewed in terms of a maturity model. Most start with the integration of a few related applications, leveraging a single messaging system. Over time, more mature enterprises implement an event mesh as an enabler for the loosely coupled integration of legacy systems and modern microservice-based applications across widely distributed topologies.

An event mesh has a key set of features that supports such multiple interaction patterns and reduces complexity for developers of modern applications that implement near real-time loosely coupled design patterns. Serverless deployments, microservices and the event mesh support an environment where computing resources scale quickly and automatically. But perhaps the most important benefit is that application developers can focus on implementing the business logic using the best tools available for the job. The use of an EDA and event mesh offers a simplicity with regard to application development. Event developers need only be concerned about their environment. Once the event is published, any consumer can make use of the event regardless of the development platform or streaming technology they're using, or the cloud the event is hosted on.

Red Hat enables a dynamic event mesh with open-source-based products in the AMQ family. For more information, follow these links.

Article: What is Apache Kafka?

Analyst Report: Event-driven applications with Red Hat AMQ Streams

Datasheet: Red Hat AMQ: Simplify Apache Kafka on Red Hat OpenShift

Detail: Event-driven architecture for a hybrid cloud blueprint

Datasheet: Red Hat Integration: Cloud-native connectivity for apps and systems

RTInsights is an independent, expert-driven web resource for senior business and IT enterprise professionals in vertical industries. We help our readers understand how they can transform their businesses to higher-value outcomes and new business models with AI, real-time analytics, and IoT. We provide clarity and direction amid the often confusing array of approaches and vendor solutions. We provide our partners with a unique combination of services and deep domain expertise to improve their product marketing, lead generation, and thought leadership activity.

Red Hat is the world's leading provider of enterprise open source software solutions, using a community-powered approach to deliver reliable and high-performing Linux, hybrid cloud, container, eventing, and Kubernetes technologies. Red Hat helps customers develop cloud-native applications, integrate existing and new IT applications, and automate and manage complex environments. A trusted adviser to the Fortune 500, Red Hat provides award-winning support, training, and consulting services that bring the benefits of open innovation to any industry. Red Hat is a connective hub in a global network of enterprises, partners, and communities, helping organizations grow, transform, and prepare for the digital future.

Copyright (©) 2020 RTInsights