

Compliments of



Apache Pulsar IN ACTION

David Kjerrumgaard



MANNING



The unified messaging & streaming platform made by the creators of Apache Pulsar.

Built for Kubernetes. Made for the cloud.
Enables multi-cloud and hybrid cloud strategies.



**Stream
Native
Cloud**

A fully managed enterprise SaaS offering of Pulsar



**Stream
Native
Platform**

A self-managed enterprise software offering of Pulsar

The StreamNative Enterprise Offering Includes:

- ✓ Enterprise-ready authentication and authorization
- ✓ Pulsar Kubernetes Operators for Pulsar clusters on Kubernetes
- ✓ Function Mesh for Pulsar Functions and connectors on Kubernetes
- ✓ Full compatibility with the Kafka, AMQP, and MQTT API

Why StreamNative

StreamNative was founded by the original creators of Apache Pulsar and has more experience designing, deploying, and running large-scale Apache Pulsar instances than any team in the world.

Trusted by Innovators



Get in Touch Today: streamnative.io/en/contact



MEAP Edition
Manning Early Access Program
Apache Pulsar in Action
Version 9

Copyright 2021 Manning Publications

For more information on this and other Manning titles go to
manning.com

welcome

As a frequent subscriber to Manning's early access program, I am thrilled to welcome you the MEAP for my very own book, *Apache Pulsar in Action!* In order to get the most out of this book, you should have some established skills in Java programming, along with basic experience running container-based applications with Docker.

Pulsar is a complex messaging system that was originally created at Yahoo and has been contributed to the Apache Software Foundation. It was designed to provide fast, scalable, and durable messaging and much more. I see Pulsar raising the standard for both messaging and storage platforms as the world shifts away from batch-processing and towards real-time stream processing.

The book is divided into three parts. Part 1 will cover the basics of messaging, streaming data, and stream processing with a focus on how Pulsar provides these features. It will also cover Pulsar unique architecture and contrast it against the other popular messaging frameworks such as Kafka.

The second part will focus on Pulsar's native stream processing capability, known as Pulsar Functions, that makes it easy to deploy lightweight computing logic using simple functions written in one of several popular programming languages including Python, Java, .NET, and Go. It will also cover how to use Pulsar Functions to develop event-driven applications and microservices by providing in-depth examples that you can download and try yourself.

The third and final part will focus on Stream storage, and how Pulsar's unique de-coupled architecture enables retain messages indefinitely on cloud storage services such as AWS S3 or Azure Blob storage. We will also explore how this capability directly supports event sourcing and the development of applications that follow the Command Query Responsibility Segregation (CQRS) architecture pattern. Lastly, we will demonstrate how to query raw message data inside Pulsar using standard SQL query syntax by leveraging the Presto execution engine.

Your feedback is essential to creating the best book possible, and I encourage you to please share your comments, questions, and suggestions in Manning's [liveBook Discussion Forum](#) for my book.

—David Kjerrumgaard

brief contents

PART 1: GETTING STARTED WITH APACHE PULSAR

- 1 Introduction to Apache Pulsar*
- 2 Pulsar Concepts and Architecture*
- 3 Interacting with Pulsar*

PART 2: APACHE PULSAR DEVELOPMENT ESSENTIALS

- 4 Pulsar Functions*
- 5 Pulsar IO Connectors*
- 6 Pulsar Security*
- 7 Schema Registry*

PART 3: HANDS-ON APPLICATION DEVELOPMENT WITH PULSAR

- 8 Pulsar Function Patterns*
- 9 Resiliency Patterns*
- 10 Data Access*
- 11 Machine Learning in Pulsar*
- 12 Edge Analytics*

APPENDIXES

- A Running Pulsar on Kubernetes*
- B Geo-Replication*

1

Introduction to Apache Pulsar

This chapter covers

- The evolution of the enterprise messaging system and why Apache Pulsar represents the next step in the evolutionary process.
- A comparison of Apache Pulsar to existing enterprise messaging systems
- How Pulsar's segment-centric storage differs from the partition-centric storage model used in Apache Kafka.
- Real-world use cases where Pulsar is used for stream processing and why you should consider using Apache Pulsar.

Developed at Yahoo in 2013, Pulsar was first open sourced in 2016, and only 15 months after joining the Apache Software Foundations' incubation program graduated to Top Level Project status. Apache Pulsar was designed from the ground up to address the gaps in current open-source messaging systems such as multi-tenancy, geo-replication, and strong durability guarantees.

The Apache Pulsar site describes it as a distributed pub-sub messaging system that provides very low publish and end-to-end latency, guaranteed message delivery, zero data loss, and a serverless lightweight computing framework for stream data processing. Apache Pulsar provides the three key capabilities for processing large data sets:

- **Real-time messaging:** Enable geo-graphically distributed applications and systems to communicate with one another in an asynchronous manner by exchanging messages. Pulsar's goal is to provide this capability to the broadest audience of clients via support for multiple programming languages and binary messaging protocols.
- **Real-time compute:** Provides the ability to perform user-defined computations on these messages inside of Pulsar itself and without the need for an external computational system to perform basic transformational operations such as data

enrichment, filtering, and aggregations.

- **Scalable storage:** Pulsar's independent storage layer and support for tiered-storage enable the retention of your message data for as long as you need. There is no physical limitation on the amount of data that can be retained and accessed by Pulsar.

1.1 Enterprise Messaging Systems

Messaging is a broad term that is used to describe the routing of data between producers and consumers. Consequently, there are several different technologies and protocols that have evolved over the years that provide this capability. Most people are familiar with messaging systems such as email, text messaging, and instant messaging applications such as WhatsApp or Facebook messenger. Messaging systems within this category are designed to transmit text data and images over the internet between two or more parties. More advanced instant messaging systems support Voice over IP and video chat capabilities as well. All of these systems were designed to support person-to-person communication over ad-hoc channels.

Another category of messaging system that people are already familiar with are the video-on-demand streaming services such as Netflix or Hulu that stream video content to multiple subscribers simultaneously. These video streaming services are examples of one-way, broadcast (one message to many consumers) transmission of data to consumers that subscribe to an existing channel in order to receive the content. While these types of applications might be what comes to mind when using the terms "messaging systems" and/or "streaming", for the purposes of this book, we will be focusing on enterprise messaging systems.

An *Enterprise Messaging System* (EMS) is the software that provides the implementation of various messaging protocols such as DDS, AMQP, MSMQ, etc. These protocols support the sending and receiving of messages between distributed systems and applications in an asynchronous fashion. However, asynchronous communication wasn't always an option, particularly during the earliest days of distributed computing when both client/server and remote procedure call (RPC) architectures were the dominant approach. A prime example of RPC were the SOAP and REST based web services that interacted with one another through fixed endpoints.

Within both of these styles, when a process wanted to interact with a remote service it needed to first determine the service's remote location via a discovery service and then invoke the desired method remotely, using the proper parameters and types, as shown in Figure 1.1.

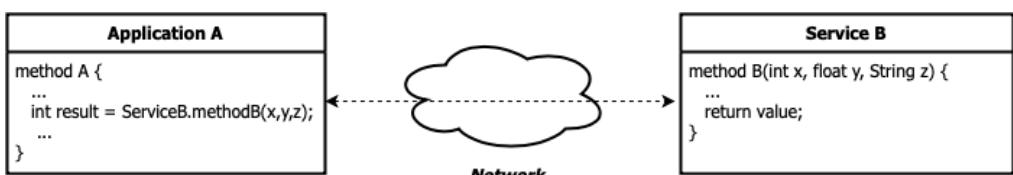


Figure 1.1: Within an RPC architecture, an application invokes a procedure on a service that is running on a

different host and must wait for that procedure call to return before it can continue processing.

The calling application would then have to wait for the called procedure to return before it could continue processing. The synchronous nature of these architectures made application based upon them inherently slow. In addition, there was the possibility that the remote service was unavailable for a period of time, which would require the application developer to use defensive programming techniques to identify this condition and react accordingly.

Unlike the point-to-point communication channels used in RPC programming where you had to wait for the procedure calls to provide a response, enterprise messaging systems allow remote applications and services to communicate with one another via an intermediate service rather than directly with one another. Rather than having to establish a direct network communication channel between the calling/receiving applications over which the parameters are exchanged. An enterprise messaging system can be used to retain these parameters in message form and are guaranteed to be delivered to the intended recipient for processing. This allows the caller to send its request asynchronously and await a response from the service they were trying to invoke. It also allows the service to communicate its response back in an asynchronous manner as well by publishing its result to the EMS for eventual delivery to the original caller.

This decoupling promotes asynchronous application development by providing a standardized, reliable intra-component communication channel that serves as a persistent buffer for handling data, even when some of the components are offline as you can see from Figure 1.2.

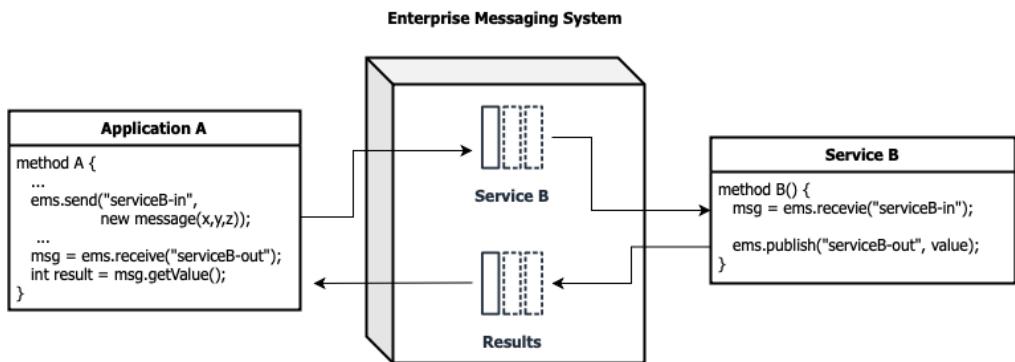


Figure 1.2 An enterprise messaging system allows distributed applications and services to exchange information in an asynchronous fashion.

Enterprise messaging systems promote loosely coupled architectures by allowing independently developed software components that are distributed across different system to communicate with one another via structured messages. These message schemas are usually defined in language-neutral formats such as XML, JSON, or Avro IDL, which allows the components to be developed in any programming language that supports those formats.

1.1.1 Key Capabilities

Now that we have introduced that concept of enterprise message systems and provided some context into the type of problems that have been used to solve, let's further refine the definition of what an EMS is, based upon the capabilities it provides.

ASYNCHRONOUS COMMUNICATION

Messaging systems allow services and applications to communicate with one another in a non-blocking manner, meaning that the message sender and receiver are not required to interact with the messaging system (or one another) at the same time. A messaging system will retain the messages until the intended all of the intended recipients consume it.

MESSAGE RETENTION

Unlike network-based messaging, such as RPC, in which the messages only exist on the network, messages published to a messaging system are retained on disk until they are delivered. Undelivered messages can be held for hours, days, or even weeks, and most messaging systems allow you to specify the retention policy.

ACKNOWLEDGEMENT

Messaging systems are required to retain messages until all of the intended recipients receive it, therefore a mechanism by which the message consumers can acknowledge the successful delivery and processing of the message is required. This allows the messaging system to purge all successfully delivered messages, and to retry message delivery to those consumers who have not yet received it.

MESSAGE CONSUMPTION

Obviously, a messaging system isn't particularly useful if it doesn't provide a mechanism by which the intended recipients can consume messages. First and foremost, an EMS must guarantee that all the messages it receives get delivered. Often times, a message might be intended for multiple consumers, and the EMS must maintain the information along with of which messages have been delivered and to whom.

1.2 Message Consumption Patterns

With enterprise messaging systems, you have the option of publishing messages to either a topic or a queue, and there are fundamental differences between the two. A topic supports multiple concurrent consumers of the same message. Any message published to a topic is automatically broadcast to all of the consumers that have *subscribed* to the topic. Any number of consumers can subscribe to a topic in order to receive the information being sent, kind of like any number of users can subscribe to Netflix and receive their streaming content.

1.2.1 Publish-Subscribe Messaging

In publish and subscribe messaging, producers publish messages to named channels known as "topics". Consumers can then subscribe to those topics to receive the incoming messages. A publish-subscribe (pub-sub) message channel receives incoming messages from multiple

producers and stores them in the exact order that they arrive. However, it differs from message queuing on the consumption side because it supports multiple consumers receiving each message in a topic via a subscription mechanism as shown below in Figure 1.3.

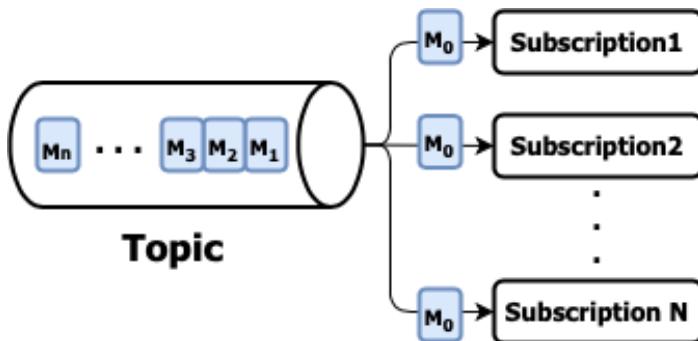


Figure 1.3 With pub-sub message consumption, each message is delivered to each and every subscription that has been established on the topic. In this case, Message M_0 was delivered to Subscriptions 1 thru N inclusive.

Publish-subscribe messaging systems are ideally suited for use cases that require multiple consumers to receive each message, or those in which the order in which the messages are received and processed is crucial to maintaining a correct system state. Consider the case of a stock price service that can be used by a large number of systems. Not only is it important that these services receive all the messages, but it is also equally important that the price changes arrive in the correct order.

1.2.2 Message Queuing

Queues on the other hand provide first in, first out (FIFO) message delivery semantics to one or more competing consumers as shown in Figure 1.4. With queues, the messages are delivered in the order they are received and only one message consumer receives and processes individual message rather than all of them. These are perfect for queuing up messages that represent events that trigger some work to be performed, such as orders into a fulfillment center for dispatch. In this scenario, you only want each order processed just once.

Message queues can easily support higher rates of consumption by scaling up the number of consumers in the event of a high number of backlogged messages. To ensure that a message is processed exactly once, each message must be removed from the queue after it has been successfully processed and acknowledged by the consumer. Due to its exactly-once processing guarantees, message queuing is ideal for work queue use cases.

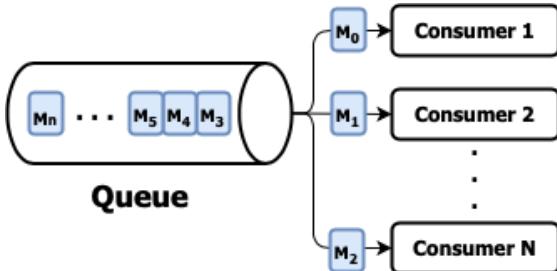


Figure 1.4 With queue-based message consumption, each message is delivered to exactly one consumer. In this case, Message M₀ was consumed by Consumer1, M₁ by Consumer2, etc.

In the event of consumer failures (meaning no acknowledgement is received within a specified timeframe), the message will be resent to another consumer. In such a scenario, the message will most likely be processed out of order. Therefore, message queues are well suited for use cases where it is critical that each message is processed exactly once, but the order in which the messages are processed is not important.

1.3 The Evolution of Messaging Systems

Now that we have clearly defined what constitutes an enterprise messaging system along with the core capabilities it provides, I would like to provide a brief historical review of messaging systems and how they have evolved over the years. Messaging systems have been around for decades and have been effectively used within many organizations, so Apache Pulsar isn't some brand new technology that emerged on the scene, but rather another step in the evolution of the messaging system. By providing some historical context, my hope is that you will be able to understand how Pulsar compares to existing messaging systems.

1.3.1 Generic Messaging System

Before I jump into specific messaging systems, I wanted to present a simplified representation of a messaging system in order to highlight the underlying components that all messaging systems have. Identifying these core features, will provide a basis for comparison between messaging systems over time.

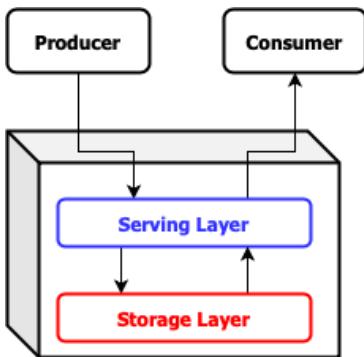


Figure 1.5: Every messaging system can be separated into two distinct architectural layers.

As you can see from Figure 1.5 every messaging system consists of two primary layers, each with its own specific responsibilities that we will explore next. We shall examine the evolution of messaging systems across each of these layers in order to properly categorize and compare different messaging systems including Apache Pulsar.

SERVING LAYER

The serving layer is a conceptual layer within an enterprise messaging system that interacts directly with the message producers and consumers. Its primary purpose is to accept incoming messages and route them to one or more destinations. Therefore, it communicates via one or more of the supported messaging protocols, such as DDS, AMQP, MSMQ, etc. Consequently, this layer is heavily dependent on network bandwidth for communication, and CPU for message protocol translation.

STORAGE LAYER

The storage layer is the conceptual layer within an enterprise messaging system that is responsible for the persistence and retrieval of the messages. It interacts directly with the serving layer to provide the requested messages and is responsible for retaining the proper order of the messages. Consequently, this layer is heavily dependent on disk for message storage.

1.3.2 Message Oriented Middleware

The first category of messaging systems is often referred to as message-oriented middleware (MOM) which was designed to provide inter-process communication and application integration between distributed systems running on different networks, and operating systems, etc. One of the most prominent MOM implementations was IBM WebSphere MQ, which debuted in 1993.

The earliest implementations were designed to be deployed upon a single machine that was often located deep within the company's datacenter on a fairly large machine. Not only was this a single-point-of-failure, it also meant that the scalability of the system was limited to the physical hardware capacity of the host machine because this single server was

responsible for handling all client requests and storing all messages as shown in Figure 1.6. The number of concurrent producers and consumers these single-server MOM systems could serve was limited by the bandwidth of the network card, and the storage capacity was limited by the physical disk on the machine.

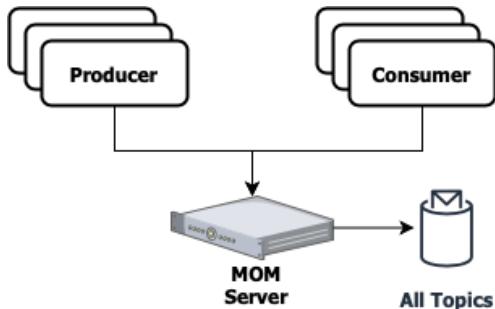


Figure 1.6: Message-Oriented Middleware was designed to be hosted on a single server and therefore hosted all of the message topic and handled requests from all the clients.

To be fair, these limitations were not limited to just IBM, but are rather a limitation of all messaging systems that were designed to be hosted on a single machine, including RabbitMQ and RocketMQ, among many others. Furthermore, this limitation wasn't limited to just messaging systems, but rather was pervasive across all types of enterprise software such as databases, etc. at the time that were designed to run on one physical host.

CLUSTERING

Eventually these scalability issues were addressed through the addition of clustering capabilities to these single-server MOM systems. This allowed multiple single-service instances to share the processing of the messages and provide some load balancing as shown in Figure 1.7. Even though the MOM was clustered, in reality it just meant that each single-service instance was responsible for serving and storing messages for a sub-set of all the topics. A similar approach, called *sharding*, was taken by relational databases during this period as well to address this scalability issue.

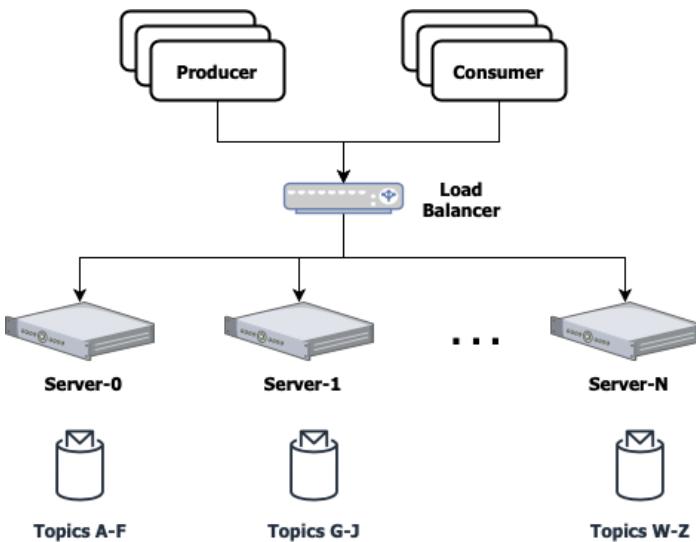


Figure 1.7: Clustering allowed the load to be spread across multiple servers instead of just one. Each server in the cluster was responsible for handling only a portion of the topics.

Even though the MOM was clustered, in reality it just meant that each single-service instance was responsible for serving and storage messages for a sub-set of all the topics, and in the event of topic “hot-spots” the unlucky server assigned that particular topic could still become a bottleneck and/or potentially run out of storage capacity as well. In the event any one of these servers in the cluster were to fail, it would take all of the topics it was serving down with it. While this did minimize the impact of the failure to the cluster as a whole, e.g., it continued to run, it was a single-point-of-failure for this particular topics/queues it was serving.

This limitation required organizations to meticulously monitor their message distribution in order to align their topic distribution to match their underlying physical hardware to ensure that the load was evenly distributed across the cluster. Even then, there was still the possibility that a single topic could still be problematic. Consider the scenario where you work for a major financial institution and you wanted a single topic to store all the trade information for a particular stock and provide this information to all the trade desk applications within your organization. The sheer number of consumers and volume of data for this one topic could easily overwhelm a single-server that was dedicated to serving just that topic. What was needed in such a scenario is the ability to distribute the load of a single topic across multiple machines, which as we shall see, is exactly what distributed messaging systems do.

1.3.3 Enterprise Service Bus

Enterprise Service Buses (ESB) emerged during the early part of this century when XML was the preferred message format used for implementing Service-Oriented-Architecture (SOA)

applications using SOAP-based web services. The core concept of ESBs was the “message bus”, as shown in Figure 1.8, that served as a communication channel between all applications and services. This centralized architecture is in direct contrast to the point-to-point integration previously used by other message-oriented middleware.

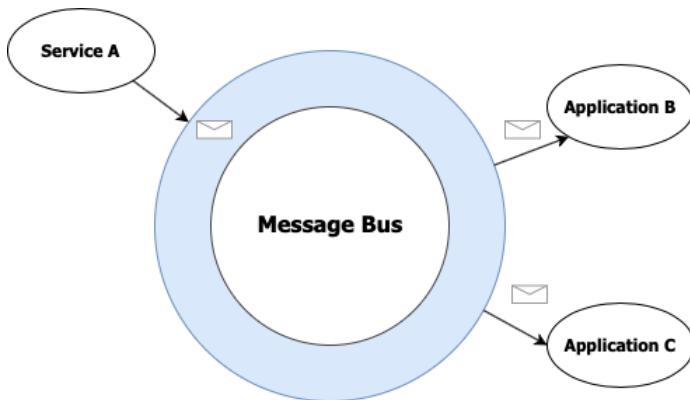


Figure 1.8: The core concept of Enterprise Service Buses is the use of a message bus in order to eliminate the need for point-to-point communication. Service A merely publishes its message to the bus, and it is automatically routed to applications B and C.

With an ESB each application or service would send and receive all its messages over a single communication channel rather than having to specify the specific topic names they wanted to publish and consume from. Each application would “register” itself with the ESB and specify a set of rules used to identify which messages it was interested in, and the ESB would handle all of the logic necessary to dynamically route messages from the bus that matched those rules. Similarly, each service was no longer required to know the intended target(s) of its messages beforehand and could simply publish its messages to “the bus” and allow it to route the messages.

Consider the scenario where you have a large XML document that contains 100s of individual line items within a single customer order, and you wanted to route only a subset of those items to a service based upon some criteria within the message itself, e.g., by product category or department. An ESB provided the capability to extract those individual messages (based on the results of a XQuery) and route them to different consumers based on the content of the message itself.

In addition to these dynamic routing capabilities, ESBs also took the first evolutionary step down the road of “stream processing”, by emphasizing the capabilities to process the messages inside the messaging system itself rather than having the consuming applications perform this task. Most ESBs provided message transformation services, often via XSLT or XQuery, that handled the translation of message formats between the sending and receiving services. They also provided message enrichment and processing capabilities into the message system itself, which up until that point had been performed by the applications receiving the messages. This was a fundamentally new way of thinking about messaging

systems, that up until that point had always been used almost exclusively as a transportation mechanism.

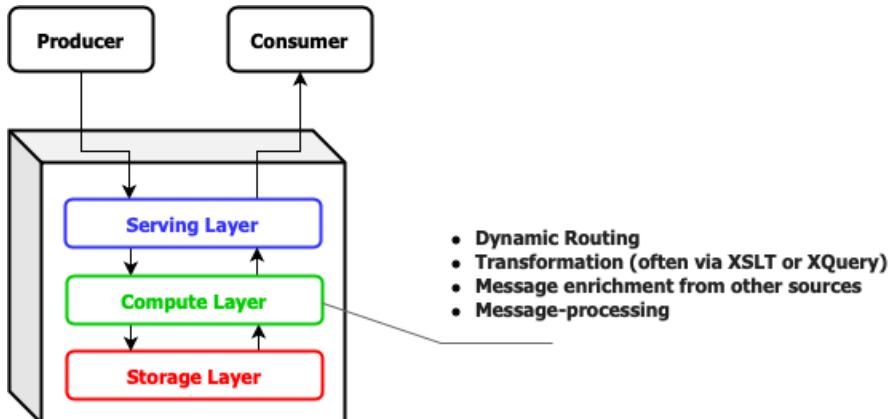


Figure 1.9: ESB's emphasis on dynamic routing and message processing represented the first time “stream processing” capabilities were added to a messaging system. This introduced a whole new architectural layer to the base messaging system architecture.

One could argue that the ESB was the first category of enterprise messaging system to introduce a third layer to the basic architecture of messaging systems, as shown in Figure 1.9. In fact, today most modern ESBs support more advanced computing capabilities including process choreography for managing business process flows, complex event processing for event correlation and pattern-matching, and out of the box implementations of several enterprise integration patterns.

The ESB’s other significant contribution to the evolution of the messaging system was its focus on integration with external systems which forced messaging systems to support a wide variety of non-messaging protocols for the first time. While ESB’s still fully-support AMQP and other pub/sub messaging protocols, a key differentiator of ESB was its ability to move data onto and off of the bus from non-message-oriented systems such as email, databases, and other third-party systems. In order to do this, ESBs provided SDKs that allowed developers to implement their own adapters to integrate with their system of choice.

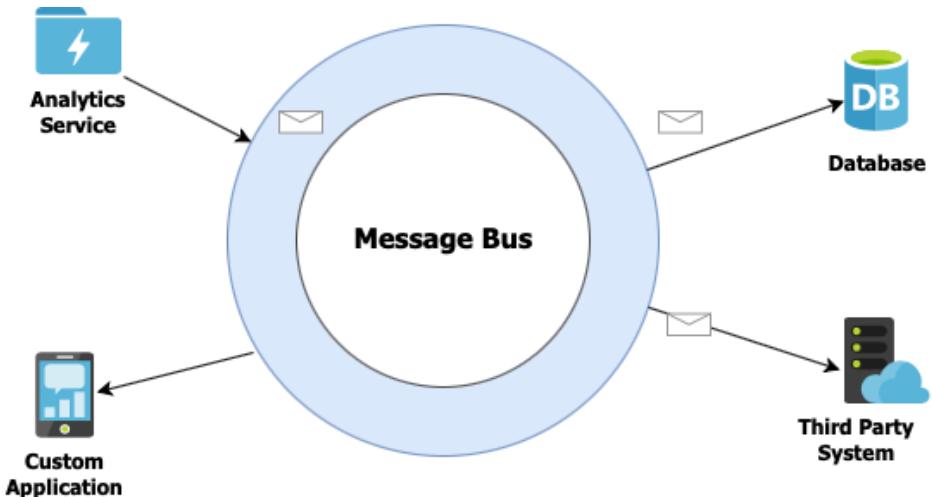


Figure 1.10: Enterprise Service Buses supported the integration of non-message-based systems into the message bus; thereby expanding the messaging capabilities beyond applications and into third-party applications such as databases.

As you can see from Figure 1.10, this allowed data to be more readily exchanged between systems, which simplified the integration of a variety of systems. In this role, the ESB served as both the message-passing infrastructure as well as the mediator between the systems that provided the protocol transformation.

While ESBs undoubtedly pushed enterprise messaging systems forward with these innovations and features and are still very popular today, they are centralized systems that are designed to be deployed on a single host. Consequently, they suffer from the same scalability issues as their MOM predecessors.

1.3.4 Distributed Messaging Systems

A distributed system can be described as a collection of computers working together to provide a service or feature such as a filesystem, key-value store, database, etc. That acts as though they are running on a single computer to the end user. That is to say, the end user isn't aware of the fact that the service is being provided by a collection of machines working together. Distributed systems have a shared state, operate concurrently, and are able to tolerate hardware failures without affecting the availability of the system as a whole.

When the distributed computing paradigm started becoming widely adopted, as popularized by the Hadoop computing framework, etc., the single-machine constraint was lifted. This ushered in an era where new systems were developed that distributed the processing and storage across multiple machines. One of the biggest benefits of distributed computing is the ability to scale the system horizontally simply by adding new machines to the system. Unlike their non-distributed predecessors that were previously constrained to the

physical hardware capacity of single machine, these newly developed systems could now leverage the resources from 100s of machines easily and cost-effectively

As you can see from Figure 1.11, messaging systems, just like databases and computation frameworks, have also made the transition to the distributed computing paradigm as well. Newer messaging systems, with Apache Kafka being the first, and more recently Apache Pulsar have adopted the distributed computing model in order to provide the scalability and performance required by modern enterprises.

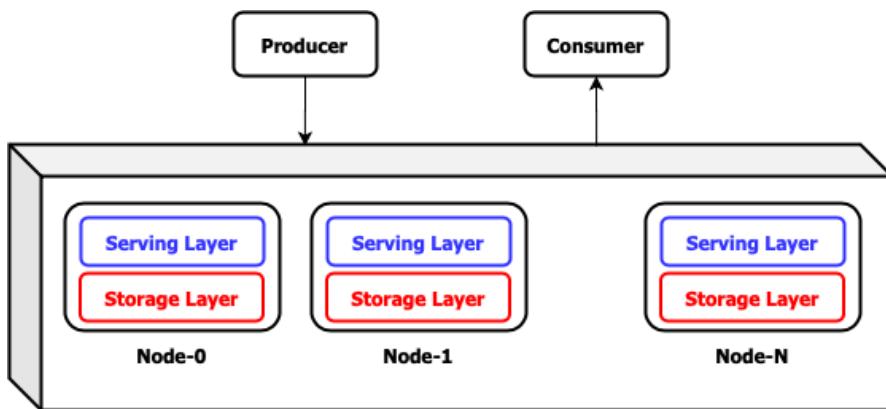


Figure 1.11: Within a distributed messaging system, several nodes act together to behave as a single logic system from the perspective of the end user. Internally, the data storage and message processing are distributed across all the nodes.

Within a distributed messaging system, the contents of a single topic are distributed across multiple machines in order to provide horizontally scalable storage at the message layer, something that was not possible with previous messaging systems. Distributing the data across several nodes in the cluster also provides several advantages including redundancy and high availability of the data, increased storage capacity for messages, increased message throughput due to the increased number of message brokers, and elimination of single point of failure within the system.

The key architectural difference between a distributed messaging system and a clustered single-node system, is the way in which the storage layer is designed. In the previous single-node systems, the message data for any given topic was all stored together on the same machine which allowed the data to be served quickly from local disk. However, as we mentioned earlier, this limited the size of the topic to the capacity of the local disk on that machine. Within a distributed messaging system, the data is distributed across several machines within the cluster. This distribution of data across multiple machines, allows us to retain messages within an individual topic that exceed the storage capacity of an individual machine. The key architectural abstraction that makes this distribution of data possible is the "write-ahead-log", which treats the contents of a message queue as single append-only data structure that messages can be stored in.

As you can see from Figure 1.12, from a logical perspective when a new message is published to the topic it is appended to the end of the log. However, from a physical perspective the message can be written to any server within the cluster.

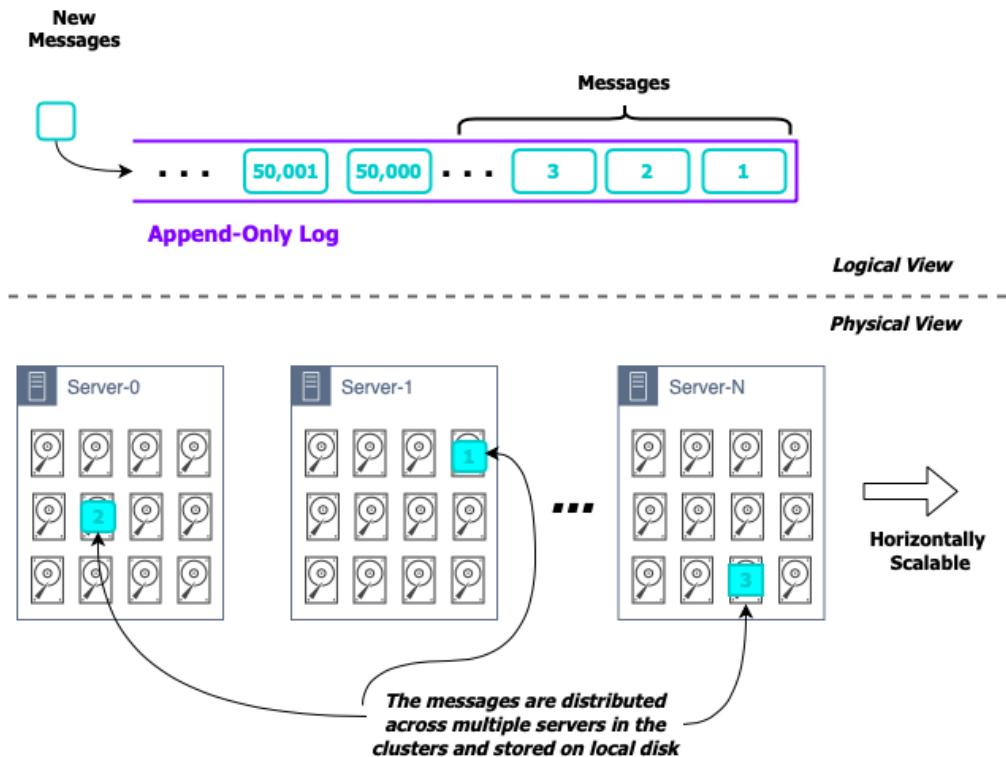


Figure 1.12: The key architectural concept underlying distributed messaging systems is the append-only log, aka the write ahead log. From a logical perspective, the messages within a topic are all stored sequentially, but are stored in a distributed fashion across multiple servers.

This provides distributed messaging systems with a far more scalable storage capacity layer than the previous generations of messaging systems. Another benefit of the distributed messaging architecture is the ability of more than one broker to serve the messages for any given topic, which increases the message production and consumption throughput by spreading the load across multiple machines. For example, messages published to the topic shown in Figure 1.12 would be handled by three separate servers, each with its own write path to disk. This would result in a higher write rate since the load is spread across multiple disks rather than just a single disk as it was in the previous generation of messaging systems. There are two distinct approaches taken when it comes to how the data is distributed across the nodes in the cluster: partition-based and segment-based.

PARTITION-CENTRIC STORAGE IN KAFKA

When using the partition-based strategy within a messaging system, the topic is divided into a fixed number of equal sized groupings known as partitions. Data that is published to the topic is evenly distributed across the partitions as shown in Figure 1.13 with each partition receiving one-third of the messages published to the topic. The total storage capacity of the topic is now equal to the number of partitions in the topic times the size of each partition. Once this limit is reached no more data can be added to the topic. Simply adding more brokers to the cluster will not alleviate this issue because you will also need to increase the number of partitions in the topic which must be performed manually. Furthermore, increasing the number of partitions also requires a rebalance to be performed which, as I will discuss, is an expensive and time-consuming process.

Within a partition-centric storage-based system, the number of partitions is specified when the topic is created, as this allows the system to determine which node will be responsible for storing the which partiton, etc. However, pre-determining the number of partitions has a few unintended side-effects including:

- A single partition can only be stored on a single node within the cluster, so the size of the partition is limited to amount of free disk space on that node.
- Since the data is evenly distributed across all partitions, each partition is limited to the size of smallest partition in the topic. For instance, if a topic is distributed across 3 nodes with 4TB, 2TB, and 1TB of free disk respectively, then the partition on the third node can only grow to 1TB in size, which in turn means all partitions in the topic can only grow to 1TB as well.
- Although it isn't strictly required, each partition is usually replicated multiple times to different nodes to ensure data redundancy. Therefore, the maximum partition size is further restricted to the size of the smallest replica.

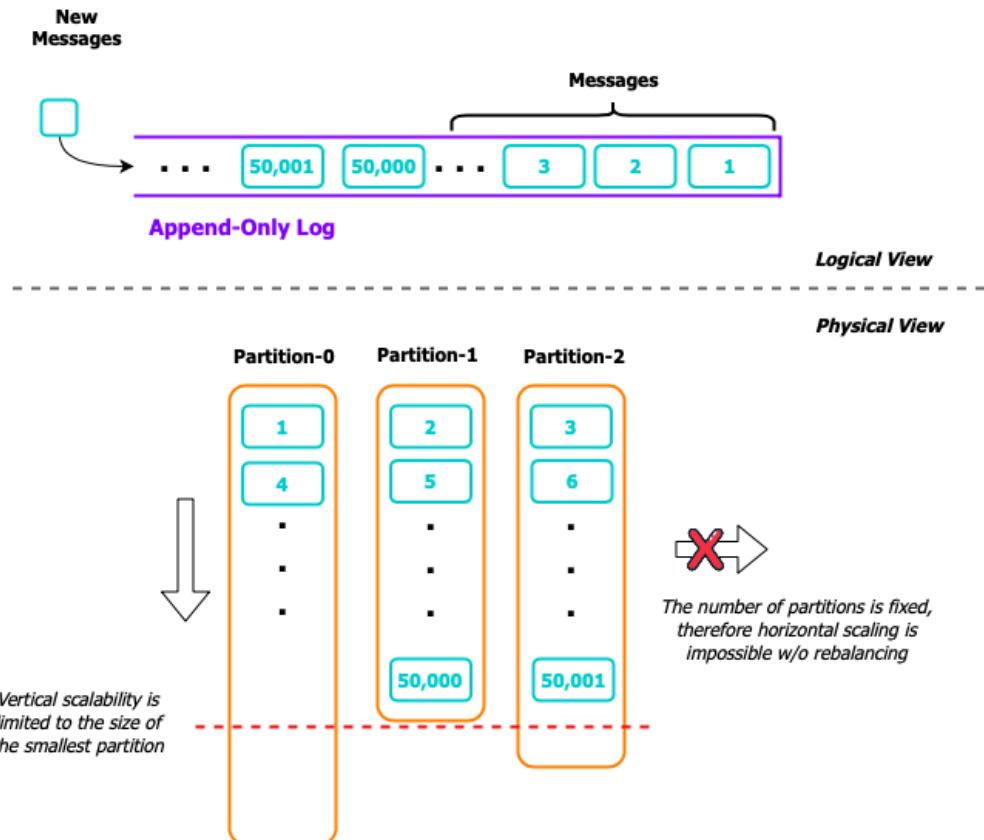


Figure 1.13: Message Storage in a Partition-Based Messaging System

In the event you run into one of these capacity limitations, your only remedy is to increase the number of partitions in the topic. However, this capacity expansion processes requires rebalancing the entire topic as shown in Figure 1.14. During this rebalancing process the existing topic data is redistributed across all of the topic partitions in order to free up disk space on the existing nodes. Therefore, when you add a fourth partition to an existing topic each partition should have approximately 25% of the total messages once the rebalancing process has completed.

This recopying of data is expensive and error prone as it consumes network bandwidth and disk I/O directly proportional to the size of the topic, e.g., rebalancing a 10TB topic would result in 10TB of data being read from disk, transmitted over the network, and written to disk on the target brokers. Only after the rebalancing process has completed, can the previously existing data be deleted, and the topic can resume serving clients. Therefore, it is advisable to choose your partition sizing wisely, as the cost to rebalance cannot be easily dismissed.

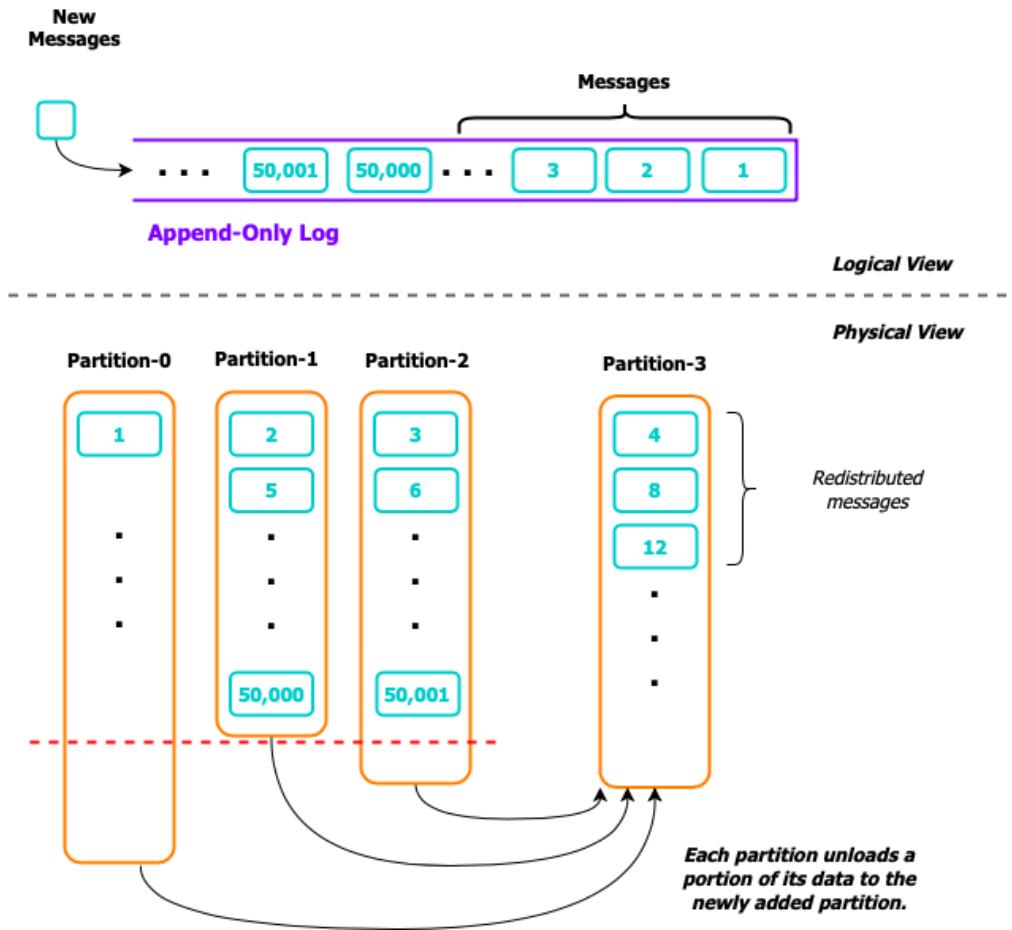


Figure 1.14: Increasing the storage capacity of a partition-based topic incurs the cost of rebalancing, in which a portion of the data from the existing partitions is copied over to the newly added partition(s) in order to free up disk space on the existing nodes.

In order to provide redundancy and fail-over for the data, you can configure the partitions to be replicated across multiple nodes. This ensures that there is more than one copy of the data available on disk even in the event of a node failure. The default replica setting is three, which means that the system will retain three copies of each message. While this is a good trade-off in terms of space for redundancy, you need to account for this additional storage requirement when you size your Kafka cluster.

SEGMENT-CENTRIC STORAGE IN PULSAR

Pulsar relies upon the Apache BookKeeper project to provide the persistent storage of its messages. BookKeeper's logical storage model is based on the concept of boundless streams

entries stored as a sequential log. As you can see from Figure 1.15, within BookKeeper each log is broken down into smaller chunks of data known as segments which in turn are comprised of multiple log entries. These segments are then written across a number of nodes known as bookies in the storage layer for redundancy and scale.

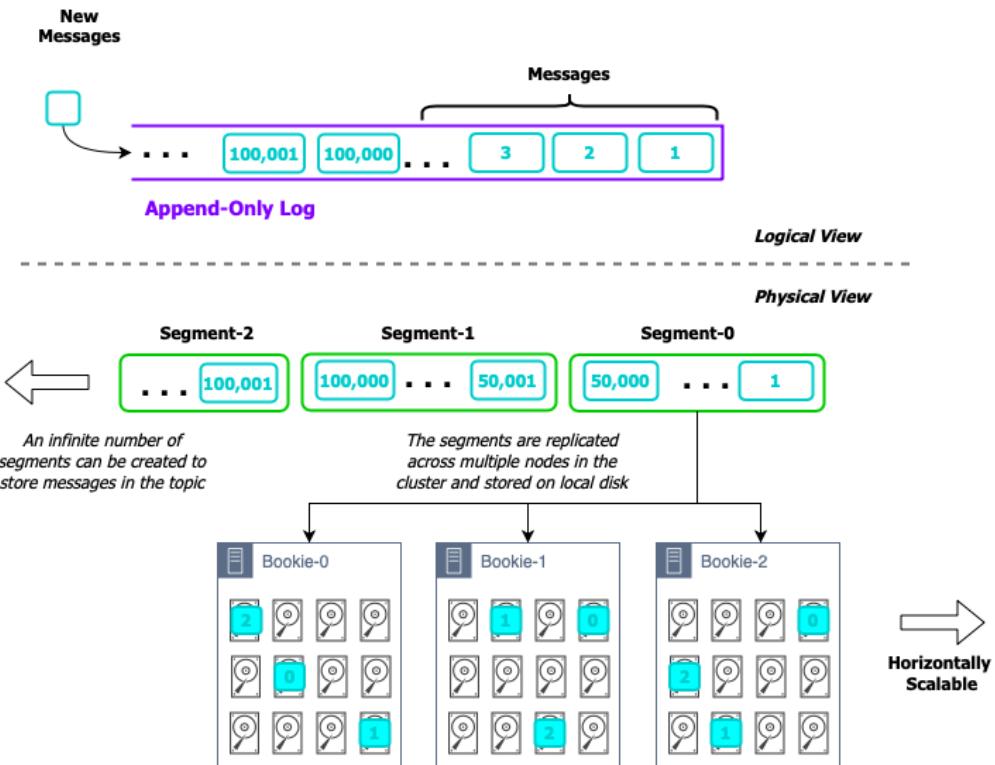


Figure 1.15: Message storage in a segment-centric messaging system is done by writing a pre-determined number of messages into a “segment” and then storing multiple replicas of the segment across different nodes in the storage layer.

As you can see from Figure 1.15, the segments can be placed anywhere on the storage layer that has sufficient disk capacity. When there isn't sufficient storage capacity in the storage layer for new segments, new nodes can be easily added and used immediately for storing data. One of the key benefits of segment-centric storage architecture is true horizontal scalability as segments can be created indefinitely and stored anywhere. Unlike partition-centric storage which imposes artificial limitations to both vertical and horizontal scaling based on the number of partitions.

1.4 Comparison to Apache Kafka

Apache Kafka and Apache Pulsar are both distributed messaging systems that have similar messaging concepts. Clients interact with both systems via topics that are logically treated as unbounded, append-only streams of data. However, there are some fundamental differences between Apache Pulsar and Apache Kafka when it comes to scalability, message consumption, data durability, and message retention.

1.4.1 Multi-Layered Architecture

Apache Pulsar's Multi-layered architecture completely decouples the message serving layer from the message storage layer, allowing each to scale independently. Traditional distributed messaging technologies such as Kafka have taken the approach of co-locating data processing and data storage on the same cluster nodes or instances. That design choice offered a simpler infrastructure and some performance benefits due to reducing transfer of data over the network, but at the cost of a lot of tradeoffs that impact scalability, resiliency, and operations.

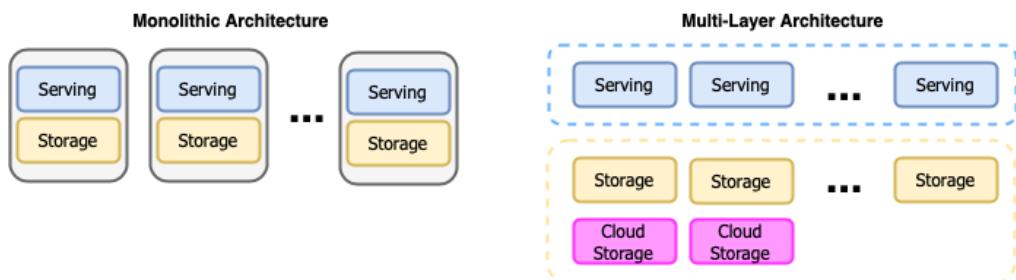


Figure 1.16: Monolithic distributed architectures co-locate the serving and storage layers. While Pulsar uses a multi-layer architecture that decouples the storage and serving layers from one another, which allows them to scale independently.

Pulsar's architecture takes a very different approach, one that's starting to gain traction in a number of "cloud-native" solutions and that is made possible in part by the significant improvements in network bandwidth that are commonplace today, namely separation of compute and storage. Pulsar's architecture decouples data serving and data storage into separate layers: data serving is handled by stateless "broker" nodes, while data storage is handled by "bookie" nodes as shown in Figure 1.16. This decoupling has several benefits including dynamic scalability, zero down-time upgrades, and infinite storage capacity upgrades just to name a few. Further, this design is container-friendly, making Pulsar the ideal technology for hosting a cloud native streaming system.

DYNAMIC SCALING

Consider the case where we have a service that is CPU-intensive and whose performance starts to degrade when the requests exceed a certain threshold. In such a scenario, we need to horizontally scale the infrastructure to provide new machines and instances of the

application to distribute the load when the CPU usage goes above 90% on the current machine. Rather than relying on a monitoring tool to alert your DevOps team to this condition and having them perform this process manually, it would be preferable to have the entire process automated.

Autoscaling is a common feature of all public cloud providers such as AWS, Microsoft Azure, and Google Cloud as well as Kubernetes. It allows auto-scaling the infrastructure horizontally based upon resource utilization metrics such as CPU/Memory, etc. without any human interaction. Now while it is true that this capability is not exclusive to Pulsar and can be leveraged by any other messaging platforms to scale up during high traffic conditions. This capability is much more useful in a multi-tiered architecture such as Pulsar's for two reasons we will discuss.

Pulsar's stateless brokers in the serving layer also enables the ability to scale the infrastructure down once the spike has past, which translates directly in cost savings in a public cloud environment. Other messaging systems that use a monolithic architecture cannot scale down the nodes due to the fact that the nodes contain data on their attached hard drives. Removal of the excess nodes can only be done once that data has been completely processed or has been moved to another node that will remain in the cluster. Neither of these can be performed in an automated fashion easily.

Secondly, in a monolithic architecture such as Apache Kafka, the broker can only serve requests for data that is stored on an attached disk. This limits the usefulness of auto-scaling the cluster in response to traffic spikes because the newly added nodes to the Kafka cluster **will not have any data to serve**, and therefore will not be able to handle any of incoming requests to read existing data from the topics. The newly added nodes will only be able to handle write requests.

Lastly, in a monolithic architecture such as Apache Kafka, horizontal scaling is achieved by adding new nodes that have both storage and serving capacity regardless of which metric you are tracking and responding to. Therefore, when you scale up your serving capacity in response to high CPU you are also scaling up your storage capacity whether you actually need additional storage or not.

AUTO-RECOVERY

Before you move your messaging platform into production, you will need to understand how to recover from various failure scenarios, starting with a single node failure. In a multi-tiered architecture such as Pulsar, the process is very straight-forward. Since the broker nodes are stateless, they can be replaced by spinning up a new instance of the service to replace the one that failed without a disruption of service or any other data replacement considerations. At the storage layer, multiple replicas of the data are distributed across multiple nodes, which can be easily replaced with new nodes in the event of a failure. In either scenario, Pulsar can rely on cloud-provider mechanisms such as auto-scaling groups to ensure that a minimum number of nodes are always running.

Monolithic architectures such as Kafka will suffer again from the fact that newly added nodes to the Kafka cluster **will not have any data to serve**, and therefore will only be able to handle incoming write requests.

1.4.2 Message Consumption

Reading messages from a distributed messaging system is a bit different from reading them from a legacy messaging system, as they were designed to support a large number of concurrent consumers. The way in which the data is consumed is driven in large part to the way that is stored inside the system itself, with both partition-centric and segment-centric systems having their own unique way of supporting pub/sub semantics for consumers.

MESSAGE CONSUMPTION IN KAFKA

Within Kafka, all consumers belong to what is referred to as a consumer group, which forms a single "logical subscriber" for a topic. Each group is composed of many consumer instances for scalability and fault tolerance, if one instance fails the remaining consumers will take over. By default, a new consumer group is created whenever an application subscribes to a Kafka topic, an application can also leverage an existing consumer group by providing the group.id as well.

According to the Kafka documentation, "The way consumption is implemented in Kafka is by dividing up the partitions in the log over the consumer instances so that each instance is the exclusive consumer of a "fair share" of partitions at any point in time." In layman's terms this means that each partition within a topic can only have one consumer at a time, and that the partitions are distributed evenly across the consumers within the group. As shown in Figure 1.17, if a consumer group has less members than partitions, then some consumers will be assigned to multiple partitions, but if you have more consumers than partitions, the excess consumers will remain idle and only take over in the event of a consumer failure.

One important side-effect of creating "exclusive" consumers is that within a consumer group, the number of active consumers can never exceed the number of partitions in the topic. This limitation can be problematic as the only way of scaling data consumption from a Kafka topic is by adding more consumers to a consumer group. This effectively limits the amount of parallelism to the number of partitions, which in turn limits the ability to scale up data consumption in the event that your consumers cannot keep up the topic producers. Unfortunately, the only remedy to this is to increase the number of topic partitions, which as we discussed earlier is not a simple, fast, or cheap operation.

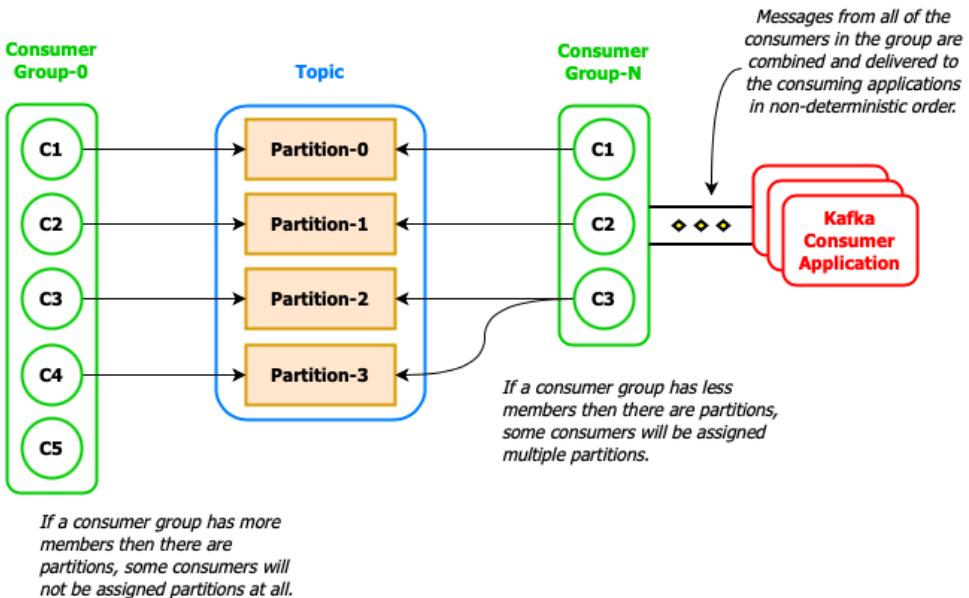


Figure 1.17: Kafka's consumer groups are closely tied to the partition concept. This limits the number of concurrent topic consumers to the number of topic partitions.

You can also see from Figure 1.17 that all of the individual consumer's messages are combined and sent back to the Kafka client. Therefore, message ordering is not maintained by the consumer group. Kafka only provides a total order over records within a partition, not between different partitions in a topic.

As I mentioned earlier, consumer groups act as a cluster to provide scalability and fault-tolerance. This means they dynamically adapt to the addition or loss of consumers within the group. When a new consumer is added to the group, it starts consuming messages from partitions previously consumed by another consumer. The same thing happens when a consumer shuts down or crashes; it leaves the group, and the partitions it used to consume will be consumed by one of the remaining consumers. This shuffling of partition ownership with a consumer group is referred to as *rebalancing* which can have some undesirable consequences including the potential for data loss, if consumer offsets aren't saved before the rebalancing occurred.

It is very common to have multiple applications that need to read data from the same topic, in fact this is one of the primary features of a messaging system. Consequently, topics are shared resources among multiple consuming applications that may have very different consumption needs. Consider a financial services company that streams in real-time stock market quote information into a topic named "stock quotes" and wants to share that information across the entire enterprise. Some of their business-critical applications, such as their internal trading platforms, algorithmic trading systems, and customer facing website,

will all need to process that topic data as quickly as it arrives. This would require a high number of partitions in order to provide the necessary throughput to meet these tight SLAs.

On the other hand, the data science team may want to feed the stock topic data through some of their machine learning models in order train and or validate their models using real stock pricing data. This would require processing the records in exactly the order they were received, which requires a single partition topic to ensure global message ordering.

The business analytics team will develop reports using KSQL that joins the stock topic data with other topic(s) based on a particular key such as the stock ticker, which would benefit from having the topic partitioned by the ticker symbol.

Efficiently providing the stock topic data for these applications with such vastly different consumption patterns would be difficult, if not impossible, given how dependent the consumer groups are tied to the partition number which is a fixed decision that cannot be easily changed. Typically, in such a scenario, your only realistic option is to maintain multiple copies of the data in different topics, each configured with the correct number of partitions for the application.

1.4.3 Data Durability

Within the context of messaging systems, the term data durability refers to the guarantees that messages that have been acknowledged by the system will survive even in the event of a system failure. In a distributed system such as Pulsar or Kafka with many nodes, failures can occur at many levels, therefore it is important to understand how the data is stored and what durability guarantees the system provides.

When a producer publishes a message, an acknowledgement is returned from the messaging system to indicate that the message was received on the topic. This acknowledgment signals to the producer that the message is safe, and that the producer can discard it without worrying about it getting lost. As we shall see the strength of these guarantees are much greater in Pulsar than Kafka.

DATA DURABILITY IN KAFKA

As we discussed earlier, Apache Kafka takes a partition-centric storage approach to message storage. In order to ensure data durability, multiple replicas of each partition are maintained within cluster to provide a configurable level of data redundancy.

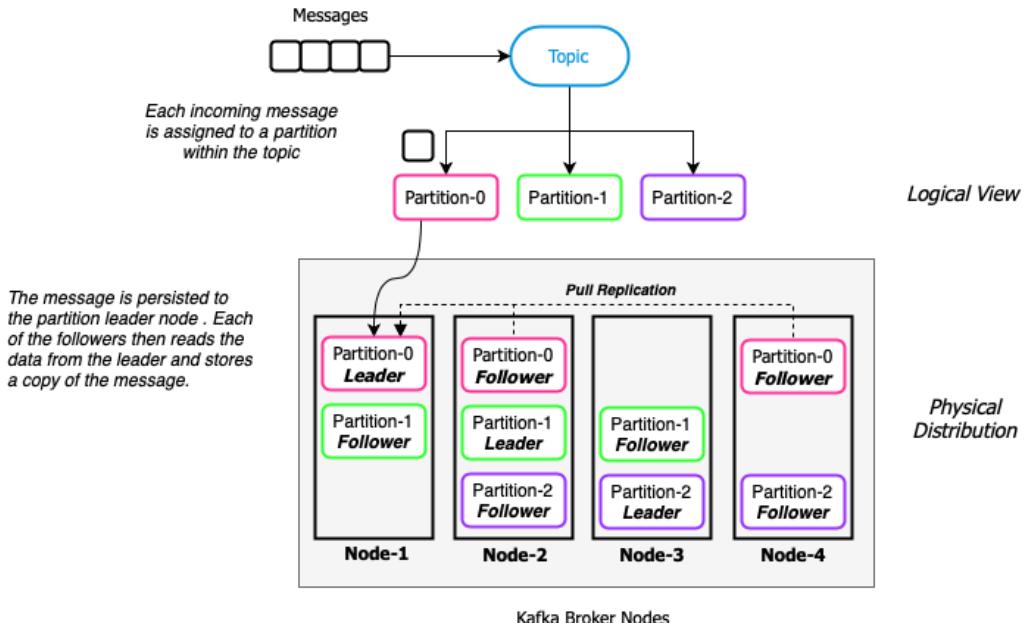


Figure 1.18: Kafka's Partition Replication Mechanism

When Kafka receives an incoming message, a hashing function is applied to message to determine which of topic's partitions the message should be written to. Once that has been determined, the message contents are written to the page cache (not the disk) of the partition "leader" replica. Once the message has been acknowledged by the leader, each of the "follower" replicas are responsible for retrieving the message contents from the partition leader as show in Figure 1.18 in a pull manner, i.e., they act as consumers and read the messages from the leader. This overall approach is what is referred to as an eventually consistent strategy, in which there is one node in a distributed system that has the most recent view of the data which is eventually communicated to other nodes until they all achieve a consistent view of the data. While this approach has the advantage of decreasing the amount of time required to store an incoming message, it also introduces two opportunities for data loss; first, in the event of a power outage or other process termination event on the leader node any data that was written to the page cache that had not been persisted to local disk will be lost. The second opportunity for data loss is when the current leader process fails and another one of the remaining followers is selected as the new leader. In the leader fail-over scenario, any messages that were acknowledged by the previous leader but not yet replicated to newly elected leader replica will be lost as well.

Kafka does provide the ability to configure this default behavior where only the leader has a copy before acknowledgement is sent, to withhold an acknowledgement until ALL replicas have received the message. However, this does not impact the underlying replication mechanism in which the followers must pull the information across the network and send a

response back to the leader. Obviously, this behavior will incur a performance penalty which is often hidden in most published Kafka performance benchmarks so you are advised to do your own performance testing with this configuration in order to get a better understanding of what the expected performance will be.

The other side-effect of this replication strategy is that only the leader replica can serve both producers and consumers as it is the only one guaranteed to have the most recent and correct copy of the data. All of the follower replicas are passive nodes that cannot alleviate any of the load from the leader during traffic spikes.

DATA DURABILITY IN PULSAR

When Pulsar receives an incoming message, it saves a copy in memory and also writes the data to a write-ahead-log (WAL), which is forced onto disk **before** an acknowledgement is sent back to the message publisher as shown in Figure 1.19. This approach is modelled after traditional database ACID transaction semantics, which ensures that the data is not lost even if the machine fails and comes back online in the future.

The number of replicas required for a topic can be configured in Pulsar, based on your data replication needs, and Pulsar guarantees that the data that has been received and acknowledged by a quorum of servers before an acknowledgement is sent to the producer. This design ensures that data can only be lost in the highly unlikely event of simultaneous fatal errors occurring on all bookie nodes to which the data was written. This is why it is recommended to distribute the bookie nodes across multiple regions and use rack-aware placement policies to ensure a copy of the data is stored in more than one region or data center.

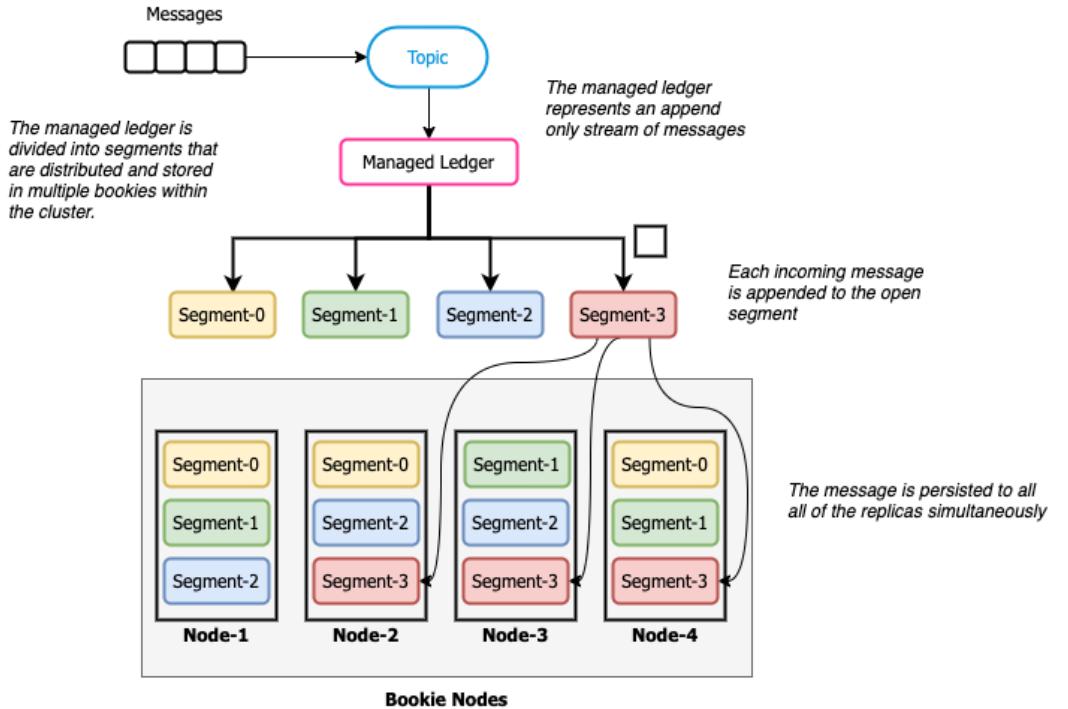


Figure 1.19: Pulsar's replication occurs when the message is initially written to the topic.

More importantly, this design eliminates the need for a secondary replication process that is responsible for ensuring that the data is kept in sync between replica and eliminates any data inconsistency issues due to any lag in the replication process.

1.4.4 Message Acknowledgement

Within a distributed messaging system, failures are to be expected. In a distributed message system such as Pulsar, both the consumers consuming the messages and the message brokers serving the messages can fail. When such a failure occurs, it is imperative to resume consumption from exactly the point where the consumers have left off once everything recovers to ensure that messages aren't skipped or reprocessed. This point from which the consumer should resume consumption is often called the *topic offset*. Kafka and Pulsar take different approaches with respect to maintaining these offsets, which has a direct impact on data durability.

MESSAGE ACKNOWLEDGEMENT IN KAFKA

The resume point is referred to as the *consumer offset* in Apache Kafka, which is controlled entirely by the consumer. Typically, a consumer increments its offset in a sequential manner as it reads records from the topic to indicate message acknowledgment. However, keeping

this offset solely in the consumers memory is dangerous. Therefore, these offsets are also stored as messages in a separate topic named '`__consumer_offsets`'. Each consumer commits a message containing its current position into that topic at periodic intervals, which is every five seconds if you use Kafka's auto commit capability. While this strategy is better than keeping the offsets solely in memory, there are consequences to this periodic update approach.

Consider a single-consumer scenario where automatic commits occur every five seconds, and that a consumer dies exactly three seconds after the most recent commit to the offset topic. In this case, the offset read from the topic will be three seconds old, so all the events that arrived in that three second window will be processed twice. While it is possible to configure the commit interval to a smaller value and reduce the window in which records will be duplicated, it is impossible to completely eliminate them.

The Kafka consumer API provides a method that enables committing the current offset at a point that makes sense to the application developer rather than based on a timer. Therefore, if you really wanted to eliminate duplicate messaging processing, you could use this API to commit the offset after every successfully consumed message. However, this pushes the burden of ensuring accurate recovery offsets onto the application developer and introduces additional latency to the message consumers who now have to commit each offset to a Kafka topic and await an acknowledgement.

MESSAGE ACKNOWLEDGEMENT IN PULSAR

Apache Pulsar maintains a ledger inside of Apache BookKeeper for each subscriber that is referred to as the cursor ledger for tracking message acknowledgments. When a consumer has read and processed a message, it sends an acknowledgement to the Pulsar Broker. Upon receipt of this acknowledgement, the Broker immediately updates the cursor ledger for that consumer's subscription. Since this information is stored on a ledger in BookKeeper, we know that it has been fsync-ed to disk and multiple copies exist across multiple bookie nodes. Keeping this information on disk ensures that the consumers will not receive the message again even if they crash and restart at a later point in time.

In Apache Pulsar, there are two ways that messages can be acknowledged, selectively or cumulatively. With cumulative acknowledgment, the consumer only needs to acknowledge the last message it receives. All the messages in the topic partition up to and including the given message id will be marked as acknowledged and will not be re-delivered to the consumer again. Cumulative acknowledgment is effectively same as offset update in Apache Kafka.

The differentiating feature of Apache Pulsar over Kafka is the ability of consumers to acknowledge messages individually, aka selective acknowledgement. This capability is critical in supporting multiple consumers per topic because it allows for message redelivery in the event of a single consumer failure.

Let's consider the single-consumer failure scenario again where the consumer individually acknowledges messages after it has successfully processed them. During the time leading up to the failure, the consumer was struggling to process some of the messages while successfully processing others. Figure 1.20 shows an example of where only 2 of the messages (4 and 7) were successfully processed and acknowledged.

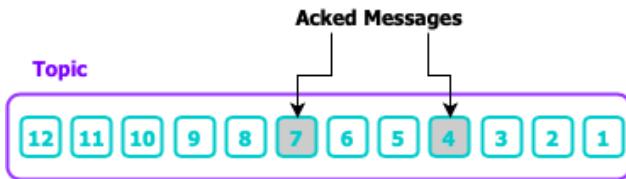


Figure 1.20: Individual message acknowledgment in Pulsar.

Given the fact that Kafka's offset concept treats consumer groups offsets as a high-water mark that marks the point up to which all messages are considered acknowledged. Therefore, in this scenario the offset would have been updated to 7 since that is the highest number message Id that was acknowledged. When the Kafka consumer is restarted on that topic it would start at message 8 and continue onward, skipping messages 1-3, 5, and 6, making them effectively "lost" because they are never processed.

Under the same scenario with Pulsar's selective acks, all of the unacknowledged messages would be redelivered, including messages 1-3, 5, and 6, when the consumer is restarted thereby avoiding message loss due to consumer offset limitations.

1.4.5 Message Retention

In contrast to legacy messaging systems such as ActiveMQ, etc., messages are not immediately removed from distributed messaging systems after they have been acknowledged by all consumers. These legacy systems took such an approach as a way to immediately reclaim as much of the local disk capacity as possible since it was a constrained resource. While distributed messaging systems such as Kafka and Pulsar have alleviated this constraint to some degree by horizontally scalable message storage, both of these systems still provide a mechanism for reclaiming disk space. It is important to understand exactly how automated message deletion is handled by both systems, as it can lead to accidental data loss, if not properly configured.

MESSAGE RETENTION IN KAFKA

Kafka retains all messages published to a topic for configurable retention period. For instance, if the retention policy is set to seven days, then for the seven days immediately after a message has been published to the topic it is available for consumption. Once the retention period has elapsed the message will be discarded to free up space. This deletion is regardless of whether or not the message has been consumed and acknowledged. Obviously, this presents the opportunity for data loss in the event that the retention period is less than the time it takes for all consumers to consume the message, such as a long-term outage of the consuming system.

The other drawback to this time-based approach is that there is a high probability that you will be retaining messages much longer than necessary, i.e. after they have been consumed by all relevant consumers, which is an inefficient use of your storage capacity.

MESSAGE RETENTION IN PULSAR

In Pulsar, when a consumer has successfully processed a message, it needs to send an acknowledgement to the broker so that the broker can discard the message. By default, Pulsar immediately deletes all messages that have been acknowledged by all the topic's consumers and retains all unacknowledged messages in a message backlog. In Pulsar, messages can only be deleted after all the subscriptions have already consumed it. Pulsar also allows you to keep messages for a longer time even after all subscriptions have already consumed them, by configuring a message retention period, which I will discuss in more depth in Chapter 2.

1.5 Why Do I Need Pulsar?

If you are just getting started with messaging or streaming data applications, you should definitely consider Apache Pulsar as a core component of your messaging infrastructure. However, it is worth noting that there are several technology options that you can choose from, several of which have become entrenched in the software community. In this section, I will attempt to bring to light some of the scenarios in which Apache Pulsar shines above the rest and clear up some common misconceptions about existing systems and point out some of the challenges users of these systems face.

Within adoption cycles there are often several misconceptions about the entrenched technology that are often perpetuated throughout the user community for a multitude of reasons. It is often an uphill battle to convince yourself and others that you need to replace a technology that sits at the very core of your architecture. It is not until we have the benefit of hindsight that we see that our traditional database systems were fundamentally incapable of scaling to meet the demands imposed by our ever-increasing data and that we needed to rethink the way we store and process data with a framework such as Hadoop. Only after we had transitioned our business analytics platforms from traditional data warehouses to Hadoop-based SQL engines such as Hive, Tez, and Impala, did we realize that those tools had inadequate response times for the end users who were used to sub-second response time. This gave rise to the rapid adoption of Apache Spark as the technology of choice for big data processing.

I wanted to highlight these two recent technologies to remind us that we cannot let our affinity for the status quo blind us from issues lurking within our core architectural systems, at put forth the notion that we need to rethink our approach to messaging systems, as the incumbent technologies in this space, such as RabbitMQ and Kafka, suffer from key architectural flaws. The team that developed Apache Pulsar at Yahoo could have easily chosen to adopt one of the existing solutions, but after careful consideration decided not to do so because they needed a messaging platform that provided the following capabilities that weren't available in the existing monolithic technologies:

1.5.1 Guaranteed Message Delivery

Because of the data durability mechanism within the platform that we have already covered, Pulsar provides guaranteed message delivery for applications. If a message successfully reaches a Pulsar broker, it will be delivered to all of the topic consumers. In order to provide

such a guarantee requires that non-acknowledged messages are stored in a durable manner until they can be delivered to and acknowledged by consumers. This mode of messaging is commonly called persistent messaging. In Pulsar, a configurable number of copies of all messages are stored and synced on disk.

By default, Pulsar message brokers ensure that incoming messages are persisted to disk on the storage layer before acknowledge receipt of the message. These messages are kept in Pulsar's infinitely scalable storage layer until they are acknowledged, thereby ensuring message delivery.

1.5.2 Infinite Scalability

In order to better understand the scalability of Pulsar, let's look at a typical Pulsar installation. As you can see from Figure 1.21, a Pulsar cluster is composed of two layers: a stateless serving layer, which is comprised of a set of brokers for handling client requests; and a stateful persistence layer, comprised of a set of bookies for persisting the messages.

This architectural pattern, which separates the storage of the messages from the layer that serves the messages differs significantly from traditional messaging systems which have historically chosen to co-locate these two services. This de-coupled approach has several advantages when it comes to scalability. For starters, making the brokers "stateless" allows you to dynamically increase or decrease the number of brokers to meet the demands of the client applications.

SEAMLESS CLUSTER EXPANSION

Any bookies that are added to the storage layer are automatically discovered by the brokers, that will then immediately begin to utilize them for message storage. Unlike Kafka, which requires re-partitioning the topics to distribute the incoming messages to the newly added brokers.

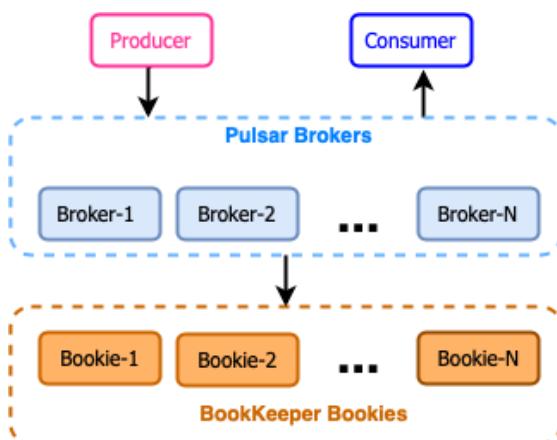


Figure 1.21: A Typical Pulsar Cluster

UNBOUNDED TOPIC PARTITION STORAGE

Unlike Kafka, the capacity of a topic partition is not limited by the capacity of any smallest node. Instead, topic partitions can scale up to the total capacity of the storage layer, which itself can be scaled up by simply adding additional bookies. As we discussed earlier, partitions within Kafka have several limitations on their size, whereas no such restrictions apply to Pulsar.

INSTANT SCALING WITHOUT DATA REBALANCING

Because message serving and storage are separated into two layers, moving a topic partition from one broker to another can happen almost instantly and without any data rebalancing (recopying the data from one node to the other). This characteristic is crucial to many things, such as cluster expansion and fast failure reaction to broker and bookie failures.

1.5.3 Resilient to Failure

Pulsar's de-coupled architecture also provided enhance resiliency by ensuring that there is no single point of failure within the system. By isolating the serving and storage layers, Pulsar is able to limit the impact of a failure within the system while making the recovery process seamless.

SEAMLESS BROKER FAILURE RECOVERY

Brokers form the stateless serving layer in Apache Pulsar. The serving layer is "stateless" because brokers don't actually store any message data locally. This makes Pulsar to resilient to broker failures. When Pulsar detects that a broker is down it can immediately transfer the incoming producers and consumers to a different broker. Since the data is kept in a separate layer there is no need to recopy data as you would in Kafka. Because Pulsar doesn't have to recopy the data, the recovery happens instantly without sacrificing the availability of any of the data on the topic.

Kafka, in contrast, directs all client requests to the "leader" replica, so it will always have the latest data. The leader is also responsible for propagating the incoming data to the other "followers" in the replica set, so that the data will eventually be available on those nodes in the event of a failure. However, due to the inherent lag between the leader and the replica, data can be lost before it is copied over.

SEAMLESS BOOKIE FAILURE RECOVERY

The stateful persistence layer utilized by Pulsar consists of Apache BookKeeper bookies to provide segment-centric storage as we mentioned previously. When a message is published to Pulsar the data is persisted to disk on all N replicas before it is acknowledged. This design ensures that the data will be available on multiple nodes, and thus will survive N-1 node failures before the data is lost.

Pulsar's storage layer is also self-healing, and if there is a node or disk failure that causes a particular segment to be under-replicated, Apache BookKeeper will automatically detect this and schedule a replica repair to run in the background. The replica repair in Apache BookKeeper is a many-to-many fast repair at the segment level, which is a much finer granularity than recopying the whole topic partition which is required in Kafka.

1.5.4 Support for Millions of Topics

Consider a scenario in which you want to model your application around some entity such as a customer, and for each one of these you wanted to have a different topic. Different events would be published for that entity; the customer is created, places an order, makes a payment, returns some items, changes their address, etc.

By placing these events in a single topic, you are guaranteed to process them in the correct chronological order, and can quickly scan the topic to determine the correct state of the customer's account, etc. However, as your business grows you will need support for millions of topics, and traditional messaging systems cannot support this requirement. There is a high cost associated with having many topics including increased end-to-end latency, file descriptors, memory overhead, and recovery time after a failure.

In Kafka, as a rule of thumb you should keep your total number of topic-partitions in the hundreds if you care about latency performance. There are several guides for how to restructure your Kafka based applications in order to avoid hitting this limitation. If you don't want a platform limitation affecting your application design with respect to how you structure your topics, then you should consider Pulsar.

Pulsar has the ability to support up to 2.8 million topics while continuing to provide consistent performance. The key to scaling the number of topics lies in how the underlying data is organized in the storage layer. If the topic data is stored in dedicated files or directories as it is in traditional messaging systems such as Kafka, then the ability to scale will be limited because the I/O will be scattered across the disk as the number of topics increases which leads to disk thrashing and results in very low throughput. In order to prevent this behavior, messages from different topics are aggregated, sorted, and stored in large files and then indexed in Apache Pulsar. This approach limits the proliferation of small files that leads to performance problems as the number of topics increases.

1.5.5 Geo-Replication and Active Failover

Apache Pulsar is a messaging system that supports both synchronous geo-replication within a single Pulsar cluster and asynchronous geo-replication across multiple clusters. It has been deployed globally in more than 10 data centers at Yahoo since 2015 with full 10x10 mesh replication for mission critical services such as Yahoo Mail and Finance.

Geo-replication is a common practice used to provide disaster recovery capabilities for enterprise systems by distributing a copy of the data to different geographical locations. This ensures that your data, and the systems that rely upon it, will be able to withstand any unforeseen disasters such as natural disasters. The geo-replication mechanisms used in different data systems can classified as either, synchronous or asynchronous. Apache Pulsar allows you to easily enable asynchronous geo-replication using just a few configuration settings.

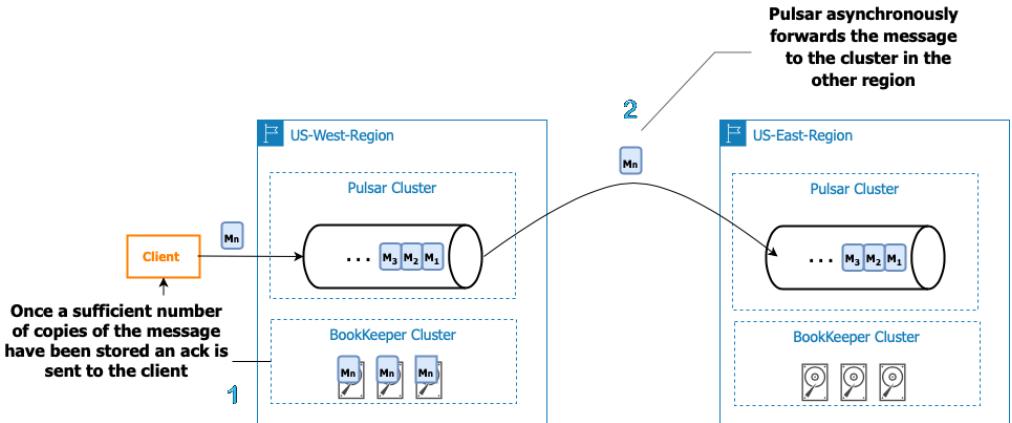


Figure 1.22: When using asynchronous geo-replication in Apache Pulsar, the message is stored locally within the BookKeeper cluster running in the same region that the receives the message. The message is asynchronously forwarded in the background to the Pulsar Cluster in the other region.

With asynchronous geo-replication the producer doesn't wait for an acknowledgement from the other data centers that they have received the message, instead the producing client receives an acknowledgement immediately after the message has been successfully persisted to the local BookKeeper cluster. The data is then replicated from the source cluster to the other data centers in an asynchronous fashion as shown in Figure 1.22.

Asynchronous geo-replication provides lower latency because the client doesn't have to wait for responses from the other data centers. However, it also results in weaker consistency guarantees, due to asynchronous replication. Since there is always a replication lag in asynchronous replication, there is always some amount of data that hasn't been replicated from source to destination.

Synchronous geo-replication is bit more complicated to achieve with Apache Pulsar than asynchronous as it requires some manual configuration to properly ensure that a message will only be acknowledged when a majority of the data centers have issued a confirmation that the message data has been persisted to disk. While I will save the details of exactly how synchronous geo-replication can be achieved with Apache Pulsar for Appendix B, I can tell you that it made possible due to pulsar's two-tiered architecture design and the ability for an Apache BookKeeper cluster to be composed of both local and remote nodes, particularly ones in different geographical regions as shown in Figure 1.23.

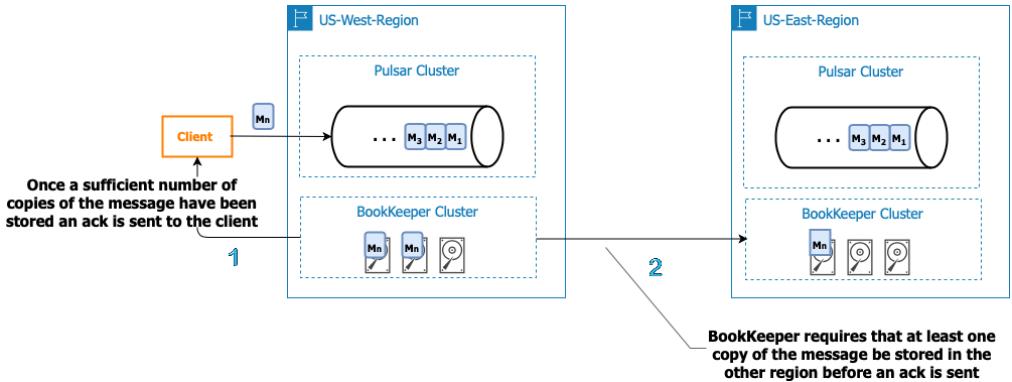


Figure 1.23: You can exploit BookKeeper's ability to use remote nodes in order to achieve synchronous geo-replication to ensure that a copy of the message is stored in the remote region

Synchronous geo-replication provides the highest availability, where all of your physically available data centers form a global logical instance for your data system. Your applications can run everywhere at any data center and still be able to access the data. It also guarantees stronger data consistency between different data centers, which your applications can easily rely on without any manual operations involved when data center failures occur.

Unlike other messaging systems that rely on external services, Pulsar offers geo-replication as a built-in feature. Users can easily enable replication of message data between Pulsar clusters in different geographical regions. Once replication is configured, data is continuously replicated to the remote clusters without any interaction on the part of the producers or consumers. I cover how to configure geo-replication in greater detail in Appendix B.

1.6 Real World Use Cases

If you are a product manager whose product includes a requirement for operating on massive amounts of data to deliver a meaningful new experience or dataset to your users in real-time, then Apache Pulsar is the key to unlocking the real time potential of your data. The beauty of Pulsar is that it has several specific scenarios in which it can excel. Before we dive further into the technical details, it might be informative to discuss at a high level some of the use cases that Pulsar has already been proven.

1.6.1 Unified Messaging System

You are probably familiar with the mnemonic "keep it simple, stupid", which is often used to remind architects that there is great value in simple designs and solutions. A system that is comprised of fewer technologies is easier to deploy, maintain, and monitor. As mentioned earlier, there are two common messaging patterns and until now if you wanted to support both messaging styles within your infrastructure, you were required to deploy and maintain two completely different systems.

As a bilingual job board, Zhaopin.com has one of the largest selections of job vacancies in China, including both prominent local and foreign companies. The company has over 2.2 million clients, and average daily page views of over 68M. As the company grew and the challenges of maintaining two separate messaging systems; Rabbit MQ for queuing, and Apache Kafka for pub-sub became increasingly difficult. By replacing them both with a single unified messaging platform based on Apache Pulsar they were able to reduce their operational overhead, infrastructure footprint, and on-going support costs in half while meeting their requirements of high durability, high throughput, and low latency.

1.6.2 Microservices Platform

Narvar provides a supply chain management and customer care platform for eCommerce customers across the world including order tracking and notifications, to seamless returns and customer care. Narvar's platform helps retailers and brands by processing data and events to ensure timely and accurate communication with their customers to 400 million consumers worldwide.

Prior to Apache Pulsar, Narvar's platform had been built using a variety of messaging and processing technologies over time, from Kafka to Amazon SQS, Kinesis Streams to Kinesis Firehose, RabbitMQ to AWS Lambda. As their traffic grew, it became apparent that the growing amount of DevOps and developer support required to maintain and scale these systems was unsustainable. Many of them were not containerized, making infrastructure configuration and management burdensome, and required frequent manual intervention.

Systems like Kafka — while reliable, popular and open source — had significant maintenance overhead as they scaled. Increasing throughput required increasing partitions, tuning consumers, and required a large amount of manual intervention by developers and DevOps. Similarly, cloud-native solutions like Kinesis Streams and Kinesis Firehose were not cloud-agnostic, making it hard to decouple the choice of cloud solutions from functionality and making it difficult to leverage technologies in other clouds, and to support customers who needed to run on other public clouds.

They decided to transition their microservice-based platform over Apache Pulsar because like Kafka, Pulsar was reliable, cloud-agnostic and open-sourced. Unlike Kafka, Pulsar entailed very little maintenance overhead and scaled with minimal manual intervention. Pulsar was containerized and built on Kubernetes from the outset making it much more scalable and maintainable.

Most importantly for Narvar was Pulsar Functions, which allowed them to develop microservices that consumed and process the incoming events directly on the messaging system itself, eliminating the need for expensive Lambda functions or standing up of additional services.

In a microservices architecture, each microservice is designed as an atomic and self-sufficient piece of software. These independent software components run as multiple processes distributed across multiple servers. A microservices-based application requires the interaction of multiple services via some sort of inter-process communication. The two most commonly used communication protocols are HTTP request/response, and lightweight messaging. Pulsar was perfect candidate for providing the lightweight messaging system that supports asynchronous messaging required by Narvar.

1.6.3 Connected Car

A major North American auto manufacturer has built a connected car service based on Apache Pulsar that collect billions of pieces of data from computing devices within its 12 million connected vehicles. Billions of pieces of data from are collected daily and used to provide real-time visibility and remote diagnostics across the world. This data is then used to provide better insights into how vehicles are performing and to identify potential problems before they occur so they can provide customers with proactive alerts.

1.6.4 Fraud Detection

As the fintech company for China Telecom, Orange Financial must analyze 50M transactions per day for financial fraud on behalf of its 500M registered users. Orange Financial faces threats from financial fraud every day, including identity theft, money laundering, affiliate fraud, merchant fraud, etc. The company runs thousands of fraud detection models against each transaction to combat these threats in its risk management system.

The company was seeking a solution that would unify the data store, computing engine, and programming language for decision development in its risk control system. From an end-user perspective, the fraud detection scanning could not impact the latency of the applications, therefore they needed a platform that allowed them to process the data as quickly as possible. Apache Pulsar allowed the transactional data to be accessed directly in the messaging layer and processed in parallel using Pulsar Functions thereby reducing the processing latency introduced from having to move the data to a secondary system for processing.

While some of the fraud detection processing has been offloaded to the Pulsar functions framework, they were still able to leverage their more complex fraud detection algorithms that were developed in Spark using Pulsar's built-in connector for the Spark computing engine. This allows them to choose the best processing framework for their models on a case-by-case basis.

1.7 Additional Resources

Pulsar has a vibrant and growing community and graduated from the Apache Incubator in August of 2018. Current documentation for the project can be found on the official project website at <http://pulsar.apache.org>.

Other resources for information on Apache Pulsar include blogs [here](#), and [here](#) & [Tutorials](#). Lastly, I would be remiss if I didn't mention the Apache Pulsar slack channel: apache-pulsar.slack.com, which I and several of the project committers monitor on a daily basis. The heavily used channel contains a wealth of information for beginners, and a concentrated community of developers who are actively using Apache Pulsar on a daily basis.

1.8 Summary

- Apache Pulsar is a modern messaging system that provides both high-performance streaming and traditional queuing messaging.
- Apache Pulsar provides a lightweight computing engine named "Pulsar Functions"

which allows developers to implement simple processing logic that is executed against each message that is published to a given topic.

- The benefits of Pulsar's decoupled storage and serving layers; including infinite scalability, zero data loss.
- Discussed specific use cases where Pulsar has been used in production, such as IoT analytics, inter-microservice communication, and unified messaging.

2

Pulsar Concepts and Architecture

This chapter covers

- Pulsar's physical architecture
- Pulsar's logical architecture
- Message consumption and the subscription types provided by Pulsar.
- Pulsar's message retention, expiration, and backlog policies.

Now that you have been introduced to the Pulsar messaging platform and how it compares to other messaging systems, we will drill down into the low-level architectural details of the platform and cover some of the unique terminology used by the platform. If you are unfamiliar with messaging systems and distributed systems, then it might be difficult to wrap your head around some of the Pulsar's concepts and terminology. Therefore, I will introduce them in a progressive fashion using a more concrete example.

2.1 Pulsar's Physical Architecture

As we mentioned, there will be a Pulsar cluster in each of the three different geographical regions including North America, Europe, and Asia for a total of three Pulsar clusters. Other messaging systems consider the cluster the highest level from an administrative and deployment perspective, which necessitates managing and configuring each cluster as an independent system. Fortunately, Pulsar provides an even higher level of abstraction known as a *Pulsar Instance*, which is comprised of one or more Pulsar clusters that act together as a single unit and can be administered from a single location, as shown in Figure 2.1.

One of the biggest reasons for using a Pulsar Instance, is to enable geo-replication. In fact, only clusters within the same instance can be configured to replicate data amongst themselves. Given that data availability is a requirement for ensuring 24x7x365 operation, we will require a Pulsar Instance to enable geo-replication within our architecture.

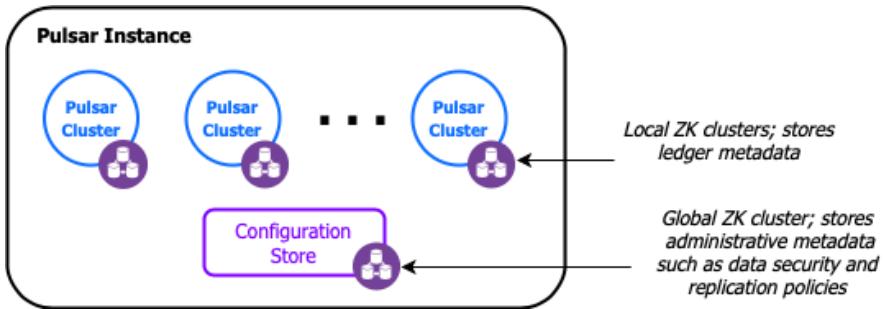


Figure 2.1: A Pulsar Instance can consist of multiple, geo-graphically dispersed clusters.

A Pulsar Instance employs an instance-wide Zookeeper cluster called the *configuration store* to retain information that pertains to multiple clusters, such as geo-replication, and tenant-level security policies. This allows you to define and manage these policies in a single location. In order to provide resiliency to the configuration store, each of the nodes within the Pulsar Instance's ZK ensemble should be deployed across multiple regions to ensure its availability in the event of a region failure, etc.

It is important to note that the availability of the Zookeeper ensemble used by the Pulsar Instance for the configuration store is required by the individual Pulsar clusters to operate even when geo-replication is enabled. When geo-replication is enabled, if the configuration store is down messages published to the respective clusters will be buffered locally and forwarded to the other regions when the ensemble becomes operational again.

2.1.1 Pulsar's Layered Architecture

Within each of the geographic regions there will be Pulsar cluster that is comprised of a stateless serving layer based on one or more Pulsar message *brokers*; and a stateful storage layer based on one or more BookKeeper *bookies* as shown in Figure 2.2. When hosted inside a Kubernetes environment, this decoupled architecture enables our DevOps team to dynamically scale the number of brokers and bookies to meet the peak demand such as during dinner time and to scale down to save cost during slower periods. Message traffic is spread across all the available Brokers as evenly as possible to provide maximum throughput.

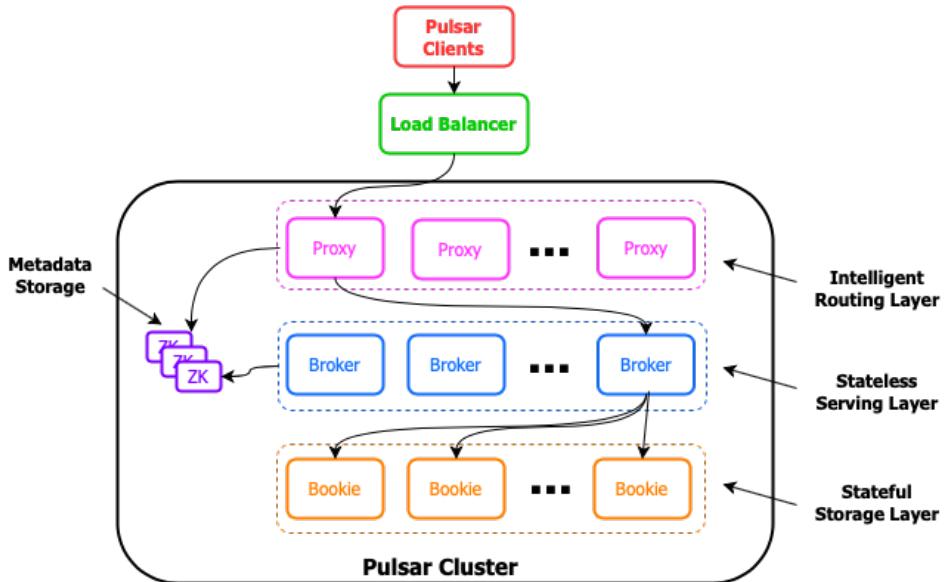


Figure 2.2: A Pulsar cluster consists of multiple layers; an optional proxy layer that routes incoming client requests to the appropriate message Broker, a stateless serving layer consisting of multiple Brokers that serve client requests, and a stateful storage layer consisting of multiple Bookies that retain multiple copies of the messages.

When a client accesses a topic that has not yet been used, a process is triggered to select the Broker best suited to acquire “ownership” of the topic. Once a Broker assumes ownership of a topic, it is responsible for handling all requests for that topic, and any clients wishing to publish to or consume data from the topic need to interact with the corresponding Broker that “owns” it. Therefore, if you want publish data to a particular topic, you will need to know which Broker owns that topic and connect to it. However, the broker assignment information is only available in the Zookeeper metadata and is subject to change based on load rebalancing, Broker crashes, etc. Consequently, you cannot connect directly to the Brokers themselves and hope that you are communicating with the one you want. This is exactly why the Pulsar Proxy was created, to act as an intermediary for all the brokers in the cluster.

THE PULSAR PROXY

If you are hosting your Pulsar cluster inside of a private and/or virtual network environment such as Kubernetes and you want to provide inbound connections to your Pulsar Brokers, then you will need to translate their private IP addresses to public IP addresses. While this can be accomplished using traditional load balancing technologies and techniques such as physical load balancers, virtual IP addresses, or DNS-based load balancing that distributes client requests across a group of brokers it is not the best approach for providing redundancy and fail-over capabilities for your clients.

The traditional load-balancer approach is not efficient as the load-balancer will not know which broker is assigned to a given topic and instead will direct the request to a random broker in the cluster. If a broker receives a request for a topic it isn't serving, it will automatically re-route the request over to the appropriate broker for processing, but this incurs the non-trivial penalty in terms of time. Which is why it is recommended to use the Pulsar Proxy instead which acts as intelligent load balancer for Pulsar Brokers.

When using the Pulsar Proxy all client connections will first travel through the proxy rather than directly to the brokers themselves. The proxy will then use Pulsar's built-in service discovery mechanism to determine which broker is hosting the topic you are trying to reach and automatically route the client request to it. Furthermore, it will cache this information in memory for future requests to streamline the lookup process even more. For performance and failover purposes, it is recommended to run more than one Pulsar Proxy behind a traditional load balancer. Unlike the Brokers, Pulsar Proxies **can** handle any request, so they can be load balanced without any issue.

2.1.2 Stateless Serving Layer

Pulsar's multi-layered design ensures that message data is stored separately from the Brokers, which guarantees that any broker can serve data from any topic at any time. This also allows the cluster to assign ownership of a topic to any broker in the cluster at any time, unlike other messaging systems that co-locate the broker and the topic data they are serving. Hence, we use the term "stateless" to describe the serving layer, since there is no information stored on the brokers themselves that is necessary to handle client requests.

The "stateless" nature of the Brokers not only allows us to dynamically scale them up and down based on demand, but also makes the cluster resilient to multiple broker failures as well. Lastly, Pulsar has an internal load-balancing mechanism to continuously rebalance the load amongst all the active Brokers based on the ever-changing message traffic.

BUNDLES

The assignment of a topic to a particular Broker is done at what is referred to as the *bundle* level. All the topics in a Pulsar cluster assigned to a specific bundle, with each bundle being assigned to a different broker as shown in Figure 2.3. This helps ensure that all topics in a are evenly distributed across all the brokers.

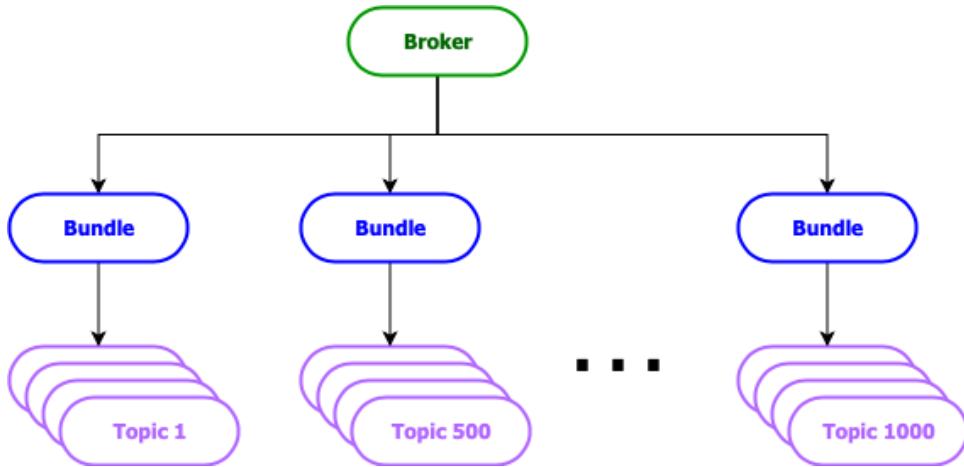


Figure 2.3: From a serving perspective, each broker is assigned a set of bundles that contain multiple topics. Bundle assignment is determined by hashing the topic name, which allows us to determine which bundle it belongs without having to keep that information in Zookeeper.

The number of bundles created per namespace is controlled by the `defaultNumberOfNamespaceBundles` property inside the broker configuration file, which has a default value of 4. You can override this setting on a per namespace level when you create the namespace by providing a different value when you create the namespace using the Pulsar admin API. In general, you want the number of bundles to be a multiple of the number of brokers to ensure that they are evenly distributed. For instance, if you have 3 Brokers and 4 bundles per namespace, then one of the Brokers will be assigned two of the bundles while the others only get one.

LOAD BALANCING

While the message traffic might be spread across as evenly as possible across the active Brokers initially, several factors can change over time that result in the load becoming unbalanced. Changes in the message traffic patterns might result in a Broker serving several topics with heavy traffic while others aren't being utilized at all. When an existing bundle exceeds some preconfigured thresholds defined by the following properties in the broker configuration file, then the bundle will be split into two new bundles, with one of them being offloaded to a new Broker:

- `loadBalancerNamespaceBundleMaxTopics`
- `loadBalancerNamespaceBundleMaxSessions`
- `loadBalancerNamespaceBundleMaxMsgRate`
- `loadBalancerNamespaceBundleMaxBandwidthMbytes`

This mechanism identifies and corrects scenarios when some bundles are experiencing heavier load than others by splitting these overloaded bundles in two. Then one of these bundles can be offloaded to a different Broker in the cluster.

LOAD SHEDDING

The Pulsar Brokers have another mechanism to detect when a particular Broker is overloaded, and automatically have it shed or offload some of its bundles to other Brokers in the cluster. When a Broker's resource utilization exceeds the preconfigured threshold defined by the `loadBalancerBrokerOverloadedThresholdPercentage` property in the broker configuration file, the Broker will offload one or more bundles to a new Broker. This property defines the maximum percentage of the total available CPU, network capacity, or memory that the Broker can consume. If any of these resources cross this threshold then the offload is triggered.

The bundle selected is left intact and just assigned to a different Broker. This is because the load shedding process solves a different problem than the load balancing process does. With load balancing, we are correcting the distribution of the topics across the bundles because one of them has much more traffic than the others, and we are attempting to spread that load out across all the bundles.

Load shedding on the other hand, corrects the distribution of the bundles across the Brokers based on the number of resources required to service them. Even though each Broker can be assigned the same number of bundles, the message traffic handled by each Broker could be dramatically different if the load is unbalanced across the bundles.

To illustrate this point, consider the scenario where there are 3 Brokers and a total of 60 bundles, with each Broker serving 20 bundles each. Furthermore, 20 of the bundles are currently handling 90% of the total message traffic. Now if most of these bundles happen to be assigned to the same Broker, it could easily exhaust that Broker's CPU, Network, and memory resources. Therefore, offloading some of these bundles to another Broker will help alleviate the problem whereas splitting the bundles themselves would only shed approximately half of the message traffic while leaving 45% of it still on the original Broker.

DATA ACCESS PATTERNS

There are generally three I/O patterns in a streaming system; *Writes*, where new data is written to the system, *Tailing Reads*, where the consumer is reading the most recently published messages immediately after they have been published, and *Catch-up Reads*, where a consumer reads a large number of messages from the beginning of the topic in order to catch up, such as when a new consumer wants to access data beginning at a point much earlier than the latest message.

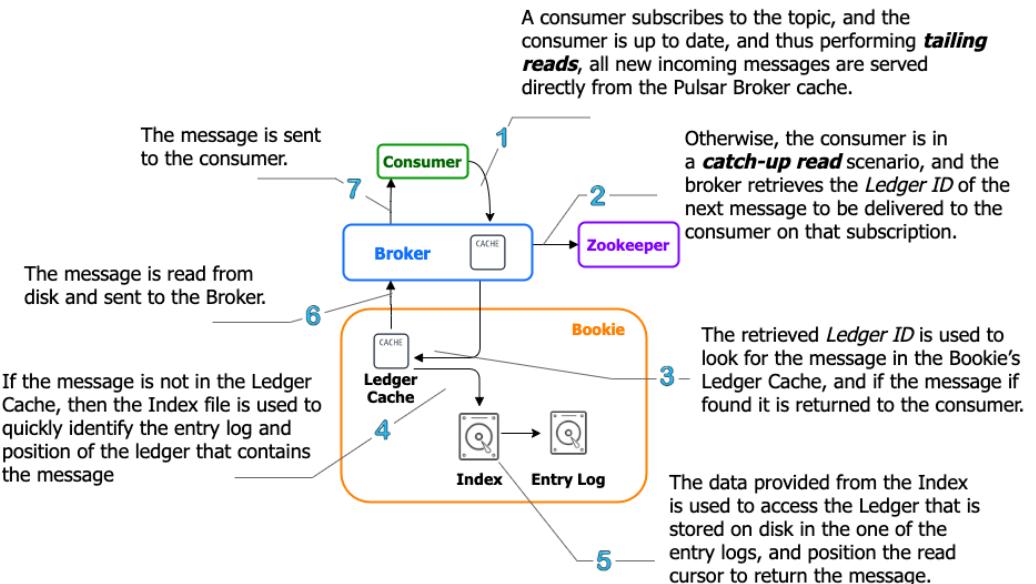


Figure 2.4: Message consumption steps in Pulsar.

When a producer sends a message to Pulsar, it is immediately written to BookKeeper. Once BookKeeper acknowledges that the data was committed, the broker stores a copy of the message in its local cache before it acknowledges the message publication to the producer. This allows the broker to serve “Tailing Read” consumers directly from memory and avoid the latency associated with disk access.

It becomes more interesting when looking at “Catch-up Reads” which access data from the storage layer. When a client consumes a message from Pulsar the message will go through the steps as shown in Figure 2.4. The most common example of Catch-up Reads is when a consumer goes offline for an extended period and then starts consuming again, although any scenario in which a consumer is NOT directly served from the Broker’s in-memory cache would be considered a Catch-up Read, such as topic re-assignment to a new Broker.

2.1.3 Stream Storage Layer

Pulsar guarantees message delivery for all message consumers. If a message successfully reaches a Pulsar broker, you can rest assured it will be delivered to its intended target. In order to provide this guarantee, all non-acknowledged messages must be persisted until they can be delivered to and acknowledged by consumers. As I have mentioned earlier, Pulsar uses a distributed write-ahead log (WAL) system called Apache BookKeeper for persistent message storage. BookKeeper is a service that provides persistent storage of log entries in sequences called ledgers.

LOGICAL STORAGE ARCHITECTURE

Pulsar topics can be thought of as infinite streams of messages that are stored sequentially in the order the messages are received. Incoming messages are appended to the end of the stream while consumers read messages further up the stream based on the data access patterns that I discussed earlier. While this simplified view makes it easy for us to reason about a consumer's position within the topic, such an abstraction cannot exist in reality due to the space limitations of storage devices. Eventually this abstract infinite stream concept must be implemented on a physical system where such limitation exists.

Apache Pulsar takes a dramatically different approach than traditional messages systems, such as Kafka when it comes to implementing stream storage. Within Kafka, each stream is separated into multiple replicas that each are stored entirely on a Broker's local disk. The great thing about this approach is that it is simple and fast because all writes are sequential, which limits the amount of disk head movement required to access the data. The downside to Kafka's approach is that a single Broker must have sufficient storage capacity to hold the partition data, as I discussed in Chapter 1.

So how is Apache Pulsar's approach different? For starters, each topic is NOT modelled as a collection of partitions but rather as a series of segments. Each of these segments can contain a fixed number of messages, with the default being 50,000. Once a segment is full, a new one is created to hold new messages. Therefore, a Pulsar topic can be thought of as an unbounded list of segments, each containing a subset of the messages as shown in Figure 2.5 which shows both the logical architecture of the Stream Storage layer and how it maps to the underlying physical implementation.

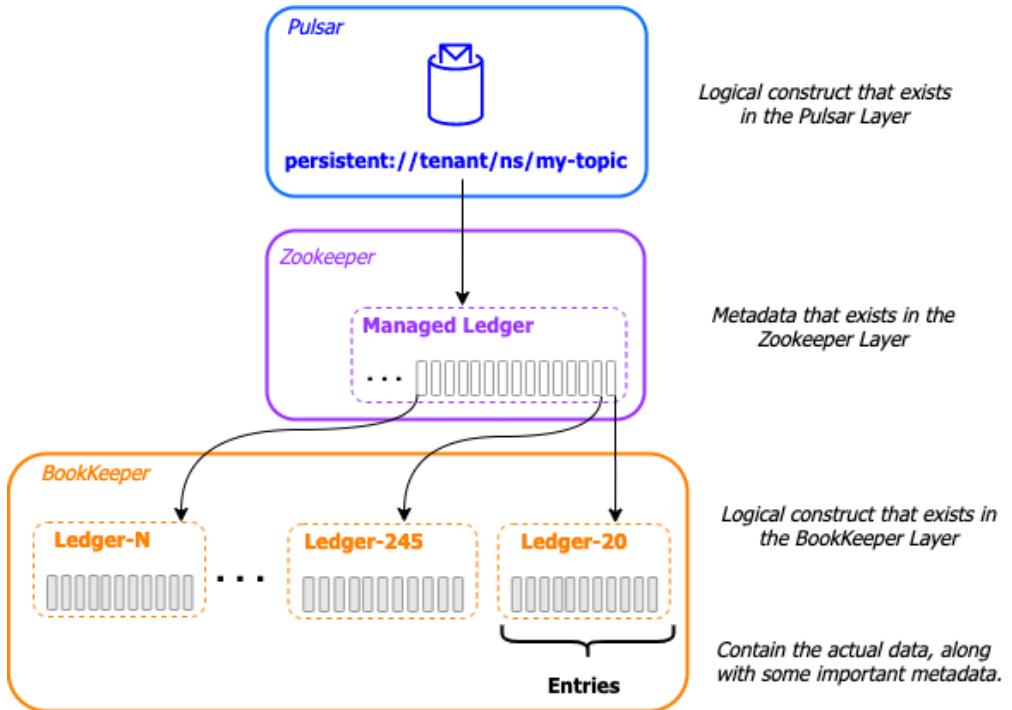


Figure 2.5: The data for a Pulsar topic is stored as a sequence of ledgers inside the BookKeeper layer. A list of these ledger IDs is stored inside a logical construct known as a managed ledger on the Zookeeper. Each ledger holds 50,000 entries that contain that store a replica of the physical data across multiple Bookies for redundancy.

A Pulsar topic is nothing more than an addressable endpoint that is used to uniquely identify a specific topic within Pulsar and is analogous to a URL in the sense that it is merely used to uniquely identify the resource that the client is attempting to connect to. The topic name must be decoded by the Pulsar Broker to determine the storage location of the data.

Pulsar adds an additional layer of abstraction on top of BookKeeper's ledgers known as managed ledgers that retains the IDs of the ledgers that hold the data published to the topic. As we can see in Figure 2.5, when data was first published to topic A, it was written to Ledger-20. After 50k records had been published to the topic the ledger was closed and another one (ledger-245) was created to take its place. This process is repeated every 50K records to store the incoming data, and the managed ledger retains this unique sequence of the ledger ids inside of Zookeeper.

Later when a consumer attempts to read the data from Topic A, the managed ledger is used to locate the data inside of BookKeeper and return it to the consumer. If the consumer is performing a catch-up read starting at the oldest message, then it would first get all the data from ledger-20, followed by ledger-245, etc. The traversal of these ledgers from oldest to youngest is transparent to the end user and create the illusion of a single sequential

stream of data. Managed ledgers allow this to happen and retain the ordering of the BookKeeper ledgers to ensure the messages are read in the same order as they were published.

BOOKKEEPER PHYSICAL ARCHITECTURE

In BookKeeper, each unit of a ledger is referred to as an entry. These entries contain the actual raw bytes from the incoming messages along with some important metadata that is used to track and access the entries. The most critical piece of metadata is the ID of the ledger to which it belongs, which is kept in the local Zookeeper instance so that the message can be retrieved quickly from BookKeeper when a consumer attempts to read the message in the future. Streams of log entries are stored in append-only data structures known as ledgers, as shown in Figure 2.6.

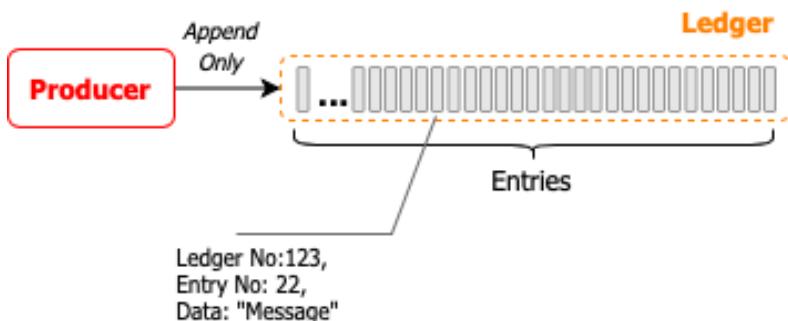


Figure 2.6: In BookKeeper, incoming entries get stored together as *ledgers* on servers known as *bookies*.

Ledgers have append-only semantics meaning that entries are written to a ledger sequentially and cannot be modified once they've been written to a ledger. From a practical perspective, this means:

- A Pulsar broker first creates a ledger, then append entries to the ledger, and finally closes the ledger. There are no other interactions permitted.
- After the ledger has been closed, either normally or because the process crashed, it can then be opened only in read-only mode.
- Finally, when entries in the ledger are no longer needed, the whole ledger can be deleted from the system.

The individual BookKeeper servers that are responsible for the storage of ledgers (more specifically, fragments of ledgers) are known as Bookies. Whenever entries are written to a ledger, those entries are written across a sub-group of bookies nodes known as an ensemble. The size of the ensemble is equal to the replication factor (R) you specify for your Pulsar topic and ensures that you have exactly R copies of the entry saved to disk to prevent data loss.

Bookies manage data in a log-structured way, which is implemented using three types of files: journals, entry logs, and index file. The journal file retains all of BookKeeper

transaction logs. Before any update to a ledger takes place, the bookie ensures that a transaction describing the update is written to disk to prevent data loss.

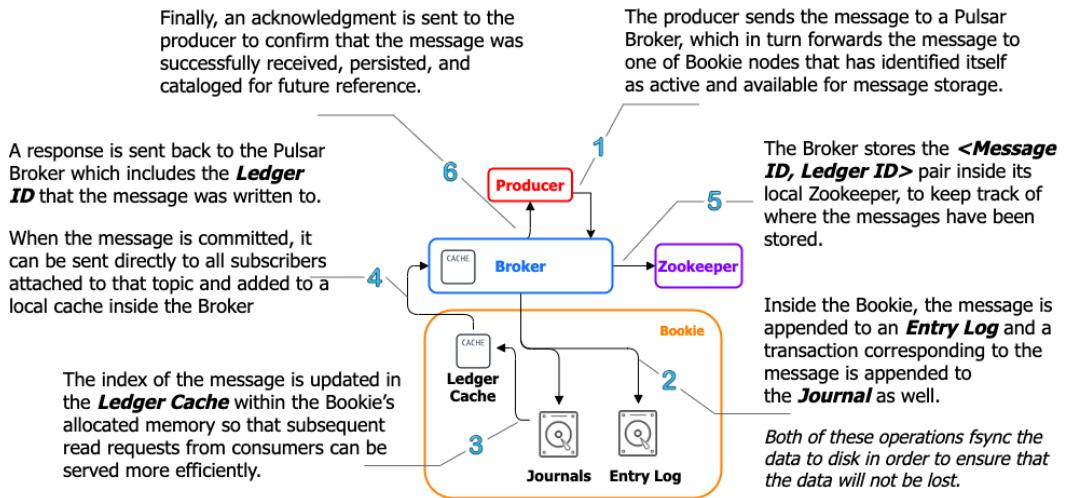


Figure 2.7: Message persistence steps in Pulsar.

The Entry log file contains the actual data written to BookKeeper. Entries from different ledgers are aggregated and written sequentially, while their offsets are kept as pointers in a ledger cache for fast lookup. An index file is created for each ledger, which contains several indexes that record the offsets of data stored in entry log files. The index file is modelled after index files in traditional relational databases and allows for quick lookups for ledger consumers. When a client publishes a message to Pulsar the message will go through the following steps as shown in Figure 2.7 to persist it to disk within a BookKeeper ledger.

By distributing the entry data across multiple files on different disk devices, bookies are able to isolate the effects of read operations from the latency of ongoing write operations allowing them to handle thousands of concurrent reads and writes.

2.1.4 Metadata Storage

Lastly, each cluster also has its own local Zookeeper ensemble that Pulsar uses to store cluster-specific configuration information for tenants, namespaces, and topics, including security and data retention policies, etc. This is in addition to the managed ledger information that we discussed earlier.

ZOOKEEPER BASICS

According to the official Apache website, "ZooKeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services.", which is an elaborate way of saying a distributed data source. Zookeeper

provides a de-centralized location for storing information, which is crucial within distributed systems such as Pulsar or BookKeeper.

Apache Zookeeper solves the fundamental problem of achieving consensus (a.k.a. agreement) that virtually every distributed system must solve. Processes in a distributed system need to agree on several different pieces of information such as the current configuration values, on the owner of a topic, etc. This is particularly a problem for distributed systems due to the fact that there are multiple copies of the same component running concurrently with no real way to coordinate information between them. Traditional databases are not an option because they introduce a serialization point within the framework where all the calling services are blocked waiting for the same lock on a table, which essentially eliminates all the benefits of distributed computing.

Having access to a consensus implementation enables distributed systems to coordinate processes in a more effective manner by providing a compare-and-swap (CAS) operation to implement distributed locks. The CAS operation compares the value retrieved from Zookeeper with an expected value and only if they are the same, updates the value. This guarantees that the system is acting based on up-to-date information. One such example would be checking that the state of a BookKeeper ledger is OPEN before writing any data to it. If some other process has closed the ledger, it would be reflected in the Zookeeper data, and the process would know not to proceed with the write operation. Conversely, if a process were to CLOSE a ledger this information would be sent to Zookeeper so that it could be propagated to the other services so that when they would know it is CLOSED before they attempt to write to it.

The Zookeeper service itself exposes a file-system-like API so that clients can manipulate simple data files (znodes) to store information. Each of these znodes form a hierarchical structure similar to a filesystem. In the following sections, I will examine the metadata that is retained within Zookeeper along with how it is used and by whom so that you can see for yourself exactly why it is needed. The best way to do this is by using the `zookeeper-shell` tool that is distributed along with pulsar as shown in Listing 2.1 to list all the znodes.

Listing 2.1 Using the `zookeeper-shell` tool to list the znodes

```
/pulsar/bin/pulsar zookeeper-shell    #A
ls /      #B
[admin, bookies, counters, ledgers, loadbalance, managed-ledgers, namespace, pulsar,
schemas, stream, zookeeper]    #C
```

#A Starts the Zookeeper shell

#B Lists the children znodes under the root level node.

#C The output of all the znodes used by Pulsar

As you can see from Listing 2.1, there are a total of eleven different znodes created inside Zookeeper for Apache Pulsar and BookKeeper, these fall into one of four categories based upon what information they contain and how it is used.

CONFIGURATION DATA

The first category of information is configuration data for tenants, namespaces, schemas, etc. All this information is slow-changing information that is only updated through the Pulsar administration API when a user creates or updates a new cluster, tenant, namespace, or

schema, and includes such things as security policies, message retention policies, replication policies, and schemas. This information is stored in the following znodes, /admin and /schemas.

METADATA STORAGE

The managed ledger information for all of the topics is stored in the /managed-ledgers znode while the /ledgers znode is used by BookKeeper to keep track of all the ledgers currently stored across all the bookies within the cluster.

Listing 2.2 Inspecting the Managed Ledger

```
/pulsar/bin/pulsar-managed-ledger-admin print-managed-ledger --managedLedgerPath  
/customers/orders/persistent/food-orders --zkServer localhost:2181 #A  
  
ledgerInfo { ledgerId: 20 entries: 50000 size: 3417764 timestamp: 1589590969679}  
ledgerInfo { ledgerId: 245 timestamp: 0} #B
```

#A The managed ledger tool allows you to lookup the ledgers by topic name

#B This topic has two ledgers, one with 50K entries that is closed, and another open one.

As you can see from Listing 2.2, there is another tool called that allows you to easily access the managed ledger information that is used by Pulsar to read & write the data from and to BookKeeper. In this case, the topic data is stored on two different ledgers; LedgerId-20 which is closed and contains 50,000 entries and LedgerId-245 which is currently open and where the incoming data will be published.

DYNAMIC COORDINATION BETWEEN SERVICES

The remaining znodes are all used for distributed coordination across the systems including /bookies which maintains a list of the Bookies registered with the BookKeeper cluster, and /namespace which is used by the proxy service to determine which broker “owns” a given topic. As we can see from Listing 2.3, the /namespace znode hierarchy is used to store the bundle ids for each namespace.

Listing 2.3 Metadata used to determine topic ownership

```
/pulsar/bin/pulsar zookeeper-shell #A  
ls /namespace  
[customers, public, pulsar] #B  
ls /namespace/customers  
[orders] #C  
ls /namespace/customers/orders  
[0x40000000_0x80000000] #D  
get /namespace/customers/orders/0x40000000_0x80000000  
{"nativeUrl": "pulsar://localhost:6650", "httpUrl": "http://localhost:8080", "disabled": false}  
#D
```

#A Starts the Zookeeper shell

#B There is one znode per tenant

#C There is one znode per namespace

#D There is one znode per bundle_id

As you recall from our discussion earlier, the topic name is hashed by the proxy to determine the bundle name, which in this case is 0x40000000_0x80000000. The Proxy then queries the /namespace/{tenant}/{namespace}/{bundle-id} znode to retrieve the URL for the Broker that "owns" the topic.

Hopefully this gives you some more insight into the role Zookeeper plays inside a Pulsar cluster and how it provides a service that can be easily accessed by nodes that have been dynamically added to the cluster so that they can quickly determine the cluster configuration and start handling client requests. One such example would be the ability of newly added Broker to start serving data from a topic by referencing the data in `/managed-ledgers` znode.

2.2 Pulsar's Logical Architecture

Our food delivery platform will have multiple services using the Pulsar messaging platform to send and receive information between one another. The ones we are concerned about for our mobile application include the driver location service which publishes the driver's location events that we use to update the customer on the status of their order while it is in-route. Since we are anticipating a large volume of customers during peak times, there could potentially be 10s of thousands of subscriptions to our driver location topic, one for each outstanding food order.

2.2.1 Tenants, Namespaces, and Topics

In this section we will cover the logical constructs that describe how data is structured and stored inside the cluster. Pulsar was designed to serve as a multi-tenant system, allowing it to be shared across multiple departments within your organization by providing each its own secure and exclusive messaging environment. This design enables a single Pulsar instance to effectively serve as the messaging platform-as-a-service across your entire enterprise. The logical architecture of Pulsar supports multi-tenancy via a hierarchy of tenants, namespaces, and finally topics as shown in Figure 2.8.

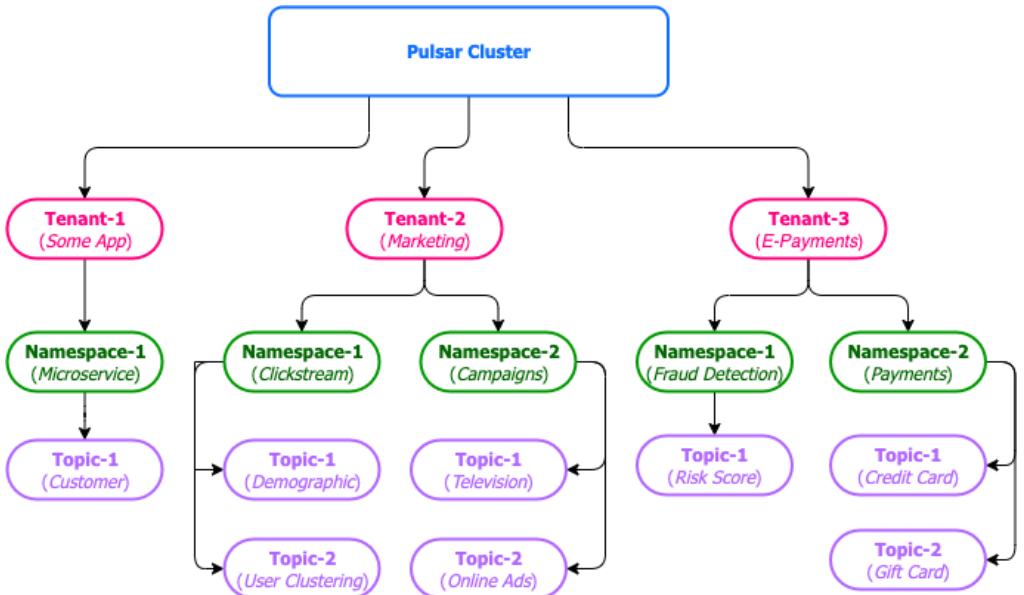


Figure 2.8: Pulsar's Logical Architecture consists of tenants, namespaces, and topics.

TENANTS

At the top of Pulsar hierarchy sits the tenants, which can represent a specific business unit, a core feature, or a product line. Tenants can be spread across clusters and can each have their own authentication and authorization scheme applied to them, thereby controlling who has access to the data stored within. They are also the administrative unit at which storage quotas, message time to live, and isolation policies can be managed.

NAMESPACES

Each tenant can have multiple *namespaces*, which is a logical grouping mechanism for administering related topics via policies. At the namespace level, you can set access permissions, fine-tune replication settings, manage geo-replication of message data across clusters, and control message expiry for all the topics in the namespace.

Let's consider how we would structure Pulsar's namespace for our food delivery application. To provide isolation for the sensitive incoming payment data, and limit access to only members of the finance team, you may configure a separate tenant named "E-Payments" as shown in Figure 2.1 and apply an access policy that restricts grants full access to only members of the finance group so they can perform audits and process the credit card transactions, etc.

Within the "E-Payments" tenant you might create two namespaces, one named "Payments" that will hold the incoming payments including credit card payments and gift card redemptions, and another named "Fraud Detection" which will contain those transactions that are flagged as suspicious for further processing. In such a deployment you

would limit the user facing application to write-only access to the “Payments” namespace while granting read-only access to the fraud detection application so it can evaluate them for potential fraud.

On the “Fraud Detection” namespace you would configure write access for the fraud detection application, so it can place potentially fraudulent payments into the “Risk Score” topic. You would also grant read-only access to the e-commerce application to the same namespace, so that it can be notified of any potential fraud and react accordingly, such as blocking the sale.

TOPICS

Topics are the only communication channel type available in Pulsar, all messages are written to and read from topics. Other messaging systems support more than one communication channel type, e.g., topics and queues that are differentiated by the type of message consumption they support. As I discussed in Chapter 1, Queues support first-in-first-out exclusive message consumption while Topics support pub-sub, one-to-many message consumption. Pulsar makes no such distinction, and instead relies on various subscription types to control the message consumption pattern.

Unless otherwise specified, topics inside Pulsar are served by a single broker which is responsible for receiving and delivering all the messages. Therefore, the throughput of a single topic is bound by the computing power of the broker serving it.

PARTITIONED TOPICS

Pulsar also supports the notion of partitioned topics that can be served by multiple brokers which allows for much higher throughput as the load is distributed across multiple machines. Behind the scenes, a partitioned topic is implemented as N internal topics, where N is the number of partitions. The distribution of partitions across brokers is handled automatically by Pulsar, effectively making the process transparent to the end user.

Implementing partitioned topics as a series of individual topics allows a user to increase the number of partitions without having to rebalance the entire topic. Instead, new internal topics will be created for the new partitions and will be available for receiving new messages immediately without impacting the other internal topics at all, e.g., consumers will still be able to read/write messages to the existing partitions without interruption.

From a consumer perspective these is no difference between partitioned topics and normal topics. All consumer subscriptions work exactly as they do on non-partitioned topics. There is a big difference in what happens when a message is published to a partitioned topic. The message producer is responsible for determining which internal topic the message is ultimately published to. If the message has a value in its key metadata field, then the producer will hash that value to determine which topic to publish to. This ensures that all messages with the same key get stored in the same topic and will be in the order in which they were published.

When publishing a message without a key the producer should be configured with a routing mode that specifies how to route messages across the partitions in the topic. The default routing mode is called `RoundRobinPartition`, which as the name implies publishes messages across all partitions in round-robin fashion. This approach evenly distributes the messages across the partitions, which maximizes the publish throughput. Alternatively, you

could use the `SinglePartition` routing mode, which randomly selects a single partition to publish all its messages into. This approach can be used to group messages from a specific producer together to maintain message ordering when you don't have a key value. You can also provide your own routing implementation as well if need more control over message distribution across your partitioned topic.

Let's look of the message flow depicted in Figure 2.9 in which the producer is configured to use the `RoundRobinPartition` publish mode. In this scenario the producer connects to the Pulsar proxy and expects back the IP address of the broker assigned to the topic it is writing to. The proxy, in turn refers to the local metastore for this information and discovers that the topic is partitioned and needs to translate the specified partition number into the name of the internal topic that is serving that partition.

The producer is configured to use a specific routing mode, e.g **RoundRobin**, which is used to determine which partition a given message is routed to.

The producer then publishes the message like any other by providing the topic name;
e.g. `publish("partitioned-topic");`

The proxy refers to the local metastore to determine which broker is serving the topic, and based on the partition value provided by the producer determines the address of the broker currently serving the given partition and returns it to the proxy.

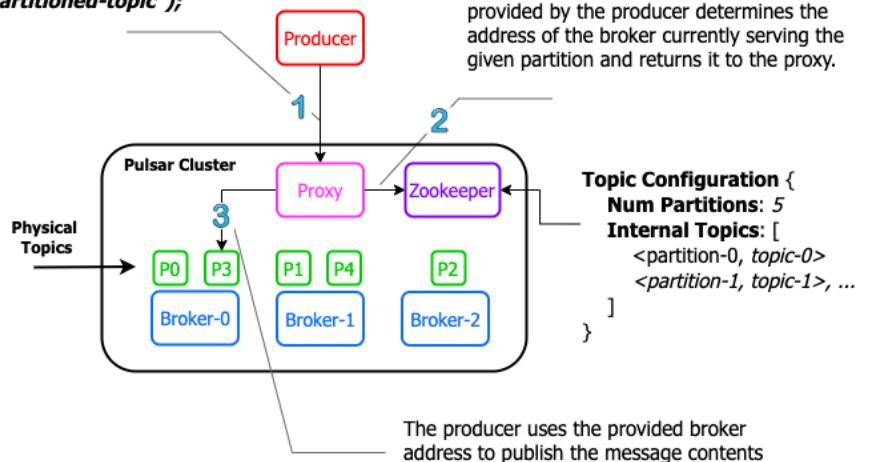


Figure 2.9: Publishing to a Partitioned Topic

In Figure 2.9 the producer's round-robin routing strategy determined that the message should be published to partition number 3 which is implemented as internal topic "p3". The proxy can also determine that internal topic "p3" is currently being served by Broker-0. Therefore, the message is routed to that broker and written to the "p3" topic. Since the routing mode is round robin, a subsequent call by the same producer will result in the message being routed to the "p4" internal topic on Broker-1, etc.

2.2.2 Addressing Topics in Pulsar

The hierarchical structure of Pulsar's logical layer is reflected in the naming convention of endpoints used to access topics within Pulsar. As you can see in Figure 2.10, each topic address within Pulsar contains both the tenant and namespace to which it belongs. The address also contains a persistency prefix that indicates whether the message contents are persisted to long-term storage or if they are only retained in the bookie's memory space. If a topic name is created with a prefix of "persistent://", then all messages that have been received but not yet acknowledged will be stored on multiple bookie nodes, and thus can survive broker failures, etc.

Pulsar also supports non-persistent topics, which retains all unacknowledged messages in the broker memory. Non-persistent topic names begin with the "non-persistent://" prefix to indicate this behavior. When using non-persistent topics, brokers immediately deliver messages to all connected subscribers without persisting them.

(non-)persistent://tenant/namespace/topic-name

Figure 2.10: Topic Addressing Scheme in Pulsar.

When using non-persistent delivery, any form of Broker failure, or disconnecting a subscriber from a topic means that all in-transit messages are lost on that (non-persistent) topic, and subscribers will never be able to receive those messages, even if they reconnect. While non-persistent messaging is usually faster than persistent messaging because it avoids the latency associated with persisting the data to disk, it is only advisable to use them if you are certain that your use case can tolerate the loss of messages.

2.2.3 Producers, Consumers, and Subscriptions

Pulsar is built on the publish-subscribe pattern, aka pub-sub. In this pattern, producers publish messages to topics. Consumers can then subscribe to those topics, process incoming messages, and send an acknowledgement when processing is complete.

A producer is any process that connects to Pulsar broker, either directly or via the Pulsar proxy, and publishes messages to a topic. While a consumer is any process that connects to Pulsar broker to receives messages from a topic. When a consumer has successfully processed a message, it needs to send an acknowledgement to the broker so that the broker knows that it has been received and processed. If no such acknowledgment is received within a preconfigured timeframe, then the Broker will redeliver it to consumers on that subscription.

When a consumer connects to a Pulsar topic, it establishes what is referred to as a "subscription", which specifies how messages are will be delivered to a group of one or more consumers. There are four available subscription modes in Pulsar: exclusive, failover, key-shared, and shared. Regardless of the subscription type, messages are delivered in the order they are received.

Information about these subscriptions is retained in the local Pulsar Zookeeper metadata and includes the http addresses of all the consumers among other things. Each subscription also has a cursor associated with it that represents the position of the last message which was consumed and acknowledged for the subscription. To prevent message re-delivery, these subscription cursors are retained on the bookies to ensure that they will survive any broker level failures.

Pulsar supports multiple subscriptions per topic, which allows multiple consumers to read data from a topic. In this case there are two subscriptions; Sub-A and Sub-B. Consumer-A connected to the topic first and is operating in exclusive consumer mode which means that all the messages in the topic will be consumed by Consumer-A. Thus far Consumer-A has only acknowledged the first 4 messages, so its cursor position for the subscription, Sub-A is currently set 5.

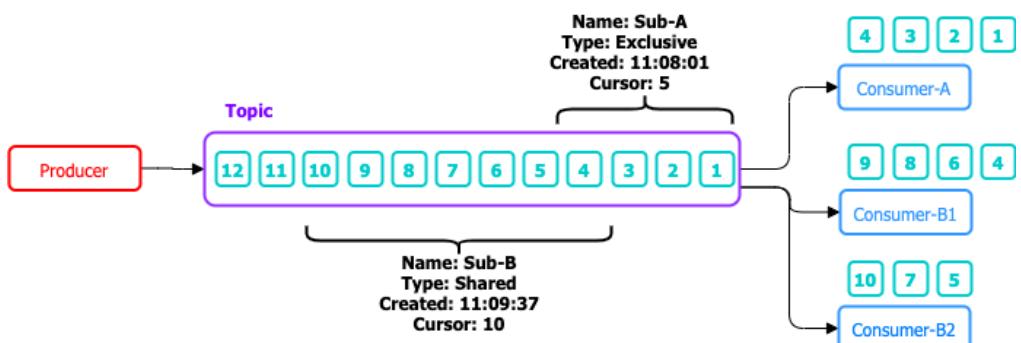


Figure 2.11: Pulsar supports multiple subscriptions per topic, which allows multiple consumers to read the same data.

The subscription named Sub-B was created after the first 3 messages were produced, therefore none of those messages were delivered to the consumers for that subscription. It is a common misconception that any subscriptions created on a topic will start at the very first message for that topic which is why I chose to illustrate that point here and show that you will only receive messages that are published to the topic AFTER you subscribe to it.

We can also see that since Sub-B is operating in shared mode, the messages have been distributed across ALL the consumers in the group, with each message only being processed by a single consumer in the group. You can also see that Sub-B's cursor is further ahead of Sub-A's cursor, which is not uncommon when you distribute the messages across multiple consumers.

2.2.4 Subscription Types

In Pulsar, all consumers use subscriptions to consume data from a topic. Subscriptions are just configuration rules that define how messages are delivered to consumers of a given topic. Pulsar subscriptions can be shared across multiple applications, and in fact, most subscription types are designed specifically for that usage pattern. Pulsar supports four

different types of subscriptions: exclusive, failover, shared, and key-shared as shown in Figure 2.12.

A Pulsar topic can support multiple subscriptions concurrently, allowing you to use a single topic to serve applications with vastly different consumption patterns. It is also important to point out that different subscriptions on the same topic don't have to be of the same subscription type. This allows you to use a single topic to serve both queuing and streaming use cases simultaneously.

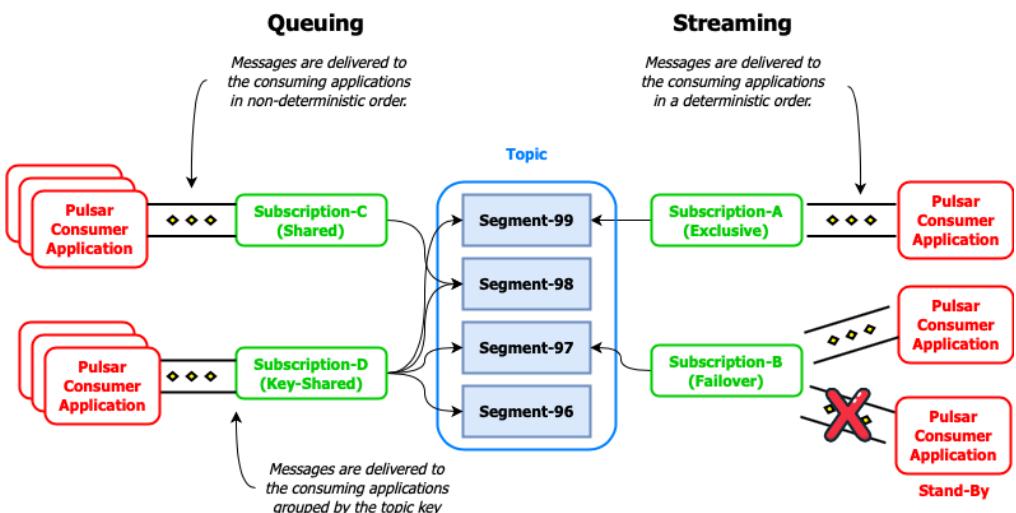


Figure 2.12 Pulsar's Subscription Modes

Each of Pulsar's subscription types serve a different type of use case so it is important to understand them to use them properly. Let's revisit the scenario where a financial services company that streams in real-time stock market quote information into a topic named "stock quotes" and wants to share that information across the entire enterprise and see how each of these subscription modes would be used for the same use cases.

EXCLUSIVE

An exclusive subscription only permits a single consumer to the messages for that subscription. If any other consumer attempts to subscribe to a topic using the same subscription, an exception will be thrown and it won't be able to connect. This mode is used when you want to ensure that each message is processed exactly once, and by a known consumer.

Within our financial services organization, the data science team would use this type of subscription to feed the stock topic data through their machine learning models in order train and or validate them. This would allow them to process the records in exactly the order they were received to provide a stream of stock quotes in the proper time sequence. Each model

would require its own exclusive subscription, as shown in Figure 2.13, to receive its own copy of the data.

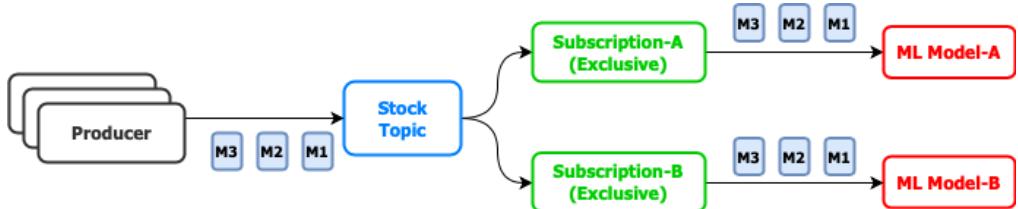


Figure 2.13: An exclusive subscription only permits a single consumer to consume the messages.

FAILOVER SUBSCRIPTIONS

Failover subscriptions allow multiple consumers to attach to the subscription, but only one consumer is selected to receive the messages. This configuration allows for you to provide a failover consumer to continue processing the messages in the topic in the event of a consumer failure. If the active consumer fails to process a message, Pulsar automatically fails over to the next consumer in the list and continues delivering the messages.

This type of subscription is useful when you want single processing semantics with high availability of the consumers. This is useful if you want your application to continue processing the messages in the event of a system failure and another consumer to take over if the first consumer were to fail for any reason. Typically, these consumers are spread across different hosts and/or data centers to ensure that the application can survive multiple outages. As you can see in Figure 2.14, Consumer-A is the active consumer while Consumer-B is the stand-by consumer that would be the next in line to receive messages if Consumer-A disconnected for any reason.

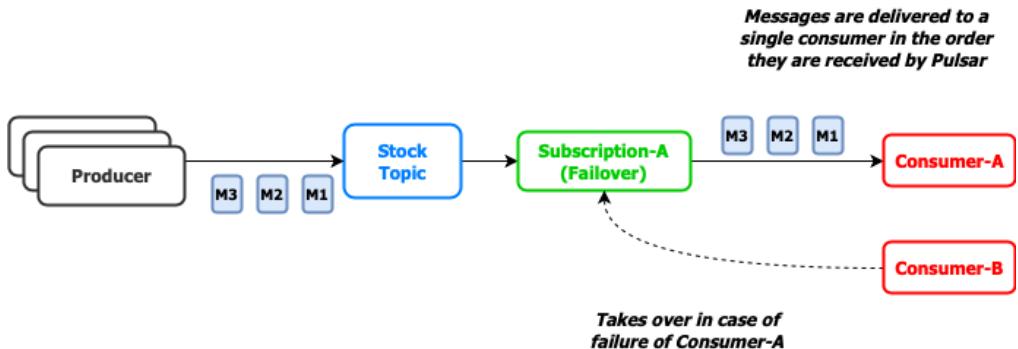


Figure 2.14: A failover subscription has only one active consumer at a time; but permits multiple stand-by consumers.

One such example would be if the data science team from our financial services company had deployed one of their models that uses data from the “stock quotes” topic that generates market volatility scores that are combined with scores from other models to produce an overall recommendation for the trading team. It would be critical that exactly one instance of this model remain up and always running to help the trading team make informed trading decisions. Having multiple instances running and generating recommendations could skew the overall recommendation.

SHARED SUBSCRIPTIONS

Shared subscriptions also allow multiple consumers to attach to the subscription, each of which can actively receive messages unlike failover subscriptions that support only one active consumer at a time. Messages are delivered in a round robin fashion to all the registered consumers, and any given message is delivered to only one consumer as shown in Figure 2.15.

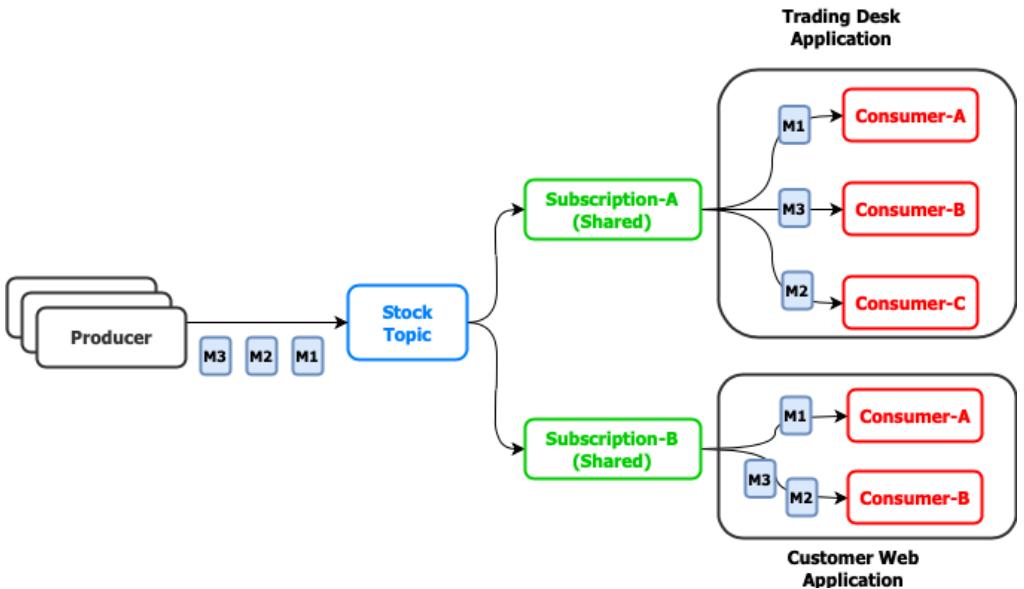


Figure 2.15: Messages are distributed across all consumers of a shared subscription.

This subscription type is useful for implementing work queues, where message ordering isn’t important, as it allows you to scale up the number of consumers on the topic quickly to process the incoming messages. There are no upper limits on the number of consumers per shared subscription, which allows you to scale up consumption that by increasing the number of consumers beyond some artificial limit that is imposed by the storage-layer.

Within our fictitious financial services organization, the business-critical applications, such as our internal trading platforms, algorithmic trading systems, and customer facing website would all benefit from such a subscription. Each of these applications would use their own

shared subscription as shown in Figure 2.15 to ensure that they each received all the messages published to the stock topic.

KEY-SHARED SUBSCRIPTIONS

The key-shared subscription also permitted multiple concurrent consumers, but unlike the shared subscription which distributes the messages in a round-robin manner amongst the consumers, it adds a secondary key index which ensures that messages with the same key get delivered to the same consumer. This subscription acts as a distributed GROUP BY in SQL, where data with similar keys are grouped together. This is particularly useful in cases where you want to pre-sort the data prior to consumption.

Consider the scenario of the business analytics team needing to perform some analytics on the data in the stock topic. By having using a key-shared subscription, they are assured that all the data for a given ticker symbol will be processed by the same consumer as depicted in Figure 2.16. Making it easier for them to join this data with other data streams

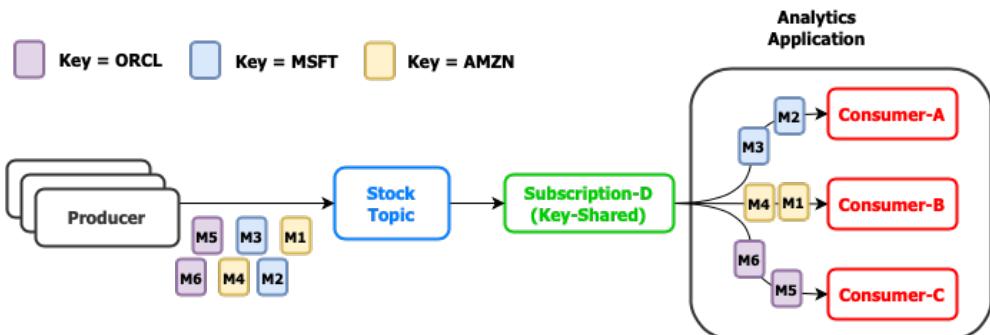


Figure 2.16: Messages are grouped together by the specified key in a shared-key subscription.

In summary, exclusive and failover subscriptions allow only one consumer per topic partition per subscription which ensures that messages are consumed in the order they were received. They are best applied to streaming use cases where strict ordering is required.

Shared subscriptions, on the other hand, allow multiple consumers per topic partition. Each consumer within the subscription receives only a portion of the messages published to a topic. Shared subscriptions are best for queuing use cases where strict message ordering is not required but high throughput is.

2.3 Message Retention and Expiration

As a messaging system, Pulsar's primary function is to move data from point A to point B. Once the data has been delivered to all the intended recipients, the presumption is that there is no need to keep it. Consequently, the default message retention policy in Pulsar does exactly that, when a message is published to a Pulsar topic it will be stored until it has been acknowledged by all the topic's consumers, at which point it is deleted. This behavior is controlled by the `defaultRetentionTimeInMinutes` and `defaultRetentionSizeInMB`

configuration properties in the broker configuration file which are both set to zero by default to indicate that no acknowledged messages should be retained.

2.3.1 Data Retention

However, Pulsar also supports namespace-level retention policies that allow you to override this default behavior for situations where you want to retain the topic data for a longer period such as if you want to access the topic data at a later point in time via the Reader interface or SQL.

These retention policies dictate how long you retain messages in persistent storage after they have been acknowledged as consumed by all its known consumers. Acknowledged messages that are not covered by the retention policy will be deleted. Retention policies are defined by a combination of size and time limits and are applied on a per-topic basis to every topic in that namespace. For instance, if you specify a size limit of 100GB, then up to 100GB worth of data will be retained in each topic within that namespace, and once this size limit is exceeded, messages will be purged from the topic (from oldest to newest) until the total data volume is under the specified limit again. Similarly, if you specify a time limit of 24 hours, then acknowledged messages for all the topics in the namespace will be retained for a maximum of 24 hours based on the time that they were received by the Broker.

The retention policies **require** you to specify **both** a size and a time limit, which are applied independently of one another. Thus, if a message violates either of these limits, it will be removed from the topic regardless of whether it complies with the other policy or not.

If you specify retention policy with time limit of 24 hours and a size limit of 10GB, as shown in Listing 2.4 for the E-payments/refunds namespace, then when either of the specified policy limits are reached, the data is deleted. Therefore, it is possible to have messages that are less than 24 hours old to be deleted if the total volume exceeds 10GB and vice-versa.

Listing 2.4 Setting Various Pulsar Retention Policies

```
./bin/pulsar-admin namespaces set-retention E-payments/payments \
--time 24h \
--size -1      #A

./bin/pulsar-admin namespaces set-retention E-payments/fraud-detection \
--time -1 \
--size 20G     #B

./bin/pulsar-admin namespaces set-retention E-payments/refunds \
--time 24h \
--size 10G     #C

./bin/pulsar-admin namespaces set-retention E-payments/gift-cards \
--time -1 \
--size -1      #D
```

#A Retains all messages less than 24 hours old, with no restriction on the size.

#B Retains up to 20GB of messages with no restriction on the time.

#C Retains up to 10GB of messages less than 24 hours old.

#D Infinite retention of messages

It is also possible to set infinite size or time by specifying a value of -1 for either of those settings when you create the retention policy and providing it for both settings effectively create an infinite retention policy for the namespace. Therefore, be careful when using that policy as the data will never be removed from the storage layer so be sure you have sufficient storage capacity and/or configure periodic offloading of the data to tiered storage.

2.3.2 Backlog Quotas

Backlog is the term used for all the unacknowledged messages in a topic that must be stored on bookies until they are delivered to all the intended recipients. By default, Pulsar retains all unacknowledged messages indefinitely. However, Pulsar supports namespace-level backlog quota policies that allow you to override this behavior to reduce the space consumed by these unacknowledged messages in situations where one or more of the consumers goes offline for an extended period due to system crash, etc.

These backlog quotas are designed to solve a very specific situation in which the topic producers have sent more messages than the consumer can possibly process without falling even further behind. Under these circumstances, you would want to prevent the consumer from getting so far behind that it will never catch-up. When this situation occurs, you need to consider the timeliness of the data that the consumer is processing and ensure that the consumer abandons older, less-recent data in favor of more recent messages that can still be processed within the agreed upon SLA. If the data in your topic becomes “stale” if it sits in there for an extended period, then implementing a backlog quota will help you focus your processing efforts to only the more recent data by limiting the size of the backlog.

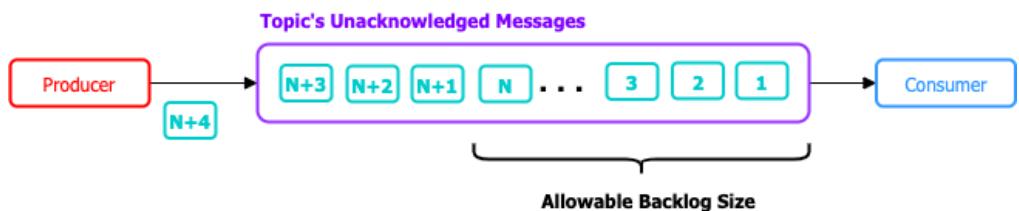


Figure 2.17: Pulsar’s Backlog Quota allows you to dictate what action the Broker should take when the volume of unacknowledged messages exceeds a certain size. This prevents the backlog from growing so large that the consumer is processing data that is of little or no value.

Unlike the message retention policies that I discussed in the previous section which are intended to extend the lifespan of acknowledged messages inside a Pulsar topic, these backlog quota policies are designed to reduce the lifespan of unacknowledged messages.

You can limit the allowable size of these message backlogs by configuring a backlog policy that specifies the maximum allowable size of the topic backlog and the action to take when this threshold is exceeded, as shown in Figure 2.17. There are three distinct options for the backlog retention policy, which dictate the behavior the Broker should take to alleviate the condition:

- The Broker can reject inbound messages by sending an exception to the producers to

- indicate that they should hold off sending new messages, by specifying the `producer_request_hold` retention policy.
- Rather than requesting that the producers hold off, the Broker will forcibly disconnect any existing producers when the `producer_exception` policy is specified.
 - If you want the Broker to discard existing, unacknowledged messages from the topic then you should specify the `consumer_backlog_eviction` policy.

Each of these provide you with three very different approaches to handling the situation shown in Figure 2.17. The first one, `producer_request_hold`, would leave the producer connected, but throw an exception to slow it down. This policy would be applicable in a scenario where you want the client application to catch the thrown exception and resend the message at a later point. So, it would be best to use this policy when you don't want to reject any messages sent from the consumer and the clients will buffer the rejected messages for a period before resending them.

While the second policy, `producer_exception`, would forcibly disconnect the producer entirely which would stop the messages from getting published, but requires the producer code to detect this condition and reconnect automatically. With this policy there is the distinct possibility of losing messages sent from the client producers during the period that they are disconnected. This policy is best to use when you know that the producers aren't capable of buffering messages, e.g., they are running inside a resource constrained environment such as an IoT device, and you don't want Pulsar's inability to receive messages to cause the client application to crash.

The last option does not impact the functionality of the producer whatsoever, and it will continue to produce messages as the current rate. However, older messages that haven't been consumed will be discarded, resulting in message loss.

2.3.3 Message Expiration

As we already discussed, Pulsar retains all unacknowledged messages indefinitely and one of the tools we must prevent these messages from backing up are Backlog Quotas. However, one of the downsides of those is that they only allow you to make your decision on whether to keep a message based on the total space consumed by the topic's unacknowledged messages. As you recall, one of the primary reasons for Backlog Quotas was to ensure that the consumer was ignoring "stale data" in favor of more recent data. Therefore, it would make more sense if there was a way to enforce exactly that based on the age of messages themselves. This is where message expiration policies come in to play.

Pulsar supports namespace-level time-to-live (TTL) policies that allow you to have messages automatically deleted if they remain unacknowledged after a certain period referred to as time-to-live or TTL. Message expiration is useful in situations where it is more important that the application consuming the data be working with more recent data, rather than a complete history. One such example in our application would be driver location data that we will be displaying to our customers while the food is in route. The customer is more interested in the most recent location of the driver than they are in the where they were 5 minutes ago. Therefore, driver location information that is older than 5 minutes would no

longer be relevant and should be purged to allow the consumers to process only the more recent data rather than trying to process messages that are no longer useful.

Listing 2.5 Setting Backlog Quota and Message Expiration Policies

```
./bin/pulsar-admin namespaces set-backlog-quota E-payments/payments \
--limit 2G
--policy producer_request_hold    #A

./bin/pulsar-admin namespaces set-message-ttl E-payments/payments \
--messageTTL 120      #B
```

#A Defines a backlog quota with a size limit of 2GB and producer_request_hold policy.

#B Sets the message TTL to 120 seconds.

A namespace can have both a Backlog Quota and a TTL policy associated with it to provide even finer control over the retention of unacknowledged messages stored inside a Pulsar topic, as shown in Listing 2.5.

2.3.4 Message Backlog vs. Message Expiration

Message retention and message expiration solve two fundamentally different problems. As you can see from Figure 2.18, message retention policies only apply to acknowledged messages, and those messages that fall with the retention policy are retained. Message expiration only applies to un-acknowledged messages and is controlled by the TTL setting, any messages that are not processed and acked within that timeframe are discarded and not processed.

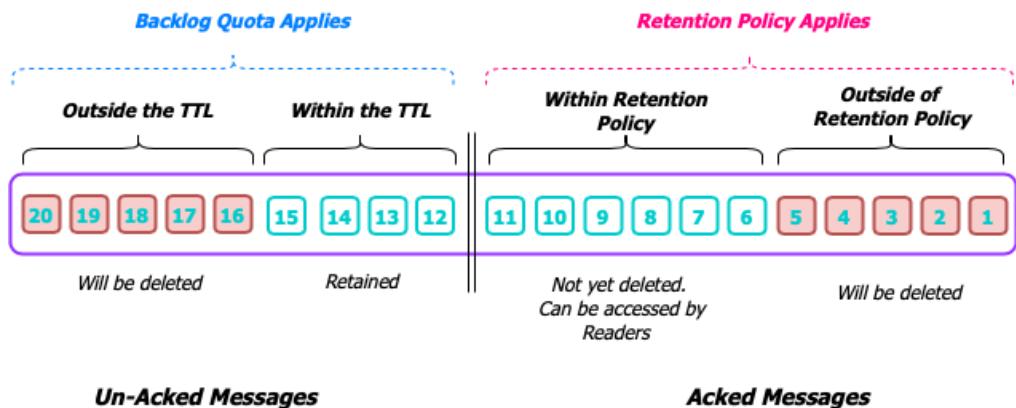


Figure 2.18: The backlog quota applies to messages that have not been acknowledged by all subscriptions and is based on the time-to-live setting, while the retention policy applies to acknowledged messages and is based on the volume of data to retain.

Message retention policies can be used in conjunction with tiered storage to support infinite message retention for critical datasets that you want to retain indefinitely for backup/recovery, event sourcing, or SQL exploration.

2.4 Tiered Storage

Pulsar's Tiered Storage feature allows older topic data to be offloaded to more cost-effective long-term storage, thereby freeing up disk space inside of the Bookies. To the end user there is no difference between consuming a topic whose data is stored inside Apache BookKeeper or on tiered storage. The clients still produce and consume messages in the same way, and the entire process is handled transparently behind the scenes.

As we discussed earlier, Apache Pulsar stores topics as an ordered list of ledgers that are spread across the Bookies in the storage layer. Because these ledgers are append-only, new messages are only written to the final ledger in the list. All the previous ledgers are sealed, so the data within the segment is immutable. Because the data is immutable, it can easily be copied to another storage system such as cloud storage.

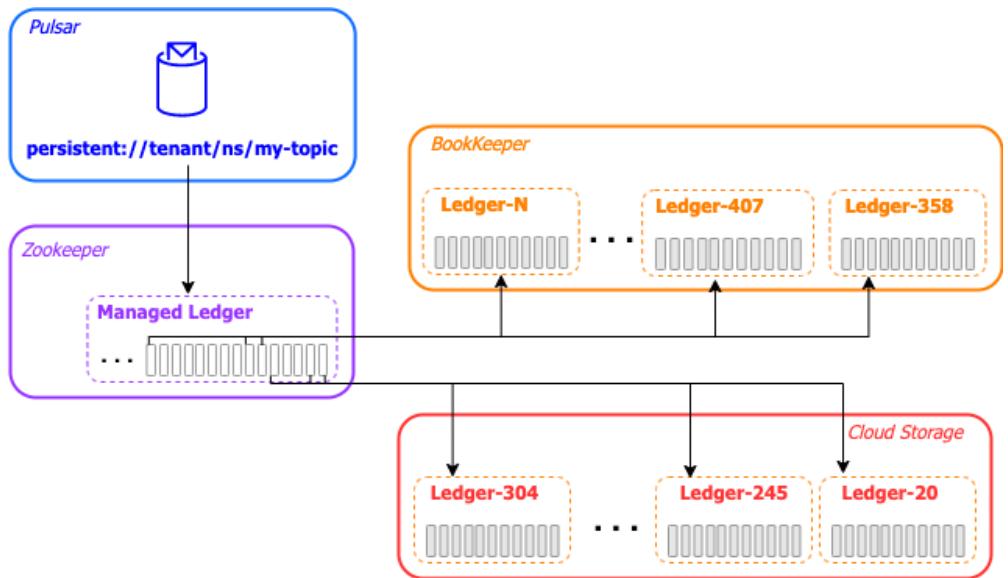


Figure 2.19: When using tiered storage, ledgers that have been closed can be copied over to Cloud Storage and removed from the Bookies to free up space. The managed ledger entries are updated to reflect the new location of the ledgers which can still be read by the topic consumers.

Once the copy is complete, the managed ledger information can be updated to reflect the new storage location of the data as shown in Figure 2.19, and the original copy of the data stored in Apache BookKeeper can be deleted. When a ledger is offloaded to an external

storage system, the ledgers are copied, one-by-one, to that storage system from oldest to newest.

Apache Pulsar currently supports multiple cloud storage systems for tiered storage, but I will focus on using AWS in this section. Please consult the documentation for more details on how to use other cloud vendor's storage systems.

AWS OFFLOAD CONFIGURATION

The first step you need to perform is to create the S3 bucket you will be using to store the offloaded ledgers and ensuring that the AWS account you are going to use has sufficient permissions to read and write data to the bucket. Once that is completed, you will need to modify the broker configuration settings again as shown in Listing 2.6.

Listing 2.6 Configuring AWS Tiered Storage in Pulsar

```
managedLedgerOffloadDriver=aws-s3      #A
s3ManagedLedgerOffloadBucket=offload-test-aws      #B
s3ManagedLedgerOffloadRegion=us-east-1      #C
s3ManagedLedgerOffloadRole=<aws role arn>      #D
s3ManagedLedgerOffloadRoleSessionName=pulsar-s3-offload      #E
```

#A Specifies the offload driver type as AWS S3.

#B The S3 bucket name used for ledger storage.

#C The AWS Region where the bucket is located.

#D If you want the offloader to assume an IAM role to perform its work use this property.

#E Specify the session name to use when assuming an IAM role.

You will need to add the AWS specific settings to tell the Pulsar where to store the ledgers inside of S3. Once these settings are made, you can save the file and restart Pulsar for the changes to take effect.

AWS AUTHENTICATION

For Pulsar to offload data to S3, it must authenticate with AWS using a valid set of credentials. As you may have already noticed, Pulsar doesn't provide any means of configuring authentication for AWS. Instead, it relies on the standard mechanisms supported by the DefaultAWSCredentialsProviderChain which searches for AWS credentials in various pre-defined locations.

If you are running your Broker on AWS instance with an instance profile that provides credentials, Pulsar will use these credentials if no other mechanism is provided. Alternatively, you can provide your credentials via environment variables. The easiest way to do this is to edit the `conf/pulsar_env.sh` file and "export" the environment variables `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` by adding the statements shown in Listing 2.7.

Listing 2.7 Providing AWS Credentials via Environment Variables

```
# Add these at the beginning of the pulsar_env.sh
export AWS_ACCESS_KEY_ID=ABC123456789
export AWS_SECRET_ACCESS_KEY=ded7db27a4558e2ea8bbf0bf37ae0e8521618f366c
```

```
# or you can set them here instead.  
PULSAR_EXTRA_OPTS="${PULSAR_EXTRA_OPTS} ${PULSAR_MEM} ${PULSAR_GC}  
-Daws.accessKeyId=ABC123456789  
-Daws.secretKey=ded7db27a4558e2ea8bbf0bf37ae0e8521618f366c  
-Dio.netty.leakDetectionLevel=disabled  
-Dio.netty.recycler.maxCapacity.default=1000  
-Dio.netty.recycler.linkCapacity=1024"
```

You only need to use one of the two methods shown in Listing 2.8. Both options work equally well, so you can take your pick. However, both methods pose a security risk as these AWS credentials will be visible in the process if you were to run a linux `ps` command. If you would prefer avoid that scenario, you can store your credentials in the traditional location for AWS credentials files; `~/.aws/credentials`, as shown in Listing 2.8, which can be modified to have read-only permissions for the user account that will be launching the Pulsar Broker, e.g. root.

Listing 2.8 Contents of the `~/.aws/credentials` file

```
[default]  
aws_access_key_id=ABC123456789  
aws_secret_access_key=ded7db27a4558e2ea8bbf0bf37ae0e8521618f366c
```

However this approach does require you to store your unencrypted credentials on disk which does introduce some security risks, so it is not recommended for production use.

CONFIGURING OFFLOAD TO RUN AUTOMATICALLY

Simply because we have configured the managed ledger offloader does not mean that the offloading will occur. We still need to define a namespace level policy to have the data offloaded automatically once a certain threshold is reached. The threshold is based on the total volume of data that a Pulsar topic has stored in BookKeeper storage layer.

Listing 2.9 Configuring Automatic Offloads to Tiered Storage

```
/pulsar/bin/pulsar-admin namespaces set-offload-threshold \  
-size 10GB \  
E-payments/payments
```

You can define such a policy as shown in Listing 2.9, which sets a threshold of 10GB for all topics in the namespace. Once a topic reaches 10GB of storage an offload of all closed segments is triggered. Setting the threshold to a zero will cause the Broker to offload ledgers as aggressively as it can and can be used to minimize the amount of topic data stored on BookKeeper. Specifying a negative value for the threshold effectively disables automatic offloaded entirely and can be used for topics with tight SLA response times that cannot tolerate the additional latency required to read data from tiered storage.

Tiered storage should be used when you have a topic for which you want to retain the data for a very long time. One example would be the driver location topic that contains the detailed information about the driver's locations at various points in time. The information inside this topic can be used you use to train the company's travel time prediction models

that are used to estimate the travel time between locations. We will want to keep that data for a long time, to develop models for different periods of time (months) to account for external factors such as weather.

While tiered storage is often used in conjunction with topics that have retention policies that encompass enormous amounts of data, there is no such requirement. It can, in fact be used with any topic.

2.5 Summary

In this chapter you learned the following:

- The logical structure of Pulsar's address space to support multi-tenancy.
- The difference between message retention and message expiration in Pulsar.
- The low-level details of how Pulsar stores and serves messages.

3

Interacting with Pulsar

This chapter covers

- Running a local instance of Pulsar on your development machine
- Administering a Pulsar cluster using its command-line tools
- Interacting with Pulsar using the Java, Python, and Go client libraries
- Troubleshooting Pulsar with its command-line tools

Now that we have covered the overall architecture and terminology of Apache Pulsar, let's start using it. For local development and testing, I recommend running Pulsar inside a Docker container on your own machine, which provides an easy way to get started with Pulsar with a minimal amount of time, effort, and money. For those of you who would prefer to use a full-size Pulsar cluster, you can refer to Appendix A for more details on how to install and run one inside a containerized environment such as Kubernetes.

In this chapter I will walk you through the process of sending and receiving messages programmatically using the Java API, starting with the process of creating a Pulsar namespace and topic using Pulsars administrative tools.

3.1 Getting Started with Pulsar

For the purposes of local development and testing, you can run Pulsar on your development machine within a Docker container. If you don't already have Docker installed, you should download the [community edition](#) and follow the instructions for your operating system. For those of you unfamiliar with Docker, it is an open-source project for automating the deployment of applications as portable, self-contained images that can be run from a single command. Each docker image bundles all the separate software components necessary to run an entire application together into a single deployment. For example, the Docker image for a simple web application would include the web server, database, and application code. In short, everything the application needs to run. Similarly, there is an existing Docker image

that includes a Pulsar broker as well as the necessary Zookeeper and BookKeeper components.

Software developers can then create these Docker images and publish them to a central repository known as “Docker Hub”, to uniquely identify an image, you can specify a tag when you upload the image. This allows people to quickly locate and download the latest version of the image to your development machine. To start the Pulsar Docker container, simply execute the command shown in Listing 3.1 which will download the container image and start all the necessary components. Note that we have specified a pair of ports (6650, and 8080) that will be exposed on your local machine. You will use these ports to interact with the Pulsar cluster later in the chapter.

Listing 3.1 Running Pulsar on Desktop

```
docker pull apache/pulsar/pulsar-standalone    #A  
  
docker run -d \  
  -p 6650:6650 -p 8080:8080 \  
  -v $PWD/data:/pulsar/data \  
  --name pulsar \  
  apache/pulsar/pulsar-standalone    #E
```

```
#A Pull down the latest version from DockerHub  
#B Configure port forwarding for these ports  
#C Retain the data on a local drive  
#D Specify the name of the container  
#E The tag for the standalone image
```

If Pulsar has successfully started, you should be able to locate INFO-level messages in the log file of the Pulsar container indicating that the messaging service is ready like those shown in Listing 3.2. You can access the Docker log files via the `docker log` command, which allows you to locate any issues if your container fails to start.

Listing 3.2 Verifying that the Pulsar cluster is

```
$docker logs pulsar | grep "messaging service is ready"  
  
20:11:45.834 [main] INFO org.apache.pulsar.broker.PulsarService - messaging service is  
ready  
20:11:45.855 [main] INFO org.apache.pulsar.broker.PulsarService - messaging service is  
ready, bootstrap service port = 8080, broker url= pulsar://localhost:6650,  
cluster=standalone
```

These log messages indicate that the Pulsar Broker is up and running and accepting connections on port 6650 of your local development machine. Therefore, all the code examples will be using the `pulsar://localhost:6650` URL to send and receive data from the Pulsar Broker.

3.2 Administering Pulsar

Pulsar provides a single administrative layer that allows you to administer the entire Pulsar instance, including all the sub-clusters, from a single endpoint. Pulsar’s admin layer controls

authentication and authorization for all tenants, resource isolation policies, storage quotas, and more as shown in Figure 3.1.

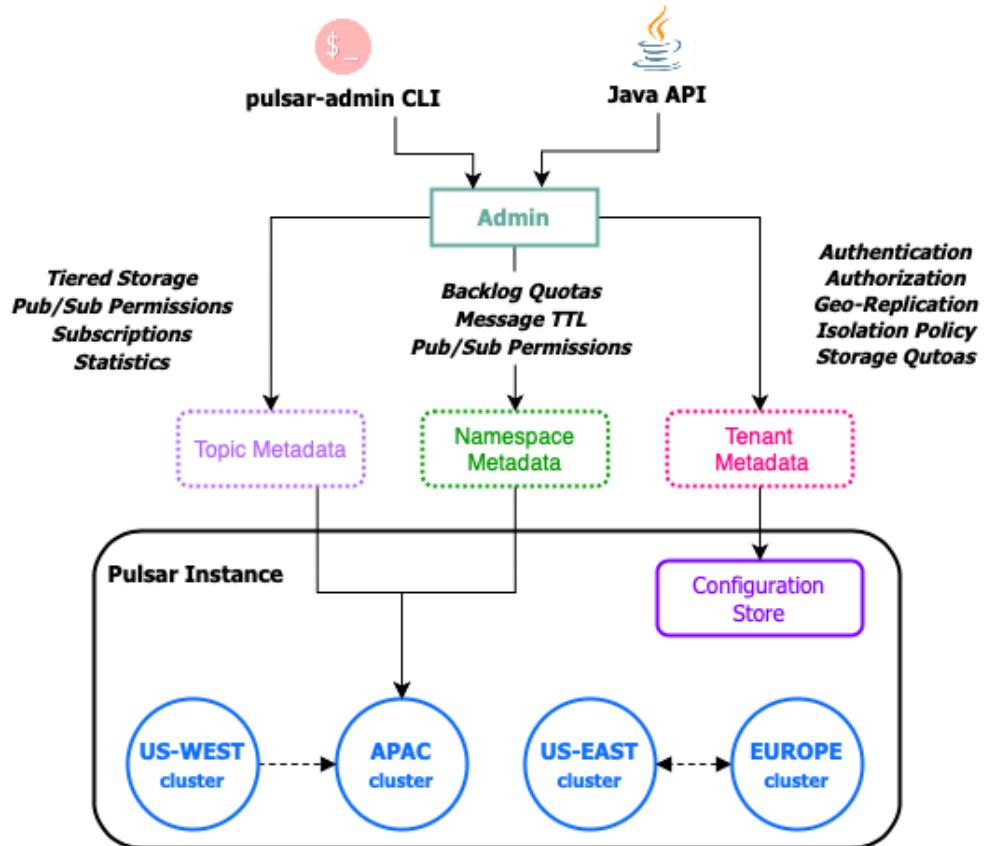


Figure 3.1: Administrative View of Pulsar.

This administrative interface allows you to create and manage all the various entities within a Pulsar cluster such as tenants, namespaces, and topics and configure their various security and data retention policies. Users can interact with this administrative interface via the pulsar admin command-line interface tool or programmatically via a Java API as shown in Figure 3.1

When you start a local standalone cluster, Pulsar automatically creates a “public” tenant with a namespace named “default” that can be used for development purposes. However, this is not a realistic production scenario, so I will demonstrate how to create a tenant and namespace.

3.2.1 Creating a Tenant, Namespace, and Topic

Pulsar provides a command-line interface (CLI) tool called **pulsar-admin** inside the bin folder of your Pulsar installation which in our case is inside the Docker container. Therefore, to use this command line tool, you must execute the command inside the running docker container. Fortunately, Docker provides a method for doing just that via its `docker exec` command. Just like the name implies, this command “executes” the given statement inside the container itself rather than on your local machine.

You can start using the `pulsar-admin` CLI by issuing the sequence of commands shown in Listing 3.3 to create a topic named `persistent://manning/chapter03/example-topic` that we will use in throughout the chapter.

Listing 3.3 Pulsar Admin Commands

```
docker exec -it pulsar /pulsar/bin/pulsar-admin clusters list      #A
"standalone"

docker exec -it pulsar /pulsar/bin/pulsar-admin tenants list      #B
"public"
"sample"

docker exec -it pulsar /pulsar/bin/pulsar-admin tenants create manning   #C

docker exec -it pulsar /pulsar/bin/pulsar-admin tenants list      #D
"manning"
"public"
"sample"

docker exec -it pulsar /pulsar/bin/pulsar-admin namespaces create manning/chapter03    #E

docker exec -it pulsar /pulsar/bin/pulsar-admin namespaces list manning
"manning/chapter03"      #F

docker exec -it pulsar /pulsar/bin/pulsar-admin topics create
    persistent://manning/chapter03/example-topic    #G

docker exec -it pulsar /pulsar/bin/pulsar-admin topics list manning/chapter03    #H
"persistent://manning/chapter03/example-topic"
```

#A List all the clusters in the Pulsar instance

#B List all the tenants in the Pulsar instance

#C Create a new tenant named “manning”

#D Confirm that the new tenant was created

#E Create a new namespace named “chapter03” under the manning tenant

#F List the namespaces under the manning tenant

#G Create a new topic

#H List the topic inside the manning/chapter03 namespace.

These commands barely scratch the surface of what you can do with the Pulsar Admin tool, and I highly recommend that you refer to the [online documentation](#) for more details on the CLI tool and all of its features. We will revisit the Pulsar Admin CLI tool later in the chapter when we need to retrieve some performance metrics from the cluster once we start publishing messages.

3.2.2 Java Admin API

Another way in which you can administer the Pulsar Instance is via the Java Admin API, which provides a programmable interface for performing administrative tasks. Listing 3.4 show how to create the `persistent://manning/chapter03/example-topic` topic using the Java API.

Listing 3.4 Using the Java Admin API

```
import org.apache.pulsar.client.admin.PulsarAdmin;
import org.apache.pulsar.common.policies.data.TenantInfo;

public class CreateTopic {
    public static void main(String[] args) throws Exception {
        PulsarAdmin admin = PulsarAdmin.builder()
            .serviceHttpUrl("http://localhost:8080")      #A
            .build();

        TenantInfo config = new TenantInfo(
            Stream.of("admin").collect(
                Collectors.toCollection(HashSet::new)),      #B
            Stream.of("standalone").collect(
                Collectors.toCollection(HashSet::new)));      #C

        admin.tenants().createTenant("manning", config);    #D
        admin.namespaces().createNamespace("manning/chapter03");   #E
        admin.topics().createNonPartitionedTopic(
            "persistent://manning/chapter03/example-topic");    #F
    }
}
```

#A Create an Admin client for the Pulsar cluster running inside Docker

#B Specify the admin roles for the tenant

#C Specify the clusters that the tenant can operate on

#D Create the tenant

#E Create the namespace

#F Create the topic

This API provides an alternative to the CLI tool and is particularly useful inside of unit tests when you want to create and tear down the necessary Pulsar topics programmatically rather than relying on an external tool.

3.3 Pulsar Clients

Pulsar provides a command-line interface (CLI) tool called **pulsar-client** that allows you to send and receive messages from a topic in a running Pulsar cluster. This tool also resides inside the bin folder of your Pulsar installation, and thus we will need to use the docker exec command again to interact with this tool.

Since the topic has already been created, we can start by first attaching a consumer to it which will establish a subscription and ensure that no messages are lost. This can be accomplished by running the command shown in Listing 3.5. The consumer is a “blocking script”; meaning it will keep consuming messages from the topic until the script is stopped by you (with Ctrl-C).

Listing 3.5 Starting a command-line consumer

```
$ docker exec -it pulsar /pulsar/bin/pulsar-client consume \
persistent://manning/chapter03/example-topic \      #A
--num-messages 0 \      #B
--subscription-name example-sub \      #C
--subscription-type Exclusive      #D

INFO  org.apache.pulsar.client.impl.ConnectionPool - [[id: 0xe410f77d, L:/127.0.0.1:39276 - R:localhost/127.0.0.1:6650]] Connected to server
18:08:15.819 [pulsar-client-io-1-1] INFO
    org.apache.pulsar.client.impl.ConsumerStatsRecorderImpl - Starting Pulsar consumer
    perf with config: {      #E
        "topicNames" : [ ],
        "topicsPattern" : null,
        "subscriptionName" : "example-sub",      #F
        "subscriptionType" : "Exclusive",      #G
        "receiverQueueSize" : 1000,
        "acknowledgementsGroupTimeMicros" : 100000,
        "negativeAckRedeliveryDelayMicros" : 60000000,
        "maxTotalReceiverQueueSizeAcrossPartitions" : 50000,
        "consumerName" : "3d7ce",
        "ackTimeoutMillis" : 0,
        "tickDurationMillis" : 1000,
        "priorityLevel" : 0,
        "cryptoFailureAction" : "FAIL",
        "properties" : { },
        "readCompacted" : false,
        "subscriptionInitialPosition" : "Latest",      #H
        "patternAutoDiscoveryPeriod" : 1,
        "regexSubscriptionMode" : "PersistentOnly",
        "deadLetterPolicy" : null,
        "autoUpdatePartitions" : true,
        "replicateSubscriptionState" : false,
        "resetIncludeHead" : false
    }
...
18:08:15.980 [pulsar-client-io-1-1] INFO
    org.apache.pulsar.client.impl.MultiTopicsConsumerImpl -
        [persistent://manning/chapter02/example] [example-sub] Success subscribe new topic
        persistent://manning/chapter02/example in topics consumer, partitions: 2,
        allTopicPartitionsNumber: 2
18:08:47.644 [pulsar-client-io-1-1] INFO  com.scurrilous.circe.checksum.Crc32cIntChecksum -
    SSE4.2 CRC32C provider initialized
```

#A The name of the topic we are consuming from

#B The number of messages to consume, 0 means consume forever

#C The unique name of the subscription

#D The type of subscription

#E Consumer configuration details

#F You can see the subscription name we specified on the command line

#G You can see the subscription type we specified on the command line

#H Start consuming from the latest available message.

In a different shell, we will start a producer by issuing the command shown in Listing 3.6 to send two messages containing the text “Hello Pulsar” to the same topic that we just started the consumer on.

Listing 3.6 Sending a message using the Pulsar command-line producer

```
$ docker exec -it pulsar /pulsar/bin/pulsar-client produce \
persistent://manning/chapter03/example-topic \      #A
--num-produce 2 \      #B
--messages "Hello Pulsar"    #C
18:08:47.106 [pulsar-client-io-1-1] INFO  org.apache.pulsar.client.impl.ConnectionPool -
[[id: 0xd47ac4ea, L:/127.0.0.1:39342 - R:localhost/127.0.0.1:6650]] Connected to
server
18:08:47.367 [pulsar-client-io-1-1] INFO
org.apache.pulsar.client.impl.ProducerStatsRecorderImpl - Starting Pulsar producer
perf with config: { #D
"topicName" : "persistent://manning/chapter02/example",
"producerName" : null,
"sendTimeoutMs" : 30000,
"blockIfQueueFull" : false,
"maxPendingMessages" : 1000,
"maxPendingMessagesAcrossPartitions" : 50000,
"messageRoutingMode" : "RoundRobinPartition",
"hashingScheme" : "JavaStringHash",
"cryptoFailureAction" : "FAIL",
"batchingMaxPublishDelayMicros" : 1000,
"batchingMaxMessages" : 1000,
"batchingEnabled" : true,
"compressionType" : "NONE",
"initialSequenceId" : null,
"autoUpdatePartitions" : true,
"properties" : { }
}
...
18:08:47.689 [main] INFO  org.apache.pulsar.client.cli.PulsarClientTool - 2 messages
successfully produced    #E
```

#A The name of the topic we are publishing to

#B The number of times to send the message

#C The message contents

#D Producer configuration details.

#E The publishing of the messages

After executing the producer command in Listing 3.6, you should see something like Listing 3.7 inside the shell where you started the consumer. This indicates that the messages were successfully published by the producer and received by the consumer.

Listing 3.7 Receipt of Messages in Consumer Shell

```
----- got message -----
key:[null], properties:[], content>Hello Pulsar
----- got message -----
key:[null], properties:[], content>Hello Pulsar
```

Congratulations, you have just successfully sent your first messages using Pulsar! Now that we have confirmed that our local Pulsar cluster is working and capable of sending and receiving messages, let's look at some more realistic examples using various programming languages. Pulsar provides a simple and intuitive client API that encapsulates all the broker-

client communication details from the user. Due to the popularity of Pulsar, there are several language specific implementations of this client including Java, Go, Python, and C++. This allows each team in our organization to use whatever language they like to implement their services.

While there are significant discrepancies in the features supported by the official Pulsar client libraries based on the programming language you chose (please refer to the official client documentation for details), under the covers they all support transparent reconnection and/or connection failover to brokers, queuing of messages until acknowledged by the broker, and heuristics such as connection retries with backoff. This allows the developer to focus on the messaging logic rather than having to handle connection exceptions in their application code.

3.3.1 The Pulsar Java Client

In addition to the Java Admin API that we looked at earlier in the chapter, Pulsar also provides a Java client that can be used to create producers, consumers, and message readers. The latest version of the Pulsar Java client library is available in the Maven central repository. To use the latest version, simply add the pulsar-client library to your build configuration as shown in Listing 3.8.

Listing 3.8 Adding the Pulsar client library to your Maven project

```
<!-- Inside your pom.xml -->
<properties>
    <pulsar.version>2.7.2</pulsar.version>
</properties>

<dependency>
    <groupId>org.apache.pulsar</groupId>
    <artifactId>pulsar-client</artifactId>
    <version>${pulsar.version}</version>
</dependency>
```

Once you have added the Pulsar client library to your project, you can start using it to interact with Pulsar by creating clients, producers, and consumers inside your Java code as we shall see in the next section.

PULSAR CLIENT CONFIGURATION IN JAVA

When an application wants to create either a producer or a consumer, you first need to instantiate a *PulsarClient* object using code like that shown in Listing 3.9 where you provide the URL of the Pulsar Broker along with any other connection configuration information that may be required, such as security credentials etc.

Listing 3.9 Creating a PulsarClient in Java

```
PulsarClient client = PulsarClient.builder()
    .serviceUrl("pulsar://localhost:6650")      #A
    .build();
```

#A The connection URL to the Pulsar Broker

The *PulsarClient* object handles all the low-level details involved in creating a connection to the Pulsar broker including automatic retries, and connection security if the Pulsar broker has TLS configured. Client instances are thread-safe and can be reused for creating and managing multiple producers and consumers.

PULSAR PRODUCERS IN JAVA

In Pulsar, producers are used to write messages to topics. Listing 3.10 shows how you can create a Producer in Java by specifying the name of the topic you are going to send messages to. While there are several configuration settings that can be used when creating a Producer, all that is required is the topic name itself.

Listing 3.10 Creating a Pulsar Producer in Java

```
Producer<byte[]> producer = client.newProducer()
    .topic("persistent://manning/chapter03/example-topic")
    .create();
```

It is also possible to attach metadata to a given message as shown in Listing 3.11, which shows how to specify the message key that is used for routing with a key-shared subscription along with some message properties. This capability can be used to tag the message with useful information such as when the message was sent, who sent the message, e.g., the device id if the message is from an embedded sensor, etc.

Listing 3.11 Specifying Metadata in Pulsar Messages

```
Producer<byte[]> producer = client.newProducer()
    .topic("persistent://manning/chapter03/example-topic")
    .create();

producer.newMessage()
    .key("some-key")          #A
    .value("my-message".getBytes())      #B
    .property("property_1", "value_1")    #C
    .property("property_2", "value_2")
    .send();
```

#A You can specify a message key

#B Send the message content as a byte array

#C You can attach as many properties as you like

The metadata values you attach to the message will be available to the message consumers who can then use that information when performing their processing logic. For example, a property containing a timestamp value that represents when the message was sent could be used to sort the incoming messages into chronological order of occurrence or used to correlate it with messages from another topic.

PULSAR CONSUMERS IN JAVA

In Pulsar, the consumer interface is used to listen on a specific topic and process the incoming messages. After a message has been successfully processed an acknowledgement should be sent back to the Broker to indicate that we are done processing the message

within the subscription. This allows the Broker to know which message in the topic needs to be delivered to the next consumer on the subscription. In Java, you can create a Consumer by specifying a topic and a subscription as shown in Listing 3.12.

Listing 3.12 Creating a Pulsar Consumer in Java

```
Consumer consumer = client.newConsumer()  
    .topic("persistent://manning/chapter03/example-topic")      #A  
    .subscriptionName("my-subscription")      #B  
    .subscribe();
```

#A You specify the topic you want to consume from
#B You must specify the unique name of your subscription

The subscribe method will attempt to connect the consumer to the topic using the specified subscription, which may fail if the subscription already exists and it isn't one of the shared subscription types, e.g., you attempt to connect to an exclusive subscription that already has an active consumer. If you are connecting to the topic for the first time using the specified subscription name, then a subscription is created for you automatically. Whenever a new subscription is created, it is initially positioned at the end of the topic by default and consumers on that subscription will begin reading the first message created after the subscription was created. If you are connecting to a pre-existing subscription, then it will begin reading from the earliest unacknowledged message within the subscription, as shown in Figure 3.2.

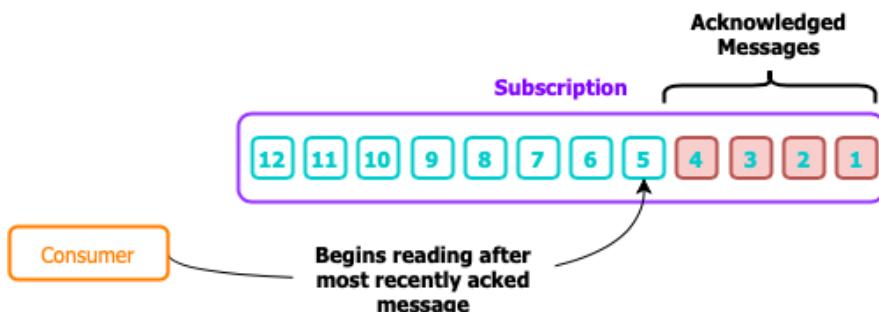


Figure 3.2: The consumer starts reading messages immediate after the most recently acknowledged message in the subscription. If the subscription is new, then it starts reading the messages that are added to the topic after the subscription was created.

One common consumption pattern is to have the consumer listen on the topic inside a while loop. In Listing 3.13, the consumer continuously listens for messages, prints the contents of any message that's received, and then acknowledges that the message has been processed. If the processing logic fails, we use negative acknowledgement to have the message redelivered at a later point in time.

Listing 3.13 Consuming Pulsar Messages in Java

```
while (true) {
    // Wait for a message
    Message msg = consumer.receive();      #A

    try {
        System.out.println("Message received: " +
                           new String(msg.getData()));      #B
        consumer.acknowledge(msg);          #C
    } catch (Exception e) {
        consumer.negativeAcknowledge(msg);  #D
    }
}
```

#A Wait for a message.

#B Process the message.

#C Acknowledge the message, so it can be deleted by the Broker.

#D Mark the message for redelivery.

The message consumer shown in Listing 3.13 processes the messages in a synchronous manner because the `receive()` method that it is using to retrieve messages is blocking method, e.g., it waits indefinitely for a new message to arrive. While this might be fine for some use cases where the message volume is low and/or we are not concerned about the latency between when a message is published and when it is processed, generally synchronous processing is not the best approach. A better approach is to process these messages in an asynchronous manner as shown in Listing 3.14 which relies on the `MessageListener` interface provided by the Java API.

Listing 3.14 Asynchronous Message Processing in Java

```
package com.manning.pulsar.chapter3.consumers;

import java.util.stream.IntStream;

import org.apache.pulsar.client.api.ConsumerBuilder;
import org.apache.pulsar.client.api.PulsarClient;
import org.apache.pulsar.client.api.PulsarClientException;
import org.apache.pulsar.client.api.SubscriptionType;

public class MessageListenerExample {

    public static void main() throws PulsarClientException {

        PulsarClient client = PulsarClient.builder()      #A
            .serviceUrl(PULSAR_SERVICE_URL)
            .build();

        ConsumerBuilder<byte[]> consumerBuilder =      #B
            client.newConsumer()
                .topic(MY_TOPIC)
                .subscriptionName(SUBSCRIPTION)
                .subscriptionType(SubscriptionType.Shared)
                .messageListener((consumer, msg) -> {      #C
                    try {
                        System.out.println("Message received: " +

```

```

        new String(msg.getData()));
    consumer.acknowledge(msg);
} catch (PulsarClientException e) {
}
})

IntStream.range(0, 4).forEach(i -> { #D
    String name = String.format("mq-consumer-%d", i);
    try {
        consumerBuilder
            .consumerName(name)
            .subscribe(); #E
    } catch (PulsarClientException e) {
        e.printStackTrace();
    }
});

...
}

```

#A The Pulsar client used to connect to Pulsar.
#B The consumer factory that will be used to create the consumer instances later.
#C The business logic to execute when a message is received.
#D Create five consumers on the topic, each with the same MessageListener implementation.
#E Connects the consumer to the topic to start receiving messages.

When using the `MessageListener` interface as shown in Listing 3.14, you pass in the code that you want executed whenever a message is received. In this case I used a Java Lambda to provide the code inline, but you can see that I still have access to the consumer that I can use to acknowledge the message. Using the listener pattern allows you to separate the business logic from the management of the threads because the Pulsar consumer automatically creates a thread pool for running the `MessageListeners` instances and handles all the threading logic for you.

Putting this all together we have a Java program in Listing 3.15 that instantiates a Pulsar client and uses it to create a Producer and a Consumer that exchange messages over the “my-topic” topic.

Listing 3.15 Endless Pulsar Producer and Consumer Pair.

```

import org.apache.pulsar.client.api.Consumer;
import org.apache.pulsar.client.api.Message;
import org.apache.pulsar.client.api.Producer;
import org.apache.pulsar.client.api.PulsarClient;
import org.apache.pulsar.client.api.PulsarClientException;

public class BackAndForth {

    public static void main(String[] args) throws Exception {
        BackAndForth sl = new BackAndForth();
        sl.startConsumer();
        sl.startProducer();
    }
    private String serviceUrl = "pulsar://localhost:6650";
}

```

```

String topic = "persistent://manning/chapter03/example-topic";
String subscriptionName = "my-sub";

protected void startProducer() {
    Runnable run = () -> {
        int counter = 0;
        while (true) {
            try {
                getProducer().newMessage()
                    .value(String.format("{id: %d, time: %tc}",
                        ++counter, new Date()).getBytes())
                    .send();
                Thread.sleep(1000);
            } catch (final Exception ex) { }
        };
        new Thread(run).start();
    }
}

protected void startConsumer() {
    Runnable run = () -> {
        while (true) {
            Message<byte[]> msg = null;
            try {
                msg = getConsumer().receive();
                System.out.printf("Message received: %s \n",
                    new String(msg.getData()));
                getConsumer().acknowledge(msg);
            } catch (Exception e) {
                System.err.printf(
                    "Unable to consume message: %s \n", e.getMessage());
                consumer.negativeAcknowledge(msg);
            }
        };
        new Thread(run).start();
    }
}

protected Consumer<byte[]> getConsumer() throws PulsarClientException {
    if (consumer == null) {
        consumer = getClient().newConsumer()
            .topic(topic)
            .subscriptionName(subscriptionName)
            .subscriptionType(SubscriptionType.Shared)
            .subscribe();
    }
    return consumer;
}

protected Producer<byte[]> getProducer() throws PulsarClientException {
    if (producer == null) {
        producer = getClient().newProducer()
            .topic(topic).create();
    }
    return producer;
}

protected PulsarClient getClient() throws PulsarClientException {
    if (client == null) {
        client = PulsarClient.builder()
            .serviceUrl(serviceUrl)

```

```
        .build();
    }
    return client;
}
}
```

As you can see, this code creates both a producer and consumer on the same topic and runs them simultaneously in separate threads. If you run this code, you should see output like what is shown in Listing 3.16.

Listing 3.16 Endless Pulsar Producer and Consumer Pair output

```
Message received: {id: 1, time: Sun Sep 06 16:24:04 PDT 2020}
Message received: {id: 2, time: Sun Sep 06 16:24:05 PDT 2020}
Message received: {id: 3, time: Sun Sep 06 16:24:06 PDT 2020}
...

```

Notice how the first two messages we sent earlier are not included in the output since the subscription was created AFTER those messages were published. This is direct contrast to the Reader interface which we will examine shortly.

DEAD LETTER POLICY

While there are several configuration options for a Pulsar consumer that are described in the [online documentation](#), I wanted to highlight the dead-letter-policy configuration, which is useful when you encounter messages that cannot be processed successfully, such as when you are parsing unstructured messages from a topic. Under normal processing conditions, these messages would cause an exception to be thrown.

At this point you have a couple of options; the first is to trap any exceptions and simply acknowledge these messages as successfully processed anyways, which effectively ignores them. Another option is to have them redelivered by negatively acknowledging them. However, this approach might result in an infinite re-delivery loop for these messages if the underlying issue with the messages cannot be resolved, e.g., a message that cannot be parsed will always throw an exception no matter how many times you process it. A third option is to route these problematic messages to a separate topic known as a dead-letter-topic. This allows you to avoid the infinite re-delivery loop while retaining the messages for further processing and/or examination at a later point in time.

Listing 3.17 Configure the Dead-Letter-Topic Policy on a Consumer

```
Consumer consumer = client.newConsumer()
    .topic("persistent://manning/chapter03/example-topic")
    .subscriptionName("my-subscription")
    .deadLetterPolicy(DeadLetterPolicy.builder()
        .maxRedeliverCount(10)      #A
        .deadLetterTopic("persistent://manning/chapter03/my-dlq"))    #B
    .subscribe();
```

#A Set the max redelivery count
#B Set the dead-letter topic name

To configure a dead-letter policy for a particular consumer, Pulsar requires you to specify a few properties, such as the max redelivery count, when you first build it as shown in Listing 3.17. When a message exceeds the user specified maximum redelivery count, it will be sent to the dead-letter topic and acknowledged automatically. These messages can then be examined at a later point in time.

PULSAR READERS IN JAVA

The reader interface allows applications to manage the position from which they will consume messages. When you connect to a topic using a reader, you must specify which message the reader will begin consuming messages from when it connects to the topic. In short, the reader interface provides Pulsar clients with a low-level abstraction that allows them to manually position themselves within a topic.

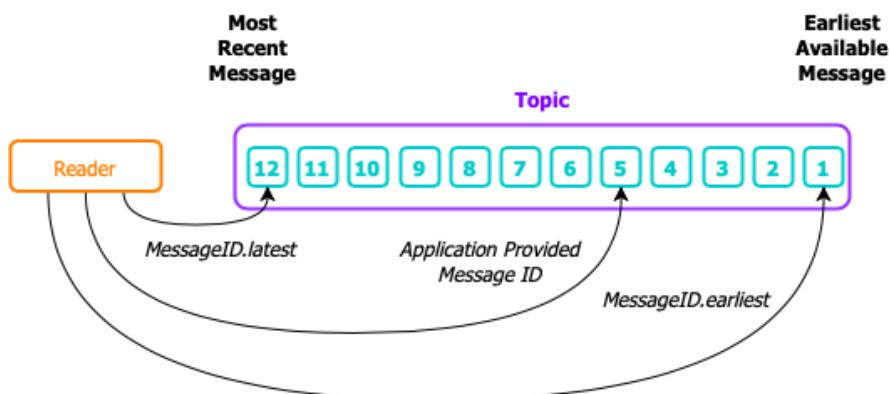


Figure 3.3: When connecting to a topic, the reader interface enables you to begin with; the earliest available message, the latest available message, or an application provided message ID.

The reader interface is helpful for use cases like using Pulsar to provide effectively once processing semantics for a stream processing system. For this use case, it's essential that the stream processing system be able to "rewind" topics to a specific message and begin reading there. If you choose to explicitly provide a message ID, then your application will be responsible for "knowing" this message ID in advance, perhaps fetching it from a persistent data store or cache. Once you've instantiated a `PulsarClient` object, you can create a Reader as shown in Listing 3.18.

Listing 3.18 Creating a Pulsar Reader

```
Reader<byte[]> reader = client.newReader()
    .topic("persistent://manning/chapter03/example-topic")      #A
    .readerName("my-reader")
    .startMessageId(MessageId.earliest)    #B
    .create();

while (true) {
    Message msg = reader.readNext();
```

```
        System.out.printf("Message received: %s \n", new String(msg.getData()));
    }
```

#A You specify the topic you want to read from

#B Specify that we want to read from the earliest message

If you run this code, you should see output like what is shown in Listing 3.19, because you would start reading from the very first message that was published to the topic, which were the two “Hello Pulsar” messages we send from the CLI tool.

Listing 3.19 Earliest Message Reader Output

```
Message read: Hello Pulsar
Message read: Hello Pulsar
Message read: {id: 1, time: Sun Sep 06 18:11:59 PDT 2020}
Message read: {id: 2, time: Sun Sep 06 18:12:00 PDT 2020}
Message read: {id: 3, time: Sun Sep 06 18:12:01 PDT 2020}
Message read: {id: 4, time: Sun Sep 06 18:12:02 PDT 2020}
Message read: {id: 5, time: Sun Sep 06 18:12:04 PDT 2020}
Message read: {id: 6, time: Sun Sep 06 18:12:05 PDT 2020}
Message read: {id: 7, time: Sun Sep 06 18:12:06 PDT 2020}
```

In the example shown in Listing 3.18, a reader is created on the specified topic and iterates over each message in the topic starting with the oldest message in the topic. There are several configuration options for a Pulsar reader that are described in the [online documentation](#), but for most cases the default options are sufficient.

3.3.2 The Pulsar Python Client

There is also an officially supported Pulsar client for the Python programming language. The latest version of the Pulsar client library can be easily installed using the pip package manager using the commands shown in Listing 3.20.

Listing 3.20 Creating a Pulsar Producer in Python

```
pip3 install pulsar-client==2.6.3 -user      #A
pip3 list      #B
Package          Version
-----
...
pulsar-client   2.6.3      #C

#A Install the pulsar client
#B List all the packages
#C Confirm that the correct version of the Pulsar client has been installed
```

Since Python 2.7 has already passed its official end-of-life, I decided to use Python3 for all the examples throughout the chapter. Once you have installed the Pulsar client libraries, you can start using them to interact with Pulsar by creating producers and consumers inside your Python code.

PULSAR PRODUCERS IN PYTHON

When a Python application wants to create either a producer or a consumer, you first need to instantiate a *client* object using code like that shown in Listing 3.21 where you provide the URL of the Pulsar Broker. As was the case for Java-based clients, the Python client object handles all the low-level details involved in creating a connection to the Pulsar broker including automatic retries, and connection security if the Pulsar broker has TLS configured. Client instances are thread-safe and can be reused for managing multiple producers and consumers.

Listing 3.21 Creating a Pulsar Producer in Python

```
import pulsar

client = pulsar.Client('pulsar://localhost:6650')      #A

producer = client.create_producer(
    'persistent://public/default/my-topic',
    block_if_queue_full=True,
    batching_enabled=True,
    batching_max_publish_delay_ms=10)      #B

for i in range(10):
    producer.send(('Hello-%d' % i).encode('utf-8'),      #C
                  properties=None,      #D
                  partition_key='my-key')      #E

client.close()      #F

#A Create a Pulsar client by providing the connection URL to the Pulsar Broker
#B Use the Pulsar client to create a producer
#C Send the message contents
#D You can attach properties to a message if you want
#E You can assign a key to the message for key-based subscriptions, etc.
#F Close the client
```

As you can see from Listing 3.21, the Python library provides several different configuration options when you create your clients, producers, and consumers so you should visit the available [online documentation](#) for the Python client to learn more about these options. In our case, we enabled message batching on the client side which means that rather than sending/receiving each individual message to/from the Broker, messages will be grouped together in batches before being transmitted. This allows us to increase the overall throughput of the messages at the expense of increased latency on each individual message.

PULSAR CONSUMERS IN PYTHON

In Pulsar, the consumer interface is used to listen on a specific topic and process the incoming messages. After a message has been successfully processed an acknowledgement should be sent back to the Broker to indicate that we are done processing the message within the subscription. This allows the Broker to know which message in the topic needs to be delivered to the next consumer on the subscription. In Python, you can create a Consumer by specifying a topic and a subscription as shown in Listing 3.22.

Listing 3.22 Creating a Pulsar Consumer in Python

```
import pulsar

client = pulsar.Client('pulsar://localhost:6650')      #A

consumer = client.subscribe(
    'persistent://public/default/my-topic',      #C
    'my-subscription',      #D
    consumer_type=pulsar.ConsumerType.Exclusive
    initial_position=pulsar.InitialPosition.Latest,
    message_listener=None,
    negative_ack_redelivery_delay_ms=60000)

while True:
    msg = consumer.receive()      #E
    try:
        print("Received message '%s' id='%s'",
              msg.data().decode('utf-8'), msg.message_id())
        consumer.acknowledge(msg)    #F
    except:
        consumer.negative_acknowledge(msg)    #G

client.close()      #H
```

#A Create a Pulsar client by providing the connection URL to the Pulsar Broker

#B Use the Pulsar client to create a consumer

#C You must specify the topic you want to consume from

#D You must specify the unique name of your subscription

#E Wait for a new message to arrive

#F Once we have successfully processed the message, acknowledge it.

#G If we encountered an error, send a negative acknowledgement to have the message resent.

#H Close the client

The subscribe method will attempt to connect the consumer to the topic using the specified subscription, which may fail if the subscription already exists and it isn't one of the shared subscription types, e.g., you attempt to connect to an exclusive subscription that already has an active consumer. If you are connecting to the topic for the first time using the specified subscription name, then a subscription is created for you automatically. Whenever a new subscription is created, it is initially positioned at the end of the topic by default and consumers on that subscription will begin reading the first message created after the subscription was created. If you are connecting to a pre-existing subscription, then it will begin reading from the earliest unacknowledged message within the subscription, as we saw earlier in Figure 3.2.

As you can see from Listing 3.22, the Python library provides several different configuration options when specifying the subscription including the subscription type, starting position, etc. I highly recommend you visit the available [online documentation](#) for the Python client to get the most up-to-date listing of these options.

The message consumer shown in Listing 3.22 processes the messages in a synchronous manner because the `receive()` method that it is using to retrieve messages is blocking method, e.g., it waits indefinitely for a new message to arrive. A better approach is to process these messages in an asynchronous manner as shown in Listing 3.23.

Listing 3.23 Asynchronous Message Processing in Python

```
import pulsar

def my_listener(consumer, msg):      #A
    # process message
    print("my_listener read message '%s' id='%s'", 
        msg.data().decode('utf-8'), msg.message_id())      #B
    consumer.acknowledge(msg)      #C

client = pulsar.Client('pulsar://localhost:6650')

consumer = client.subscribe(
    'persistent://public/default/my-topic',
    'my-subscription',
    consumer_type=pulsar.ConsumerType.Exclusive,
    initial_position=pulsar.InitialPosition.Latest,
    message_listener=my_listener,      #D
    negative_ack_redelivery_delay_ms=60000)

client.close()
```

#A The listener function needs to accept the consumer and the message

#B We can access the message contents

#C We can use the consumer to acknowledge the message

#D Sets a message listener for the consumer

Using the listener pattern allows you to separate the business logic from the management of the threads because the Pulsar consumer automatically creates a thread pool for running the message listener instances and handles all the threading logic for you.

PULSAR READERS IN PYTHON

The Python client also provides a reader interface that enables consumers to manage the position from which they will consume messages. When you connect to a topic using a reader, you must specify which message the reader will begin consuming messages from when it connects to the topic. If you choose to explicitly provide a message ID, then your application will be responsible for "knowing" this message ID in advance and should store that information in a persistent data store somewhere such as a database or cache.

Listing 3.24 Creating a Pulsar Reader in Python

```
import pulsar

client = pulsar.Client('pulsar://localhost:6650')      #A

reader = client.create_reader(
    'persistent://public/default/my-topic',      #B
    pulsar.MessageId.earliest)      #C

while True:
    msg = reader.read_next()      #D
    print("Read message '%s' id='%s'", 
        msg.data().decode('utf-8'), msg.message_id())

client.close()      #E
```

```
#A Create a Pulsar client by providing the connection URL to the Pulsar Broker  
#B Use the Pulsar client to create a reader on the specified topic  
#C Specify that we want to read from the earliest message  
#D Read the messages  
#E Close the client
```

The code shown in Listing 3.24 connects to the topic and starts reading messages from the earliest available message and outputs their contents.

3.3.3 The Pulsar Go Client

There is also an officially supported Pulsar client for the Golang programming language and the latest version of the pulsar client library can be installed using the following command: `go get -u "github.com/apache/pulsar-client-go/pulsar"`. Once you have installed the Pulsar client libraries, you can start using them to interact with Pulsar by creating producers and consumers inside your Go code.

CREATING A PULSAR CLIENT WITH GO

When a Go application wants to create either a producer or a consumer, you first need to instantiate a *Client* object using code like that shown in Listing 3.25 where you provide the URL of the Pulsar Broker along with any other connection configuration information that may be required, such as security credentials etc.

Listing 3.25 Creating a Pulsar Client in Go

```
import (  
    "log"  
    "time"  
    "github.com/apache/pulsar-client-go/pulsar"      #A  
)  
  
func main() {  
    client, err := pulsar.NewClient(      #B  
        pulsar.ClientOptions{      #C  
            URL:                  "pulsar://localhost:6650",  
            OperationTimeout: 30 * time.Second,  
            ConnectionTimeout: 30 * time.Second,  
        })  
  
    if err != nil {      #D  
        log.Fatalf("Could not instantiate Pulsar client: %v", err)  
    }  
  
    defer client.Close()  
}
```

```
#A Import the pulsar client library  
#B Create a new client using the specified client options  
#C The client options including the Broker URL  
#D Check to see if the client was able to connect
```

The `client` object handles all the low-level details involved in creating a connection to the Pulsar broker including automatic retries, and connection security if the Pulsar broker has TLS configured. Client instances are thread-safe and can be reused for creating and managing multiple producers and consumers. Once you have created a client, you can use it to create producers, consumers, and readers.

PULSAR PRODUCERS IN GO

As you can see from Listing 3.26, after you have created a client object you can use it to create a producer on any topic you choose. While there are several configuration options for a Pulsar producer that are described in the [online documentation](#), I wanted to highlight the delayed message delivery configuration that we used in this example, which allows us to defer delivery of the messages to the topic consumers for a specified amount of time.

Listing 3.26 Creating a Pulsar Producer in Go

```
import (
    "context"
    "fmt"
    "log"
    "time"

    "github.com/apache/pulsar-client-go/pulsar"      #A
)

func main() {
    ...          #B
    producer, err := client.CreateProducer(pulsar.ProducerOptions{
        Topic: topicName,      #C
    })

    ctx := context.Background()
    deliveryTime := (time.Minute * time.Duration(1)) +
        (time.Second * time.Duration(30))      #D

    for i := 0; i < 3; i++ {
        msg := pulsar.ProducerMessage{      #E
            Payload: []byte(fmt.Sprintf("Delayed-messageId-%d", i)),
            Key: "message-key",
            Properties: map[string]string{      #F
                "delayed": "90sec",
            },
            EventTime: time.Now(),      #G
            DeliverAfter: deliveryTime,      #H
        }

        messageID, err := producer.Send(ctx, &msg)      #I
        ...
    }
}
```

#A Import the pulsar client library

#B Code that creates the pulsar client

#C Create a new producer for the specified topic

#D Calculate the delivery time you want for the message

#E Create the message to send

```
#F The Go client supports providing both key and properties metadata
#G Provide the event timestamp metadata
#H Specify the delivery time for the message
#I Send the message
```

Delayed message delivery is useful if you do not want the message to be immediately processed, but rather processed at a future point in time. Consider the scenario where you receive a bunch of messages that contain new subscriptions to your company's newsletter that contains daily specials and promotions. Rather than immediately sending these customers the previous day's flier, you want to wait until the new edition is available. So, if your marketing team has committed to having a fresh version of the newsletter available every morning at 9:00 am, you can delay the message delivery until after 9:00 am to ensure the customers get the latest version of the newsletter.

PULSAR CONSUMERS

As we have seen, the consumer interface is used to listen on a specific topic and process the incoming messages. After a message has been successfully processed an acknowledgement should be sent back to the Broker to indicate that we are done processing the message within the subscription. This allows the Broker to know which message in the topic needs to be delivered to the next consumer on the subscription. In Go, you can create a Consumer by specifying a topic and a subscription as shown in Listing 3.27.

The message consumer shown in Listing 3.27 processes the messages in a synchronous manner because the `receive()` method that it is using to retrieve messages is blocking method, e.g., it waits indefinitely for a new message to arrive. Unlike, the previous two client libraries I have discussed, the Go client doesn't currently support asynchronous message consumption using the message listener pattern. Therefore, if you want to perform asynchronous processing, you will need to write all the threading logic yourself.

Listing 3.27 Creating a Pulsar Consumer in Go

```
import (
    "context"
    "fmt"
    "log"
    "time"

    "github.com/apache/pulsar-client-go/pulsar"      #A
)

func main() {
    ...      #B

    consumer, err := client.Subscribe(pulsar.ConsumerOptions{      #C
        Topic:          topicName,
        SubscriptionName: subscriptionName,
    })

    if err != nil {
        log.Fatal(err)
    }

    for {
```

```

        msg, err := consumer.Receive(context.Background())      #D
        if err != nil {
            log.Fatal(err)
            consumer.Nack(msg)      #E
        } else {
            fmt.Printf("Received message : %v\n", string(msg.Payload()))
        }

        consumer.Ack(msg)      #F
    }
}

```

#A Import the pulsar client library
#B Code that creates the pulsar client
#C Create a new consumer for the specified topic
#D Blocking call to receive incoming messages
#E Send a negative acknowledgment to have the message re-delivered
#F Acknowledge the message so it can be marked as processed.

The subscribe method will attempt to connect the consumer to the topic using the specified subscription, which may fail if the subscription already exists and it isn't one of the shared subscription types, e.g., you attempt to connect to an exclusive subscription that already has an active consumer. If you are connecting to the topic for the first time using the specified subscription name, then a subscription is created for you automatically. Whenever a new subscription is created, it is initially positioned at the end of the topic by default and consumers on that subscription will begin reading the first message created after the subscription was created. If you are connecting to a pre-existing subscription, then it will begin reading from the earliest unacknowledged message within the subscription, as we saw earlier in Figure 3.2.

As you can see from Listing 3.27, the Go library provides several different configuration options when specifying the subscription including the subscription type, starting position, etc. I highly recommend you visit the available [online documentation](#) for the Go client to get the most up-to-date listing of these options.

PULSAR READERS

The Go client also provides a reader interface that enables consumers to manage the position from which they will consume messages. When you connect to a topic using a reader, you must specify which message the reader will begin consuming messages from when it connects to the topic. If you choose to explicitly provide a message ID, then your application will be responsible for "knowing" this message ID in advance and should store that information in a persistent data store somewhere such as a database or cache.

Listing 3.28 Creating a Pulsar Reader in Go

```

import (
    "context"
    "fmt"
    "log"
    "time"

    "github.com/apache/pulsar-client-go/pulsar"      #A
)

```

```

func main() {
    ...
    #B

    reader, err := client.CreateReader(pulsar.ReaderOptions{
        Topic:          topicName,
        StartMessageID: pulsar.EarliestMessageID(),
    })
    #D

    for {
        msg, err := reader.Next(context.Background())
        #E
        if err != nil {
            log.Fatal(err)
        } else {
            fmt.Printf("Received message : %v\n", string(msg.Payload()))
        }
    }
}

#A Import the pulsar client library
#B Code that creates the pulsar client
#C Create a new reader for the specified topic
#D Start at the earliest message available
#E Read the next message

```

The code shown in Listing 3.28 connects to the topic and starts reading messages from the earliest available message and outputs their contents.

3.4 Advanced Administration

Pulsar acts as a black box from a Producer or Consumer perspective, i.e., you simply connect to the cluster to send and receive messages. While it is good to have the implementation details hidden from the end-user, this can be problematic when you need to troubleshoot issues with the message delivery itself. For instance, if your consumer isn't receiving any messages, how do you go about diagnosing the issue? Fortunately, the pulsar-admin CLI tool provides some tools that give you deeper insights into the inner workings of the Pulsar cluster.

3.4.1 Persistent Topic Metrics

Internally, Pulsar collects a lot of topic level metrics that can help you diagnose and troubleshoot issues between your producers and consumers such as your consumer not receiving any messages, or back-pressure when consumers cannot keep pace with your producers which would be reflected in the unacknowledged message count growing. You can access these topic statistics from the Pulsar admin CLI tool we used earlier to create the tenant, namespace, and topic by issuing the command shown in Listing 3.27.

Listing 3.27 Retrieving Pulsar topic statistics from the command-line

```
$docker exec -it pulsar /pulsar/bin/pulsar-admin topics stats
    persistent://manning/chapter03/example-topic      #A
{
    "msgRateIn" : 137.49506548471038,      #B
```

```

"msgThroughputIn" : 13741.401370605108,      #C
"msgRateOut" : 97.63210798236112,      #D
"msgThroughputOut" : 9716.05449008063,      #E
"bytesInCounter" : 1162174,
"msgInCounter" : 11538,      #F
"bytesOutCounter" : 150009,
"msgOutCounter" : 1500,      #G
"averageMsgSize" : 99.94105113636364,
"msgChunkPublished" : false,
"storageSize" : 1161944,      #H
"backlogSize" : 1161279,
"publishers" : [ {
    "msgRateIn" : 137.49506548471038,      #I
    "msgThroughputIn" : 13741.401370605108,
    "averageMsgSize" : 99.0,
    "chunkedMessageRate" : 0.0,
    "producerId" : 0,
    "metadata" : { },
    "producerName" : "standalone-12-6",
    "connectedSince" : "2020-09-07T20:44:45.514Z",      #J
    "clientVersion" : "2.6.1",
    "address" : "/172.17.0.1:40158"      #K
} ],
"subscriptions" : {      #L
    "my-sub" : {
        "msgRateOut" : 97.63210798236112,      #M
        "msgThroughputOut" : 9716.05449008063,
        "bytesOutCounter" : 150009,
        "msgOutCounter" : 1500,      #N
        "msgRateRedeliver" : 0.0,
        "chunkedMessageRate" : 0,
        "msgBacklog" : 9458,      #O
        "msgBacklogNoDelayed" : 9458,
        "blockedSubscriptionOnUnackedMsgs" : false,
        "msgDelayed" : 0,
        "unackedMessages" : 923,      #P
        "type" : "Shared",
        "msgRateExpired" : 0.0,
        "lastExpireTimestamp" : 0,
        "lastConsumedFlowTimestamp" : 1599511537220,
        "lastConsumedTimestamp" : 1599511537452,      #Q
        "lastAckedTimestamp" : 1599511545269,      #R
        "consumers" : [ {
            "msgRateOut" : 97.63210798236112,
            "msgThroughputOut" : 9716.05449008063,
            "bytesOutCounter" : 150009,
            "msgOutCounter" : 1500,
            "msgRateRedeliver" : 0.0,
            "chunkedMessageRate" : 0.0,
            "consumerName" : "5bf2b",
            "availablePermits" : 0,
            "unackedMessages" : 923,
            "avgMessagesPerEntry" : 6,
            "blockedConsumerOnUnackedMsgs" : false,      #S
            "lastAckedTimestamp" : 1599511545269,
            "lastConsumedTimestamp" : 1599511537452,
            "metadata" : { },
            "connectedSince" : "2020-09-07T20:44:45.512Z",
            "clientVersion" : "2.6.1",
        } ]
    }
}

```

```

        "address" : "/172.17.0.1:40160"      #T
    } ],
    "isDurable" : true,
    "isReplicated" : false
},
"example-sub" : {
    "msgRateOut" : 0.0,      #U
    "msgThroughputOut" : 0.0,
    "bytesOutCounter" : 0,
    "msgOutCounter" : 0,
    "msgRateRedeliver" : 0.0,
    "chuckedMessageRate" : 0,
    "msgBacklog" : 11528,      #V
    "msgBacklogNoDelayed" : 11528,
    "blockedSubscriptionOnUnackedMsgs" : false,
    "msgDelayed" : 0,
    "unackedMessages" : 0,
    "type" : "Exclusive",
    "msgRateExpired" : 0.0,
    "lastExpireTimestamp" : 0,
    "lastConsumedFlowTimestamp" : 1599509925751,
    "lastConsumedTimestamp" : 0,
    "lastAckedTimestamp" : 0,
    "consumers" : [ ],
    "isDurable" : true,
    "isReplicated" : false
}
},
"replication" : { },
"deduplicationStatus" : "Disabled"
}

```

#A The name of the topic we want statistics from
#B The sum of all local and replication publishers' rates in messages per second
#C The sum of all local and replication publishers' rates in bytes per second
#D The sum of all local and replication consumers' dispatch rates in messages per second
#E The sum of all local and replication consumers' dispatch rates in bytes per second
#F The total number of messages published to the topic
#G The total number of messages consumed from the topic
#H The total amount of disk space used to store the topic messages in bytes
#I Total rate of messages published by the publisher in messages per second
#J Timestamp of when the publisher first connected to the topic
#K The IP address of the producer
#L A list of all the subscriptions for the topic
#M Total rate of messages delivered on this subscription in bytes per second
#N Total number of messages delivered on this subscription
#O Number of messages in the subscription backlog that haven't been delivered yet.
#P Number of messages that have been delivered but haven't been acknowledge yet.
#Q Timestamp of when the last message was consumed on this subscription
#R Timestamp of when the last message acknowledgement was received on this subscription
#S Whether or not the consumer is blocked due to too many unacknowledged messages
#T The IP address of the consumer
#U Indicative of a subscription without any active consumers.
#V Number of messages in the subscription backlog.

As you can see, pulsar collects an extensive set of metrics for each persistent topic which can be very useful when attempting to diagnose an issue. The metrics returned include the

connected producers and consumers along with all the message production and consumption rates, message backlog, and subscriptions. Therefore, if you are trying to determine why a particular consumer isn't receiving messages, you can verify that the consumer is connected, and look at the message consumption rate for its corresponding subscription.

All these metrics are published to Prometheus by default and can be easily viewed through a Graphana dashboard that comes bundled with the Pulsar Kubernetes deployment defined in a Helm chart inside the open-source project. You can also configure any observability tool that works with Prometheus as well.

3.4.2 Message Inspection

Sometimes you may want to view the contents of a particular message or group of messages within a Pulsar topic. Consider the scenario where one of the message producers changes the output format of its messages by encrypting the message contents. Consumers that are currently subscribed to the topic would suddenly start encountering exceptions when they attempt to process these encrypted contents which would result in the messages not getting acknowledged. Eventually these messages would accumulate on the topic since they never get acknowledged. If the change to the producer code was not coordinated with you, then you will be unaware of the underlying issue.

Fortunately, you can use the ***peek-messages*** command of the pulsar-admin CLI tool to view the raw bytes of the messages inside a given subscription as shown in Listing 3.28, which shows the syntax for peeking at the last 10 messages for the subscription "example-sub" on the persistent://manning/chapter03/example-topic.

Listing 3.28 Peeking at Messages inside Pulsar

```
$ docker exec -it pulsar /pulsar/bin/pulsar-admin \
Topic peek-messages \
--count 10 \      #A
--subscription example-sub \
persistent://manning/chapter03/example-topic

Batch Message ID: 19460:9:0      #B
Tenants:
{
  "X-Pulsar-num-batch-message" : "1",
  "publish-time" : "2020-09-07T20:20:13.136Z"      #C
}
+-----+
| 0 1 2 3 4 5 6 7 8 9 a b c d e f |
+-----+
|00000000| 7b 69 64 3a 20 31 30 2c 20 74 69 6d 65 3a 20 4d |{id: 10, time: M|
|00000010| 6f 6e 20 53 65 70 20 30 37 20 31 33 3a 32 30 3a |on Sep 07 13:20:|
|00000020| 31 33 20 50 44 54 20 32 30 32 30 7d                |13 PDT 2020} |
+-----+          +-----+          +-----+          +-----+
#D
```

#A Request the last ten messages

#A Request the last
#B The message ID

#C The time the message was published by the producer

#D The message contents in raw bytes

As you can see, the peek-messages command provides a lot of details about the message, including the message ID, publish time, and the message contents as raw bytes (and as a String). This information should make it easier to determine the issue with the message contents.

3.5 Summary

- Docker is an open-source container framework that allows you to bundle entire applications into a single image and publish them for reuse.
- There is a completely self-contained Docker image of Pulsar that you can use to run Pulsar on your machine for development.
- Pulsar provides command-line tools that can be used to administer tenants, namespaces, and topics including creating, listing, and deleting them.
- Pulsar provides client libraries for several popular programming languages, including Java, Python, and Go that allow you to create Pulsar producers, consumers, and readers.
- You can use Pulsar's command-line tools to retrieve topic statistics that are useful for monitoring and troubleshooting.

4

Pulsar Functions

This chapter covers

- Introduction to the Pulsar Functions framework
- The Pulsar Functions programming model and API
- Writing your first Pulsar Function in Java
- Configuring, submitting, and monitoring a Pulsar Function

In our previous chapter, we started to see how you can work with Pulsar using some of the various client libraries. In this chapter we will look at a stream native processing engine known as Pulsar Functions that makes the development of Pulsar-based applications much simpler. This lightweight processing framework automatically handles a lot of the boilerplate coding required to setup Pulsar consumers and producers, allowing you to focus on the processing logic itself rather than the consumption and processing of the messages.

4.1 Stream Processing

While there isn't an official definition, the term stream processing generally refers to the processing of unbounded datasets that stream in continuously from some source system. There are several datasets that occur naturally as continuous streams: sensor events, user activity on a website, financial trades, etc. that can be processed in this manner.

Prior to stream processing, these datasets had to first be stored into a database, file system, or other persistent storage before it could be accessed. Often, there was an additional data processing phase required to extract the information, transform it into correct format, and load the data into these systems. Only after the ETL process was completed was the data ready to be analyzed using traditional SQL-based tools, etc. As you can imagine, there was a significant latency between the time an event occurred and when it was available for analysis. The goal of stream processing is to minimize that latency so that critical business decisions can be made against the most recent data. There are three basic

approaches to processing these datasets, batch processing, micro-batching, and streaming native processing and each take different approaches on how and when to process these endless datasets.

4.1.1 Traditional Batching

Historically, the vast majority of data processing frameworks have been designed for batch processing. Traditional data warehouses, Hadoop, and Spark are just a few common examples of systems that process large datasets in batches. Data is often fed into these systems via long-running and complex ETL pipelines, that cleanse, prepare, and format the incoming data for consumption. Messaging systems often serve as little more than intermediate buffers that store and route the data between the various processing stages of the pipeline.

These long-running data ingestion pipelines are often implemented using stream processing engines such as Apache Spark, or Flink that were designed to process large datasets efficiently by performing the processing in parallel. Newly arriving data elements are collected and then processed together at some point in time in the future as batch. To maximize the throughput of these frameworks, the accumulation would take place over very large time intervals (hours) or until a certain amount of data (10s GBs) had been collected, which introduced an artificial delay in the data processing pipeline.

4.1.2 Micro-Batching

One technique that was introduced to address the processing latency that plagued these traditional batch processing engines was to dramatically reduce either the batch size or the processing interval. In micro-batch processing newly arriving data elements are still collected batches as shown in Figure 4.1, but the size of the batches is dramatically reduced by adjusting the time interval to a few seconds, etc.

Even though the processing may occur more frequently, the data is still processed one batch at a time, so it is often referred to as micro-batching and is used by such processing frameworks as Spark Streaming.

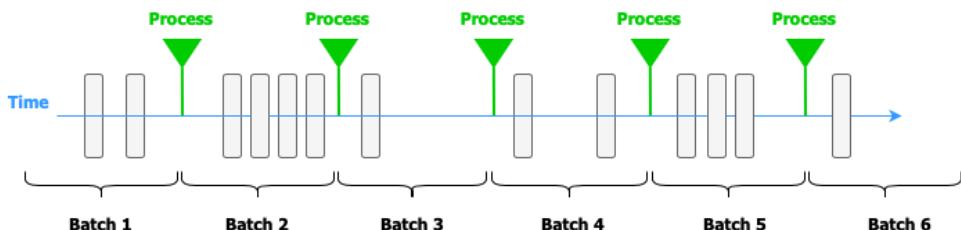


Figure 4.1 With batch processing, the message processing occurs at pre-determined intervals and follows a consistent cadence.

While this approach does decrease the processing latency between when a data element arrives and when it was processed, it still introduces artificial delays into the process that

compound as the complexity of the data pipeline increases. Consequently, even micro-batch processing applications cannot rely on consistent response times and need to account for delays between when the data arrived and when it has been processed.

This makes micro-batch processing more appropriate for use cases that do not require having the most recent data and can tolerate slower response times, whereas stream native processing is better suited for use cases that require near real-time responsiveness such as fraud detection, real-time pricing, and system monitoring.

4.1.3 Stream Native Processing

With stream native processing, each new piece of data is processed as soon as it arrives, as illustrated below in Figure 4.2. Unlike batch processing, there are no arbitrary processing intervals, and each individual data element is processed separately.

Although it may seem as though the differences between stream processing and micro-batch are just a matter of timing, there are implications for both the data processing systems and the applications that rely on them. The business value of data decreases rapidly after it is created, particularly in use cases such as fraud prevention or anomaly detection. The high-volume, high-velocity datasets used to feed these use cases often contain valuable, but perishable, insights that must be acted upon immediately. A fraudulent business transaction, such as transferring money or downloading licensed software must be identified and acted upon before the transaction completes, otherwise it will be too late to prevent the thief from obtaining the funds illegally.

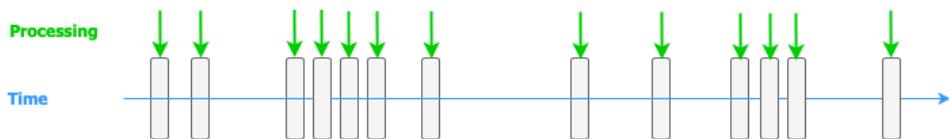


Figure 4.2 With stream processing, the processing is triggered by the arrival of each message, so the processing cadence is irregular and unpredictable.

To maximize the value of their data for these use cases, developers must fundamentally change their approach to processing real-time data by focusing on reducing the processing latency introduced from traditional batch processing frameworks and utilize a more reactive approach such as stream processing.

4.2 What are Pulsar Functions?

Included with Apache Pulsar is a lightweight computing engine named “Pulsar Functions” which allows developers to deploy a simple function implementation in Java, Python, or Golang. This feature allows users to enjoy the benefits of serverless computing, like AWS Lambda computing within an open-source messaging platform without being tied to a cloud providers proprietary API.

Pulsar Functions allow you to apply processing logic to data as it is routed through the messaging system itself. These lightweight compute processes execute natively within the

Pulsar messaging system itself as close to the message as you can be and without the need for another processing framework such as Kafka Streams. Unlike other messaging systems which act as “dumb pipes” for moving data from system to system, Pulsar Functions provides the capabilities to perform simple computations on the messages before they are routed to the consumers.

Pulsar Functions consumes messages from one or more Pulsar topics, apply a user-supplied function (processing logic) to each incoming message, and publishes the results to one or more Pulsar topics as shown in Figure 4.3.

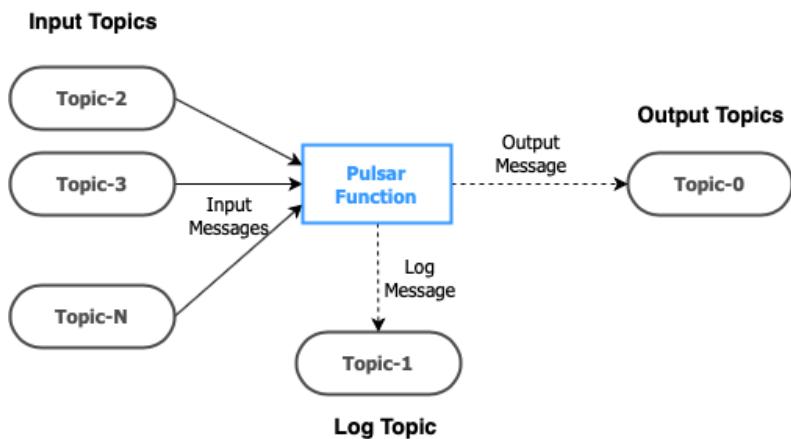


Figure 4.3: Pulsar Functions execute user-defined code on data published to Pulsar topics.

Pulsar Functions can be best characterized as Lambda-style functions that are specifically designed to use Pulsar as the underlying message bus. This is because they take several design cues from the popular AWS Lambda framework that allows you to run code without provisioning or managing servers to host the code. Hence the common term for this programming model is “serverless”.

The Pulsar function framework allows users to develop simple self-contained pieces of code and then deploy them with a simple REST call. Pulsar takes care of the underlying details required to run your code, including creating the Pulsar consumer and producers for the Function’s input and output topics, etc. Now developers can focus on the business logic itself and not have to worry about the boilerplate code necessary to send messages with Pulsar. in short, the Pulsar Functions framework provides a ready-made computing infrastructure on your existing Pulsar cluster.

4.2.1 Programming Model

The programming model behind Pulsar Functions is very straightforward. Pulsar Functions receive messages from one or more **input** topics, and every time a message is published to the topic, the function code is executed. Upon being triggered, the function code executes its processing logic upon the incoming message and writes its (optional) output to an **output**

topic. Although all Functions are required to have an input topic, they are not strictly required to produce any output to an output topic.

It is possible to have the output topic of one Pulsar Function be the input topic of another, allowing us to effectively create a directly acyclic graph (DAG) of Pulsar Functions, as shown in Figure 4.4. In such a graph, each edge represents a flow of data and each vertex represents a Pulsar Functions that applies the user-defined logic to process the data.

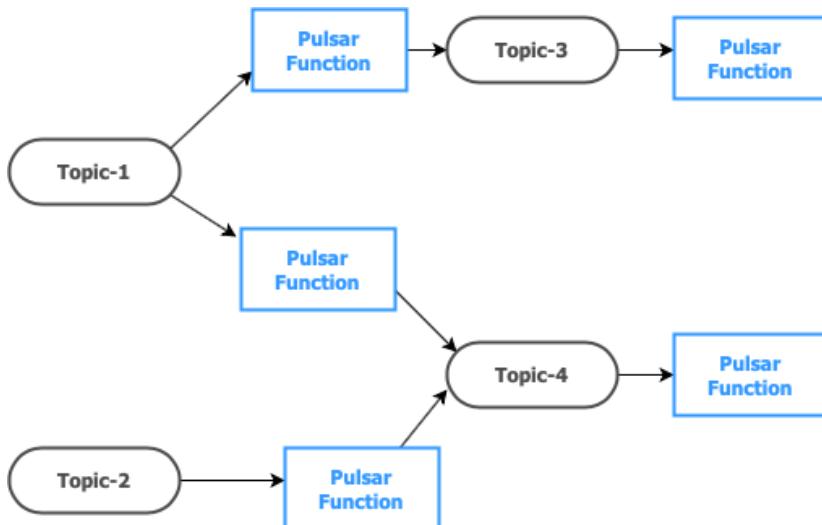


Figure 4.4: Pulsar Functions can be logically structured into a processing network.

The combinations of Pulsar Functions into these DAGs are endless, and it is possible to write an application that is entirely composed of Pulsar Functions structured as a DAG if you so choose.

4.3 Developing Pulsar Functions

Pulsar Functions can currently be written in Java, Python, and Go. Therefore, if you are already familiar with either of these popular languages, you will be able to develop Pulsar Functions relatively quickly.

When writing Pulsar Functions in Java you have two choices, you can either write "native-language" functions by implementing the `java.util.Function` interface, or you can use the Pulsar Function SDK to access some additional features. To write Pulsar Functions in Java, you'll need to install the proper dependencies and package your function along with all its dependencies as either a "fat" JAR or a NAR file.

4.3.1 Language Native Functions

Pulsar supports what is commonly referred to as “language-native” Functions which means NO Pulsar-specific libraries or special dependencies are required. The benefit of language-native functions is that they don’t have any dependencies beyond what’s already available in the programming language itself, which makes them extremely lightweight and easy to develop. Currently language native Functions can only be developed using Java or Python. Golang support for this feature is not yet available.

JAVA LANGUAGE NATIVE FUNCTIONS

In order for a piece of Java code to be used as a “language-native” function, it must implement the `java.util.Function` interface, which has just a single `apply` method as shown in Listing 4.1.

Listing 4.1 Java Native Function

```
import java.util.Function;

public class EchoFunction implements Function<String, String> {    #A

    public String apply(String input) {    #B
        return input;
    }
}
```

#A Specifies that the input topic content will be a String, and that we will return a String

#B The only method defined in the Function interface, which is executed when a message is received.

While this simplistic function merely echoes back any String value it receives, it does demonstrate just how easy it is to develop a Function using only features of the Java language itself. Any sort of complex logic can be included inside the `apply` method to provide more robust stream processing capabilities.

PYTHON LANGUAGE NATIVE FUNCTIONS

For a piece of Python code to be used as a “language-native” function, it must have a method named `process` like the functions shown in Listing 4.2 that merely appends an exclamation point to any String value it receives.

Listing 4.2 Python Native Function

```
def process(input):    #A
    return "{}!".format(input)    #B
```

#A The method that gets called when a message arrives.

#B Return the provided input with an exclamation point appended to the end.

As you can see, the language-native approach provides a clean, API-free way of writing Pulsar Functions, and is ideal for the development of simple stateless functions.

4.3.2 The Pulsar SDK

Another option is to develop your functions using the Pulsar Functions SDK, which leverages Pulsar-specific libraries that provide a range of functionality not available in the “native” interfaces, such as state management or user configuration. Additional capabilities and information can be accessed through a `Context` object that is defined inside the SDK including:

- The name, version, and ID of the Pulsar Function.
- The message ID of each message.
- The name of the topic on which the message was sent.
- The names of all input topics as well as the output topic associated with the function.
- The tenant and namespace associated with the function.
- The logger object used by the function, which can be used to write log messages.
- Access to arbitrary user config values supplied via the CLI.
- An interface for recording metrics.

An implementation of the Pulsar SDK is available for Java, Python, and Golang and each specifies a functional interface that includes the `Context` object as a parameter that is populated and provided by the Pulsar Function runtime environment.

JAVA SDK FUNCTIONS

To get started developing Pulsar Functions using the Java SDK, you’ll need to add a dependency on the `pulsar-functions-api` artifact to your project as shown in Listing 4.2.

Listing 4.2 Adding Pulsar SDK Dependencies to your pom.xml file

```
<properties>
    <pulsar.version>2.7.2</pulsar.version>
</properties>

...
<dependency>
    <groupId>org.apache.pulsar</groupId>
    <artifactId>pulsar-functions-api</artifactId>
    <version>${pulsar.version}</version>
</dependency>
```

When you are developing a Pulsar Function that is based upon the SDK then the function should implement the `org.apache.pulsar.functions.api.Function` interface. As you can see from Listing 4.3, this interface specifies only one method that you need to implement called `process`.

Listing 4.3 The Pulsar SDK Function Interface Definition

```
@FunctionalInterface
public interface Function<I, O> {
    O process(I input, Context context) throws Exception;
}
```

The `process` method is called at least once, depending on the processing guarantees you specify for the function, for every message that is published to the configured input topic of the function. The incoming input bytes are serialized to the input type `I` for JSON based messages as well as simple Java types such as `String`, `Integer`, `Float`, etc. If these types do not meet your needs, you can also use your own types as long as you provide your own implementation of the `org.apache.pulsar.functions.api.SerDe` interface for the type as well or you can register the incoming message type in the Pulsar Schema Registry, which I will cover in greater detail in Chapter 7. An implementation of the Echo function that demonstrates several different features of the SDK such as logging and recording metrics is shown in Listing 4.4.

Listing 4.4 Pulsar SDK Function in Java

```
import java.util.stream.Collectors;
import org.apache.pulsar.functions.api.Context;
import org.apache.pulsar.functions.api.Function;      #A
import org.slf4j.Logger;

public class EchoFunction implements Function<String, String> {

    public String process(String input, Context ctx) {      #B
        Logger LOG = ctx.getLogger();      #C
        String inputTopics =
            ctx.getInputTopics().stream()
                .collect(Collectors.joining(", "));      #D

        String functionName = ctx.getFunctionName();      #E

        String logMessage =
            String.format("A message with a value of \"%s\""
                + "has arrived on one of the following topics: %s\n",
                input, inputTopics);

        LOG.info(logMessage);      #F
        String metricName =
            String.format("function-%s-messages-received", functionName);

        ctx.recordMetric(metricName, 1);      #G
        return input;
    }
}
```

#A The class must implement the Pulsar Function interface.

#B The interface defines a single method with 2 parameters, including a Context object.

#C We use the Context object to access the LOGGER object.

#D We use the Context object to get the list of input topics

#E We use the Context object to get the function name

#F We generate a log message

#G We record a user-defined metric.

The Java SDK's Context object enables you to access key/value pairs provided to the Pulsar Function via the command line (as JSON). This feature allows us to write generic functions that can be used multiple times, but with a slightly different configuration. For instance, let's say you want to write a function that filters events based on a user-defined regular

expression. When an event arrives, the contents are compared to the configured regex, and those entries that contain the match the provided pattern are returned, and all others are ignored. Such a function could be useful if you want to verify the format of the incoming data before you begin processing it. An example of such a function that accesses the regular expression from the key/value pairs in the Context object is shown in Listing 4.5.

Listing 4.5 User Configured Pulsar Function in Java

```
import java.util.regex.Pattern;
import org.apache.pulsar.functions.api.Context;
import org.apache.pulsar.functions.api.Function;

public class RegexMatcherFunction implements Function<String, String> {

    public static final String REGEX_CONFIG = "regex-pattern";

    @Override
    public String process(String input, Context ctx) throws Exception {
        Optional<Object> config =
            ctx.getUserConfigValue(REGEX_CONFIG);      #A

        if (config.isPresent() && config.get().getClass()
            .getName().equals(String.class.getName())) {    #B

            Pattern pattern = Pattern.compile(config.get().toString());
            if (pattern.matcher(input).matches()) {      #C
                String metricName =
                    String.format("%s-regex-matches",ctx.getFunctionName());

                ctx.recordMetric(metricName, 1);
                return input;
            }
        }
        return null;      #D
    }
}
```

#A Retrieve the regex pattern from the user provided configs

#B If a regex String was provided then compile the regex

#C if the input matches the regex, allow it to pass

#D Otherwise return null

Pulsar Functions can publish results to an output topic, but this isn't required. You can also have functions that filter don't always return a value, such as the Function in Listing 4.5 that filters out non-matching input. In such a scenario, you can simply return a value of `null` from the function.

PYTHON SDK FUNCTIONS

To get started developing Pulsar Functions using the Python SDK, you'll need to add the pulsar client dependency to your Python installation. The latest version of the Pulsar client library can be easily installed using the pip package manager using the commands shown in Listing 4.6. Once this is installed on your local development environment, you will be able to start developing Pulsar Functions in Python that leverage the SDK.

Listing 4.6 Adding Pulsar SDK Dependencies to your Python Environment

```
pip3 install pulsar-client==2.6.3 -user    #A  
pip3 list    #B  
  
Package      Version  
----- -----  
...  
pulsar-client 2.6.3    #C  
  
#A Install the pulsar client  
#B List all the packages  
#C Confirm that the correct version of the Pulsar client has been installed
```

Let's look a Python-based implementation of the Echo function to demonstrate some of the SDK capabilities, shown in Listing 4.7.

Listing 4.7 Pulsar SDK Function in Python

```
from pulsar import Function  
  
class EchoFunction(Function):  
    def __init__(self):  
        pass  
  
    def process(self, input, context):    #A  
        logger = context.get_logger()    #B  
        evtTime = context.get_message_eventtime()  
        msgKey = context.get_message_key();  #C  
  
        logger.info("""A message with a value of {0}, a key of {1},  
        and an event time of {2} was received"""  
        .format(input, msgKey, evtTime))  
  
        metricName = """function-%s-  
        messages-received""".format(context.get_function_name())  
        context.record_metric(metricName, 1)    #D  
  
        return input    #E  
  
#A The function definition required by the Pulsar SDK  
#B The Python SDK provides access to the logger  
#C The Python SDK provides access to message metadata  
#D The Python SDK supports metrics  
#E Echo back the original input value
```

The Python SDK's Context object provides nearly all the same capabilities as the Java SDK, with two notable exceptions: The first is that as of version 2.6.0, Python-based Pulsar Functions do not support schemas, which I will discuss in greater detail in Chapter 7, but essentially this means that Python function is responsible for the serialization and deserialization of the message bytes into the expected format. The second capability that is not present in Python-based Pulsar Functions is access to the Pulsar Admin API which, as I discussed in Chapter 3, is only available in Java.

GOLANG SDK FUNCTIONS

To get started developing Pulsar Functions using the Golang SDK, you'll need to add the pulsar client dependency to your Golang installation. The latest version of the pulsar client library can be installed using the following command: `go get -u "github.com/apache/pulsar-client-go/pulsar"`. Let's look at a Golang-based implementation of the Echo function that we used earlier to demonstrate some of the SDK capabilities, shown in Listing 4.8.

Listing 4.8 Pulsar SDK Function in Go

```
package main

import (
    "context"
    "fmt"

    "github.com/apache/pulsar/pulsar-function-go/pf"      #A
    log "github.com/apache/pulsar/pulsar-function-go/logutil"    #B
)

func echoFunc(ctx context.Context, in []byte) []byte {    #C
    if fc, ok := pf.FromContext(ctx); ok {
        log.Infof("This input has a length of: %d", len(in))    #D

        fmt.Printf("Fully-qualified function name is:%s\\%s\\%s\n",    #E
            fc.GetFuncTenant(), fc.GetFuncNamespace(), fc.GetFuncName())
    }
    return in    #F
}

func main() {
    pf.Start(echoFunc)    #G
}
```

#A import the SDK library
#B Import the function logger library
#C The function code with the correct method signature
#D The Golang SDK provides access to the logger
#E The Golang SDK provides access to function metadata
#F Echo back the original input value
#G Register the echoFunc with the Pulsar Functions framework

When writing Golang-based Functions, remember that you need to provide the name of function you wish to perform the actual logic to the `pf.Start()` method inside the `main()` method call as shown in Listing 4.8. This registers the function with the Pulsar Functions framework and ensures that the specified function is the one that is invoked when a new message arrives. In this case, we named used the `echoFunc` function, but it can be named anything provided that the method signature matches any of the supported one shown in Listing 4.9. Any other function signatures will not be accepted and consequently no processing logic will be executed inside your Golang function.

Listing 4.9 Supported Method Signatures in Go

```
func ()  
func () error  
func (input) error  
func () (output, error)  
func (input) (output, error)  
func (context.Context) error  
func (context.Context, input) error  
func (context.Context) (output, error)  
func (context.Context, input) (output, error)
```

There are currently some limitations when it comes to using the SDK to develop Golang-based Pulsar Functions, but this is subject to change as the project matures, so I highly recommend checking the most recent version of the online documentation for the latest capabilities. However, as of the time of this book, Golang functions do not support the recording of function-level metrics, e.g., there isn't a `recordMetric` method defined inside the `Context` object of the Golang SDK. Furthermore, you cannot implement stateful Functions using Golang at this time.

4.3.3 Stateful Functions

Stateful functions utilize information gathered from previous messages that it has processed to generate their output. One such application would be a Function that receives temperature reading events from an IoT sensor and calculates the average temperature of the sensor. Providing this value would require us to calculate and store a running average of the previous temperature readings.

When using the Pulsar SDK to develop your Functions, regardless of which of the three supported languages you are using, the the second parameter in the `process` method is a `Context` object that is automatically provided by the Pulsar Function framework.

If you are using Java or Python, then `Context` object's API also provides two separate mechanisms for retaining information between successive calls to a Pulsar Function. The first mechanism that the `Context` API provides is map interface for storing and retrieving key/value pairs via its `putState` and `getState` methods. These methods act like any other map interface you are familiar with and allow you store and retrieve values of any type using `String` values as the keys.

The other state mechanism provided by the `Context` object, are counters, which only allow you to retain numeric values using `Strings` as the keys. These are essentially a specialized version of the key/value mapping functionally that not only limits itself to the storage of only `long` values, but also restricts the operation on these values to addition or subtraction via incrementing with a negative number.

Let's take a Function that receives temperature reading events from an IoT sensor and calculates the average temperature of the sensor as an example to show how you would utilize state inside a Pulsar Function. The Function shown in Listing 4.10, receives a sensor reading and compares it to the average temperature reading it has calculated from the previous reading to determine if it should trigger and alarm of some sort.

Listing 4.10 Average Sensor Reading Function

```
public class AvgSensorReadingFunction implements
    Function<Double, Void> {      #A

    private static final String VALUES_KEY = "VALUES";

    @Override
    public Void process(Double input, Context ctx) throws Exception {
        CircularFifoQueue<Double> values = getValues(ctx);      #B

        if (Math.abs(input - getAverage(values)) > 10.0) {
            // trigger an alarm.      #C
        }

        values.add(input);      #D
        ctx.putState(VALUES_KEY, serialize(values));      #E
        return null;
    }

    private Double getAverage(CircularFifoQueue<Double> values) {
        return StreamSupport.stream(valuesspliterator(), false)
            .collect(Collectors.summingDouble(Double::doubleValue))
            / values.size();      #F
    }

    private CircularFifoQueue<Double> getValues(Context ctx) {
        if (ctx.getState(VALUES_KEY) == null) {
            return new CircularFifoQueue<Double>(100);      #G
        } else {
            return deserialize(ctx.getState(VALUES_KEY));      #H
        }
    }

    . . .
}
```

#A The function takes in a double and doesn't produce an output message

#B We deserialize the Java object used to store the previous sensor readings

#C If the current reading is significantly different then we generate an alert

#D We add the current value to the list of observed values.

#E We store the updated Java object in the state store

#F We use the Streams API to calculate the average

#G Instantiate the Java object if none exists in the state store

#H Convert the bytes in the state store into the Java object we need.

There are a few points I want to highlight from the Function in Listing 4.9. The first thing you may notice is that the return type of Function is defined as Void, this means that the Function does not produce an output value that is published to an output topic. Another point I want to highlight is the fact that the Function relies upon Java Serialization to store and retrieve a list of the last 100 values (sensor readings) it has received. It relies on a third-party library implementation of a FIFO queue to retain the 100 most recent values to compute the average before comparing it to the most recent sensor reading. If that value significantly deviates from the average, then an alert is raised. Finally, the most recent reading is added to the FIFO queue which is then serialized and written to the state store.

When the Functions runs again, it will retrieve the bytes of the FIFO queue and deserialize them back into the Java object and use it to calculate the average again, etc. This process repeats indefinitely and only retains the most recent values for comparison against the trend, e.g., the moving average of the sensor readings. This approach is very different from the windowing capability provided by Pulsar Functions that is discussed in Chapter 12. In short, the windowing capability provided by the Pulsar Function framework permits the collection of multiple inputs before executing the Function method, based on either time or a fixed count. Once the window is filled, the Function is provided the entire list of inputs at one time whereas the Function shown in Listing 4.6 is provided the values one at a time, and most maintain the previous values inside its state.

So, you might be asking yourself why you wouldn't just use Pulsar's built-in windowing for our use case? In our case we want to react to every individual reading as it becomes available rather than waiting for a list of 500 readings. This allows us to detect an issue much sooner.

Now let's wrap up our discussion on stateful functions by looking at the counter interface provided by the Context API. A good example of how and when to use this functionality provided would be a WordCount Function that stores the number of each individual word using the counter methods provided by the Context object API as shown in Listing 4.11.

Listing 4.11 WordCount Function using Stateful Counters

```
package com.manning.pulsar.chapter4.functions.sdk;

import java.util.Arrays;
import java.util.List;

import org.apache.pulsar.functions.api.Context;
import org.apache.pulsar.functions.api.Function;

public class WordCountFunction implements Function<String, Integer> {
    @Override
    public Integer process(String input, Context context) throws Exception {
        List<String> words = Arrays.asList(input.split("\\."));
        words.forEach(word -> context.incrCounter(word, 1));
        return Integer.valueOf(words.size());
    }
}
```

The logic of the function is straightforward, it first splits the incoming String object into multiple words using a regex pattern and then for each word generated from the split increments the corresponding counter by one. This function is a good candidate for exactly-once processing semantics to ensure an accurate result. If you were to use at-least-once processing semantics instead, then you could potentially end up processing the same message more than once in a failure scenario which would result in double counting of multiple words.

4.4 Testing Pulsar Functions

In this section I will walk you through the process of developing and testing your first Pulsar Function. Let's use with the `KeywordFilterFunction` shown in Listing 4.12 to demonstrate the software development lifecycle for a Pulsar Function. This function takes in a user provided keyword and filters out any input String that does not contain that keyword. An example application of this function would be to scan a Twitter feed for tweets related to a particular topic or containing a certain phrase.

Listing 4.12 KeywordFilterFunction

```
package com.manning.pulsar.chapter4.functions.sdk;

import java.util.Arrays;
import java.util.List;
import java.util.Optional;
import org.apache.pulsar.functions.api.Context;
import org.apache.pulsar.functions.api.Function;

public class KeywordFilterFunction
    implements Function<String, String> {    #A

    public static final String KEYWORD_CONFIG = "keyword";
    public static final String IGNORE_CONFIG = "ignore-case";

    @Override
    public String process(String input, Context ctx) {
        Optional<Object> keyword =
            ctx.getUserConfigValue(KEYWORD_CONFIG);    #B
        Optional<Object> ignoreCfg =
            ctx.getUserConfigValue(IGNORE_CONFIG);    #C

        boolean ignoreCase = ignoreCfg.isPresent() ?
            (boolean) ignoreConfig.get(): false;

        List<String> words = Arrays.asList(input.split("\\s"));    #D

        if (!keyword.isPresent()) {
            return null;    #E
        } else if (ignoreCase && words.stream().anyMatch(    #F
            s -> s.equalsIgnoreCase((String) keyword.get()))) {
            return input;
        } else if (words.contains(keyword.get())) {    #G
            return input;
        }
        return null;
    }
}
```

#A The function takes in a String and returns a String

#B Get the keyword from the context object

#C Get the ignore-case setting from the context object

#D Split the input String into individual words

#E Without a keyword, nothing can match

#F Evaluate each word ignoring case

#G Check for an exact match

While this code is simplistic, I will walk through the testing process you would typically use when developing a function for production use. Since this is just plain Java code, we can leverage any of the existing unit testing frameworks such as Junit or TestNG to test the function logic.

4.4.1 Unit Testing

The first step would be to write a suite of unit tests that test some of the more common scenarios to validate that the logic is correct and produces accurate results for various sentences that we send it. Since this code uses the Pulsar SDK API, we will need to use a Mocking library, such as Mockito to mock the `Context` object as shown in Listing 4.13.

Listing 4.13 KeywordFilterFunction Unit Tests

```
package com.manning.pulsar.chapter4.functions.sdk;

import static org.mockito.Mockito.*;
import static org.junit.Assert.*;

public class KeywordFilterFunctionTests {
    private KeywordFilterFunction function = new KeywordFilterFunction();

    @Mock
    private Context mockedCtx;

    @Before
    public final void setUp() {
        MockitoAnnotations.initMocks(this);
    }

    @Test
    public final void containsKeywordTest() throws Exception {
        when(mockedCtx.getUserConfigValue(
            KeywordFilterFunction.KEYWORD_CONFIG))
            .thenReturn(Optional.of("dog"));      #A

        String sentence = "The brown fox jumped over the lazy dog";
        String result = function.process(sentence, mockedCtx);
        assertNotNull(result);      #B
        assertEquals(sentence, result);
    }

    @Test
    public final void doesNotContainKeywordTest() throws Exception {
        when(mockedCtx.getUserConfigValue(
            KeywordFilterFunction.KEYWORD_CONFIG))
            .thenReturn(Optional.of("cat"));      #C

        String sentence = "It was the best of times, it was the worst of times";
        String result = function.process(sentence, mockedCtx);
        assertNull(result);      #D
    }

    @Test
    public final void ignoreCaseTest() {
        when(mockedCtx.getUserConfigValue(
```

```

        KeywordFilterFunction.KEYWORD_CONFIG))
    .thenReturn(Optional.of("RED"));      #E

    when(mockedCtx.getUserConfigValue(
        KeywordFilterFunction.IGNORE_CONFIG)
    .thenReturn(Optional.of(Boolean.TRUE));     #F

    String sentence = "Everyone watched the red sports car drive off.";
    String result = function.process(sentence, mockedCtx);
    assertNotNull(result);      #G
    assertEquals(sentence, result);
}
}

#A Configure the keyword to be "dog"
#B We expect the sentence to be returned since it contained the keyword
#C Configure the keyword to be "cat"
#D We don't expect the sentence to be returned since it did not contain the keyword
#E Configure the keyword to be "RED"
#F Configure the function to ignore case when filtering on the keyword
#G We expect the sentence to be returned since it contained a lower-case version of the keyword

```

As you can see these unit tests cover the very basic functionality of Function and rely on the use of a Mock object for the Pulsar context object. This type of test suite is just like one you would write to test any Java class that wasn't a Pulsar Function.

4.4.2 Integration Testing

After we are satisfied with our unit testing results, we will want to see how the Pulsar Function will perform on a Pulsar cluster. The easiest way to test a Pulsar Function is to start a Pulsar server and run the Pulsar Function locally using the LocalRunner helper class. In this mode, the function runs as a standalone process on the machine it is submitted from. This option is best when you are developing and testing your Functions as it allows you to attach a debugger to the Function process on the local machine as shown in Figure 4.5.

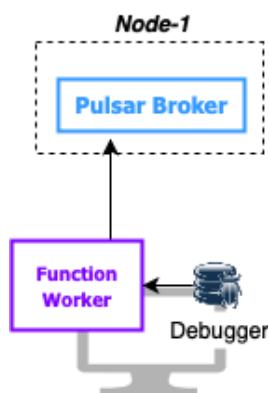


Figure 4.5: When you run a Pulsar Function using the LocalRunner, the function runs on the local machine, allowing you to attach a debugger and step through the code.

To use the LocalRunner, you must first add a few dependencies to your maven project as shown in Listing 4.14, which brings in the LocalRunner class that is used to test the Function against a running Pulsar cluster.

Listing 4.14 Including the LocalRunner Dependencies

```
<dependencies>
    ...
    <dependency>
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-core</artifactId>
        <version>${jackson.version}</version>
    </dependency>
    <dependency>
        <groupId>org.apache.pulsar</groupId>
        <artifactId>pulsar-functions-local-runner-original</artifactId>
        <version>${pulsar.version}</version>
    </dependency>
</dependencies>
```

Next, we need to write a class to configure our and launch the LocalRunner, as shown in Listing 4.15. As you can see this code must first configure the Pulsar Function to execute on the LocalRunner and specifies the address of the actual Pulsar cluster instance that will be used for the testing.

Listing 4.15 Testing the KeywordFilterFunction with the LocalRunner

```
public class KeywordFilterFunctionLocalRunnerTest {
    final static String BROKER_URL = "pulsar://localhost:6650";
    final static String IN = "persistent://public/default/raw-feed";
    final static String OUT = "persistent://public/default/filtered-feed";

    private static ExecutorService executor;
    private static LocalRunner localRunner;
    private static PulsarClient client;
    private static Producer<String> producer;
    private static Consumer<String> consumer;
    private static String keyword = "";

    public static void main(String[] args) throws Exception {
        if (args.length > 0) {
            keyword = args[0];      #A
        }
        startLocalRunner();
        init();
        startConsumer();
        sendData();
        shutdown();
    }

    private static void startLocalRunner() throws Exception {
        localRunner = LocalRunner.builder()
            .brokerServiceUrl(BROKER_URL)      #B
            .functionConfig(getFunctionConfig()) #C
            .build();
        localRunner.start(false);      #D
    }

    private static FunctionConfig getFunctionConfig() {
        FunctionConfig config = new FunctionConfig();
        config.setLambda(new Lambda("KeywordFilterFunction", "com.manning.function.chapter4.KeywordFilterFunction"));
        config.setParallelism(1);
        return config;
    }

    private static void init() {
        client = PulsarClient.builder()
            .serviceUrl(BROKER_URL)
            .build();
        producer = client.newProducer().topic(IN).create();
        consumer = client.newConsumer().topic(OUT).subscribe();
    }

    private static void sendData() {
        String message = "Hello world";
        producer.send(new StringMessage(message));
    }

    private static void shutdown() {
        consumer.close();
        producer.close();
        client.close();
    }
}
```

```

}

private static FunctionConfig getFunctionConfig() {
    Map<String, ConsumerConfig> inputSpecs =
        new HashMap<String, ConsumerConfig> ();

    inputSpecs.put(IN, ConsumerConfig.builder()      #E
        .schemaType(Schema.STRING.getSchemaInfo().getName())
        .build());

    Map<String, Object> userConfig = new HashMap<String, Object>();
    userConfig.put(KeywordFilterFunction.KEYWORD_CONFIG, keyword);
    userConfig.put(KeywordFilterFunction.IGNORE_CONFIG, true);      #F

    return FunctionConfig.builder()
        .className(KeywordFilterFunction.class.getName())      #G
        .inputs(Collections.singleton(IN))      #H
        .inputSpecs(inputSpecs)      #I
        .output(OUT)      #J
        .name("keyword-filter")
        .tenant("public")
        .namespace("default")
        .runtime(FunctionConfig.Runtime.JAVA)      #K
        .subName("keyword-filter-sub")
        .userConfig(userConfig)      #L
        .build();
}

private static void init() throws PulsarClientException {      #M
    executor = Executors.newFixedThreadPool(2);
    client = PulsarClient.builder()
        .serviceUrl(BROKER_URL)
        .build();
    producer = client.newProducer(Schema.STRING).topic(IN).create();
    consumer = client.newConsumer(Schema.STRING).topic(OUT)
        .subscriptionName("validation-sub").subscribe();
}

private static void startConsumer() {      #N
    Runnable runnableTask = () -> {
        while (true) {
            Message<String> msg = null;
            try {
                msg = consumer.receive();
                System.out.printf("Message received: %s \n", msg.getValue());
                consumer.acknowledge(msg);
            } catch (Exception e) {
                consumer.negativeAcknowledge(msg);
            }
        };
    };
    executor.execute(runnableTask);
}

private static void sendData() throws IOException {      #O
    InputStream inputStream = Thread.currentThread().getContextClassLoader()
        .getResourceAsStream("test-data.txt");
}

InputStreamReader streamReader = new InputStreamReader(inputStream,
    StandardCharsets.UTF_8);

```

```

        BufferedReader reader = new BufferedReader(streamReader);
        for (String line; (line = reader.readLine()) != null;) {
            producer.send(line);
        }
    }

    private static void shutdown() throws Exception {      #P
        executor.shutdown();
        localRunner.stop();
    . .
}

```

#A Get the user provided keyword
#B The service URL for the Pulsar cluster the function will run on.
#C Pass in the function configuration to the LocalRunner
#D Start the LocalRunner and Function
#E Specifies that the data inside the input topic will be Strings
#F Initialize the user configuration properties with the user provided keyword
#G Specifies the class name of the function want to run locally
#H Specifies the input topic
#I Pass in the input topic configuration properties
#J Specifies the output topic
#K Specifies that we want to use a Java Runtime environment for the Function
#L Pass in the user configuration properties
#M initialize a producer and consumer to use for testing
#N Launch the consumer in a background thread to read messages from the Function output topic
#O Send data to the Function input topic
#P Stop the LocalRunner and Consumer, etc.

The easiest way to gain access to a Pulsar cluster is to launch the Pulsar Docker container like we did in Chapter 3 by running the following command in a bash window, which will start a Pulsar cluster in standalone mode inside the container. Note that we are also mounting the directory where you cloned the GitHub project associated with this book onto local machine.

```

$ export GIT_PROJECT=<CLONE_DIR>/pulsar-in-action/chapter4
$ docker run --name pulsar -id \
-p 6650:6650 -p 8080:8080 \
-v $GIT_PROJECT:/pulsar/dropbox
apachepulsar/pulsar:latest bin/pulsar standalone

```

Typically, you would run the LocalRunner test from inside your IDE to attach a debugger and step through the function code to identify and resolve any errors you have encountered. However, in this scenario I want run the LocalRunner test using the command line. Therefore, I must first bundle the test class that is located under the chapter4/src/main/test folder of the GitHub repo associated with this book into a jar file along with all the necessary dependencies, including the LocalRunner class by running the maven assemble command. Once that is complete, we can start the LocalRunner commands as shown in Listing 4.16.

Listing 4.16 Starting the LocalRunner and entering some data.

```
mvn clean compile test-compile assembly:single    #A
...
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time:  29.279 s
[INFO] Finished at: 2020-08-15T15:43:58-07:00

java -cp ./target/chapter4-0.0.1-fat-tests.jar
      com.manning.pulsar.chapter4.functions.sdk.KeywordFilterFunctionLocalRunnerTest
      Director    #B

org.apache.pulsar.functions.runtime.thread.ThreadRuntime - ThreadContainer starting
  function with instance config InstanceConfig(instanceId=0, functionId=0bc39b7d-fb08-
  4549-a6cf-ab641d583edd, functionVersion=7786da28-0bb6-4c11-97d9-3d6140cc4261,
  functionDetails=tenant: "public"
namespace: "default"
name: "keyword-filter"
className: "com.manning.pulsar.chapter4.functions.sdk.KeywordFilterFunction"    #C
userConfig: "{\"keyword\":\"Director\",\"ignore-case\":true}"
autoAck: true
parallelism: 1
source {    #D
  typeClassName: "java.lang.String"
  subscriptionName: "keyword-filter-sub"
  inputSpecs {
    key: "persistent://public/default/raw-feed"
    value {
      schemaType: "String"
    }
  }
  cleanupSubscription: true
}
sink {    #E
  topic: "persistent://public/default/filtered-feed"
  typeClassName: "java.lang.String"
  forwardSourceMessageProperty: true
}
. . .

Message received: At the end of the room a loud speaker projected from the wall. The
  Director walked up to it and pressed a switch.    #F
Message received: The Director pushed back the switch. The voice was silent. Only its thin
  ghost continued to mutter from beneath the eighty pillows.
Message received: Once more the Director touched the switch.
```

#A Builds the fat jar containing the LocalRunner test and all its dependencies

#B Run the LocalRunner test and specify a keyword of "Director" to filter on

#C The output should indicate that the Function was deployed as expected.

#D The output should display the Function's input topic

#E The output should display the Function's output topic

#F The output should contain only sentences with the keyword "Director" in them.

Let's review the steps that just occurred; The KeywordFilterFunction, which was running locally inside the JVM launched by the java command connected to the Pulsar instance that

was running inside the docker container we launched earlier. The Function's input and output topics were created inside that Pulsar instance, and all of the data published by the producer running inside the `KeywordFilterFunctionLocalRunnerTest`'s `sendData` method was written to the Pulsar topic inside the Docker container.

The `KeywordFilterFunction` was listening on this same topic and the processing logic was applied to each line published to that topic. Only those messages that contained the keyword, which in this case was "Director", were written to the Function's configured output topic. The consumer running inside the `KeywordFilterFunctionLocalRunnerTest`'s `startConsumer` method was also reading messages from this output topic and writing them to standard out so that we could verify the results.

4.5 Deploying Pulsar Functions

After you have compiled and tested your Pulsar Functions, you will eventually want to deploy them to a production cluster. The `pulsar-admin functions` command-line tool was designed specifically for this purpose, and allows you to provide several configuration properties for the functions, including tenant, namespace, input, and output topics, etc. In this section I will walk through the process of configuring and deploying a Pulsar Function using this tool.

4.5.1 Generating a Deployment Artifact

When you are deploying a Pulsar Function, the first step is to generate a deployment artifact that contains the Function code along with all its dependencies. The type of artifact varies depending on the programming language used to develop the Function.

JAVA SDK FUNCTIONS

For Java-based Functions, the preferred artifact type is a NAR (NiFi archive) file, although jar files are also acceptable if they include all the necessary dependencies. This holds true whether you want to deploy a simple Java native language function, or one that has been developed using the Pulsar SDK. In either case, an archive file is the deployment artifact.

To have your Pulsar Function packaged as a NAR file, you need to include a special plugin inside your `pom.xml` file, as shown in Listing 4.17, that will bundle the Function class along with all its dependencies for you.

Listing 4.17 Add the NAR Maven Plugin to your pom.xml file

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.nifi</groupId>
      <artifactId>nifi-nar-maven-plugin</artifactId>
      <version>1.2.0</version>
      <extensions>true</extensions>
      <executions>
        <execution>
          <phase>package</phase>
          <goals>
            <goal>nar</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

```
        </execution>
    </executions>
</plugin>
...
</plugins>
</build>
```

Once this plugin has been added to your project's pom.xml file, all you need to do is run the `mvn clean install` command to generate the NAR file in your project's `target` folder. This NAR file is the deployment artifact for Java-based Pulsar Functions.

PYTHON SDK FUNCTIONS

For Python-based Pulsar Functions, there are two deployment options depending upon whether you used the Pulsar SDK or not. If you didn't use the SDK, and only want to deploy a python native language function, then the Python source file, e.g., `my-function.py` is the deployment artifact and no further packaging is required.

However, if you wish to deploy a Python-based Pulsar Function that depends upon packages outside of the Python standard libraries including the Pulsar SDK, then you must first package all the required dependencies into a single artifact (ZIP file) before you can deploy it. A file named `requirements.txt` is required inside your Python project folder and is used to maintain a list of all the project dependencies. It is up to the developer to keep this file up to date manually. Note that, `pulsar-client` is not needed as a dependency as it is provided by Pulsar.

When you are ready to create your Python deployment artifact, you first need to run the command shown in Listing 4.18 to download all the Python dependencies specified in the `requirements.txt` file into the `deps` folder of your Python project.

Listing 4.18 Downloading the Python Dependencies

```
pip download \      #A
--only-binary :all: \   #B
--platform manylinux1_x86_64 \   #C
--python-version 37 \   #D
-r requirements.txt \   #E
-d deps      #F

#A Use the pip package manager for Python
#B We need the binary version of all dependencies
#C Specify the target execution platform operating system
#D Specify the Python version
#E Relative path to the requirements.txt file
#F Target download directory
```

After the download is complete, you need to create a destination folder with the desired package name, e.g., `echo-function`. Next you must copy over both the `src` and `deps` folders into it and compress the folder into a ZIP archive as shown in Listing 4.19.

Listing 4.19 Packaging a Python-based Pulsar Function for Deployment

```
mkdir -p /tmp/echo-function    #A  
cp -R deps /tmp/echo-function/  
cp -R src /tmp/echo-function/   #B  
zip -r /tmp/echo-function.zip /tmp/echo-function    #C
```

#A Create a new target directory for the function dependencies and source
#B From inside the Python project folder, copy over dependencies and source
#C Use the zip command to create a ZIP file containing the contents of the target directory

The ZIP file generated from the command in Listing 4.19 is the deployment artifact for Python-based Pulsar Functions.

GOLANG SDK FUNCTIONS

For Golang-based Pulsar Functions, the preferred artifact type is a single binary executable file that contains the machine byte code for your function along with all the supporting code needed to execute the code on any computer, regardless of whether that system has the .go source files, or even a Go installation. You can use the Go toolchain as shown in Listing 4.20 to generate this binary executable file.

Listing 4.20 Packaging a Go-based Pulsar Function for Deployment

```
cd chapter4    #A  
go build echoFunction.go    #B  
  
go: downloading github.com/apache/pulsar/pulsar-function-go v0.0.0-20210723210639-  
      251113330b08  
go: downloading github.com/sirupsen/logrus v1.4.2  
go: downloading github.com/golang/protobuf v1.4.3  
go: downloading github.com/apache/pulsar-client-go v0.5.0  
...      #C  
  
ls -l    #D  
  
-rwxr-xr-x  1 david  staff  23344912 Jul 25 15:23 echoFunction    #E  
-rw-r--r--  1 david  staff      538 Jul 25 15:20 echoFunction.go    #F
```

#A Make sure you are in the project directory
#B Use the Go build command to generate the binary executable
#C Go will automatically download any packages that you need and include them in the binary
#D After the build command finishes, check the directory contents
#E The executable binary that was generated
#F The Go source file containing the Pulsar Function.

If you are running macOS or Linux, the binary executable file will be generated inside the directory where you run the command and is named after the source file. This is the artifact that must be deployed to run the Golang-based Function inside of Pulsar.

4.5.2 Configuration and Deployment

Now that we know how to generate a deployment artifact, the next step is to configure and deploy our Pulsar Functions. All Functions are required to provide some basic configuration details when they are deployed to Pulsar such as the input and output topics, etc. There are two ways that this configuration information can be specified, the first is as command-line arguments passed to the pulsar-admin functions CLI tool using the [create](#) and [update](#) function commands, e.g., bin/pulsar-admin functions create , and the second one is file a single configuration file.

Which method you choose is a matter of preference, but I would strongly encourage the use of the later, as it not only simplifies the deployment process, but it also allows you to store the configuration details in source-control along with the source code. This allows you to refer to the properties at any time and ensures that you are always running your Functions with the correct configuration. If you choose this approach then you will only need to provide two configuration values when you create deploy your functions: the name of the function artifact file containing the executable Function code, and the name of the file containing all the configuration settings.

The contents of a configuration file named `function-config.yaml` are shown in Listing 4.21. We are going to use this file to deploy the `KeywordFilterFunction` to the Pulsar cluster running inside the Docker container we started earlier in the chapter.

Listing 4.21 The Function Configuration File for the KeywordFilterFunction

```
className: com.manning.pulsar.chapter4.functions.sdk.KeywordFilterFunction
tenant: public      #A
namespace: default   #B
name: keyword-filter
inputs:      #C
- persistent://public/default/raw-feed
output: persistent://public/default/filtered-feed
userConfig:    #D
  keyword : Director
  ignore-case: false

#####
# Processing
#####
autoAck: true      #E
logTopic: persistent://public/default/keyword-filter-log
processingGuarantees: ATLEAST_ONCE    #F
retainOrdering: false     #G
timeoutMs: 30000
subName: keyword-filter-sub    #H
cleanupSubscription: true
```

#A Every function must have an associated tenant

#B Every function must have an associated namespace

#C There can be more than one input topic per function

#D The user configuration key, value map

#E Whether the function should acknowledge the message after it has processed the message

#F The delivery semantics applied to the function

#G Whether the function should process the input messages in the order they were published or not.

#H The name of the subscription on the input topic

As you see, this file provides all the properties needed to run the Pulsar Function including the class name, input/output directories and much more. It also contains a few other configuration settings that are available to all Functions that control how the messages are consumed by the Function that I wanted to highlight:

- `auto_ack`: Whether messages consumed from the input topic(s) are automatically acknowledged by the Function framework or not. When set to the default value of true each message that doesn't result in an exception is acked automatically when the Function is done executing. If set to false, the Function code is responsible for message acknowledgement.
- `retain-ordering`: Whether messages are consumed in exactly the order they appear on the input topics or not. When set to true, then negatively acknowledged messages will be re-delivered before any unprocessed messages.
- `processing-guarantees`: The processing guarantees (delivery semantics) applied to messages consumed by the function. Possible values are: [ATLEAST_ONCE, ATMOST_ONCE, EFFECTIVELY_ONCE]

When executing a stream processing application, you may want to specify the delivery semantics for the data processing within your Pulsar Functions. These guarantees are meaningful since you must always assume that there is the possibility of failures in the network, machines, etc. that can result in data loss.

In the context of Pulsar Functions, these processing guarantees determine how often a message is processed and how it is handled in the event of failure. Pulsar Functions support three distinct messaging semantics that you can apply to any function. By default, Pulsar Functions provide at-most-once delivery guarantees, but you can configure your Pulsar Function to provide any of the following message processing semantics instead.

AT-MOST-ONCE

At-most-once processing does not provide any guarantee that data is processed, and no additional attempts will be made to reprocess the data if it was lost before the Pulsar Function could process it. Each message that is sent to the function will either be processed one time at the most or not at all.

As you can see in Figure 4.6, when a Pulsar Function is configured to use at-most-once processing, the message is immediately acknowledged after it is consumed, regardless of whether the message is successfully processed or not. In this scenario, message M2 will be processed next by the Function even if message M1 caused a processing exception.

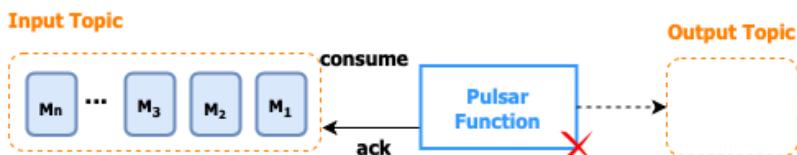


Figure 4.6: With at-most-once processing, the incoming messages are acknowledged regardless of processing success or failure. This gives each message just one chance at being processed.

You only want to use this processing guarantee if your application can handle periodic data loss without impacting the correctness of the data. One such example would be a Function that calculates the average value of sensor reading, such as a temperature. Over the lifetime of the Function, it will process 10s of millions of individual temperature readings, and the loss of handful of these readings will be inconsequential to the accuracy of the computed average.

AT-LEAST-ONCE

At-least-once processing guarantees that any data published to one of the Functions input topics will be successfully processed by the Function at least one time. In the event of a processing failure the message will automatically get redelivered. Therefore, there is the possibility that any given message could be processed more than once.

With this processing semantic, the Pulsar Function reads the message from the input topic, executes its logic, and acknowledges the message. If the Function fails to acknowledge the message it is re-processed. This process is repeated until the Function acknowledges the message. Figure 4.7 depicts the scenario in which the Function consumes the message but encounters a processing error that causes it to fail to send and acknowledgement. In this scenario, the next message processed by the Function would be M1 and would continue to be M1 until the Function succeeds.

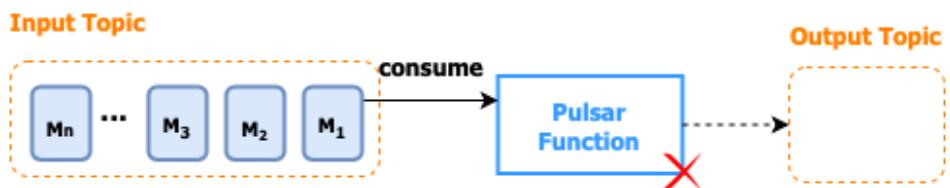


Figure 4.7: With at-least-once processing, if the Function encounters an error and fails to acknowledge the message, then the same message will be processed again.

You will only want to use this processing guarantee if your application can handle processing the same data multiple times without impacting the correctness of the data. One such scenario would be if the incoming messages represented records that would be updated in a database. In such a scenario, multiple updates with the same values would have no impact to the underlying database.

EFFECTIVELY-ONCE

With effectively-once guarantees, a message can be received and processed more than once, which is common in the presence of failures. The crucial thing is that the actual outcome of the Function processing on the resulting state will be as if the re-processed events were observed only once. This is most often what you will want to achieve, the system will behave as if each message is processed once and only once.

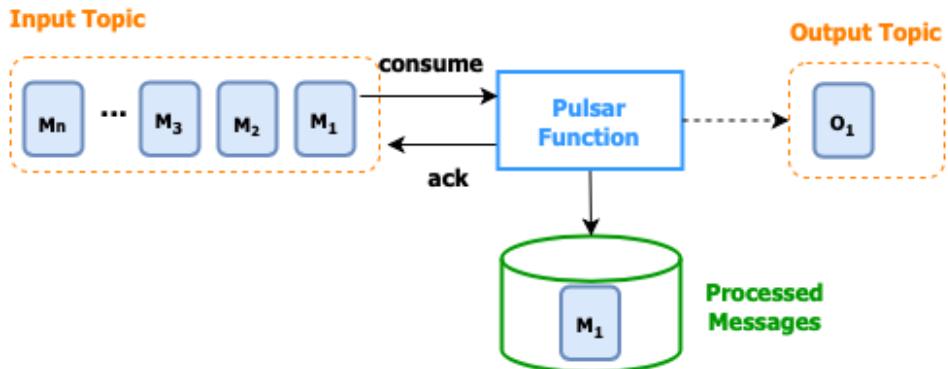


Figure 4.8: With effectively once processing, duplicate messages will be ignored.

Figure 4.8 depicts the scenario in which an upstream producer to the Function’s input topic has re-sent the same message, M₁. When configured to provide effectively once processing, the Function will check to see if has previously processed the message (based on user-defined properties) and if so, ignore the message and send an acknowledgment so that it will not be re-processed.

Now that we have defined the configuration we want to use for the Function, the next step is to deploy the Function onto a running Pulsar cluster. As we mentioned earlier, the Pulsar cluster that we started earlier in the chapter that is running inside a Docker container will do just fine. From inside a bash shell, you can execute the command shown in Listing 4.22 to deploy the `KeywordFilterFunction` on that Pulsar cluster.

Listing 4.22 Deploy the KeywordFilterFunction

```
$ docker exec -it pulsar bin/pulsar-admin functions create \#A
  --jar /pulsar/dropbox/target/chapter4-0.0.1.nar \#B
  --function-config-file /pulsar/dropbox/src/main/resources/function-config.yaml    \#C
```

#A Using the pulsar-admin tool inside the Docker container

#B Specify the deployment artifact, which must either a locally accessible file or a URL address

#C Specify the function configuration file which must either a locally accessible file or a URL address

If everything went as expected, you should see “Created successfully” in the output of the command which indicates that the Function was created and deployed on the Pulsar cluster. Let’s take a moment to review what happened and why the command was able to work. First off, we used the `pulsar-admin` CLI tool which only exists inside the Docker container. Therefore, we needed to use the `Docker exec` command to run the `pulsar-admin` CLI tool inside the Docker container. In the command we provided two switches, one to specify the location of the deployment artifact, i.e., the NAR file and the other to specify the location of the function configuration file. Since we were deploying a Java-based Pulsar Function, we used the `--jar` switch to specify the location of the artifact. Had this been a Python-based Function, we would have had to use the `--py` switch instead, and similarly the `--go` switch

for a Golang-based Function. Using the correct switch is critical because Pulsar uses those switch values to determine which runtime execution environment that is needed to run the Function, i.e., whether to spin up a JVM, a Python interpreter, or Go runtime for the Function to run in.

Both the configuration file and the function artifact files must either be on the same machine as the pulsar-admin CLI tool, or downloadable via a URL. Therefore, we mounted the \$GIT_PROJECT directory to the /pulsar/dropbox folder inside the Docker container. That made both files locally accessible to the pulsar-admin CLI tool. While this is a great trick for local development, bear in mind that in a real production scenario, these files should be physically moved to the Pulsar cluster, preferably as part of the CI/CD release process.

We can also check the status of our deployed function using the `function getstatus` command as shown in Listing 4.23, which will give us useful information about the function including the status of the function, how many messages it has processed, the average processing latency of the function, as well as any exceptions that the Function may have thrown.

Listing 4.23 Checking the status of the KeywordFilterFunction

```
# docker exec -it pulsar /pulsar/bin/pulsar-admin functions getstatus --name keyword-filter
{
    "numInstances" : 1,      #A
    "numRunning" : 1,       #B
    "instances" : [ {
        "instanceId" : 0,
        "status" : {
            "running" : true,   #C
            "error" : "",       #D
            "numRestarts" : 0,
            "numReceived" : 0,   #E
            "numSuccessfullyProcessed" : 0,   #F
            "numUserExceptions" : 0,
            "latestUserExceptions" : [ ],
            "numSystemExceptions" : 0,
            "latestSystemExceptions" : [ ],
            "averageLatency" : 0.0,   #G
            "lastInvocationTime" : 0,   #H
            "workerId" : "c-standalone-fw-localhost-8080"
        }
    } ]
}
```

#A The number of requested instances of the function.

#B The actual number of running instances of the function.

#C Status indicator.

#D If any errors were thrown, they would be displayed here.

#E The number of messages received by the function

#F The number of messages that have been successfully processed by the function.

#G The average processing latency per message.

#H The last time a message was processed by the function.

If you ever need to debug a running Pulsar Function, then the `function getstatus` command is an excellent place to start. The pulsar-admin API provides another command that is will show you the configuration settings of a Pulsar Function as well. You can check

the configuration of a Pulsar Function by using the `function get` command as shown in Listing 4.24.

Listing 4.24 Checking the configuration of the KeywordFilter Function

```
# docker exec -it pulsar /pulsar/bin/pulsar-admin functions get --name keyword-filter
{
  "tenant": "public",
  "namespace": "default",
  "name": "keyword-filter",
  "className": "com.manning.pulsar.chapter4.functions.sdk.KeywordFilterFunction",
  "inputSpecs": {
    "persistent://public/default/raw-feed": {      #A
      "isRegexPattern": false,
      "schemaProperties": {}
    }
  },
  "output": "persistent://public/default/filtered-feed",      #B
  "logTopic": "persistent://public/default/keyword-filter-log",
  "processingGuarantees": "ATLEAST_ONCE",      #C
  "retainOrdering": false,      #D
  "forwardSourceMessageProperty": true,
  "userConfig": {      #E
    "keyword": "Director",
    "ignore-case": false
  },
  "runtime": "JAVA",      #F
  "autoAck": true,      #G
  "subName": "keyword-filter-sub",
  "parallelism": 1,      #H
  "resources": {      #I
    "cpu": 1.0,
    "ram": 1073741824,
    "disk": 10737418240
  },
  "timeoutMs": 30000,
  "cleanupSubscription": true
}
```

#A The input topic(s).

#B The output topic.

#C The processing guarantee for the function.

#D Whether the messages are to be processed in the order they were published to the topic or not.

#E Any user configuration values.

#F The runtime environment of functions, e.g., JAVA, Python, etc.

#G Whether the function will automatically acknowledge messages

#H The number of parallel instances of the function that were requested

#I The computing resources allocated to the function.

4.5.3 The Function Deployment Lifecycle

As we saw in the previous section there are quite a few configuration parameters available when creating and updating a function, in this section will we cover how some of those parameters are used within the Pulsar Function deployment lifecycle. When a Function is first created, the associated library bundle is stored in Apache BookKeeper, so that it can be accessed by any Pulsar broker node in the cluster. The bundle is associated with the fully

qualified function name, which is a combination of the tenant, namespace, and function name to ensure that it is globally unique across the Pulsar cluster.

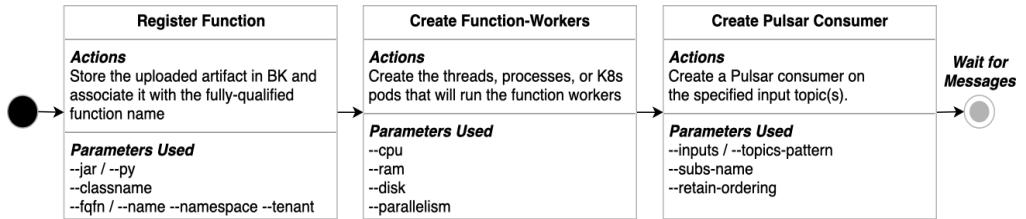


Figure 4.9: The Pulsar Function Deployment Lifecycle.

When a Pulsar Function is first created, the steps shown in Figure 4.9 are performed in sequence. After the Function is registered and all its details are persisted to BookKeeper, function workers are created based on the provided configuration parameters and the Pulsar cluster's deployment mode (which we will discuss in the next section). The function workers are the runtime instantiations of the Pulsar Function code and can be either threads, processes, or Kubernetes pods. Lastly, within each function worker, a Pulsar consumer and subscription is created on the configured input topics. The function workers then await the arrival of incoming messages and perform their processing logic on the incoming messages.

4.5.4 Deployment Modes

As we have seen, to deploy and manage Pulsar Functions you need to have a Pulsar cluster running, however there are a couple of options for where your Pulsar Function instances will run. You can have the Pulsar Function run on your local development machine, aka **localrun mode**, in which case it interacts with the Broker over the network. This was what we did with the LocalRunner test case. For production, you will want to deploy your Functions in **cluster mode**.

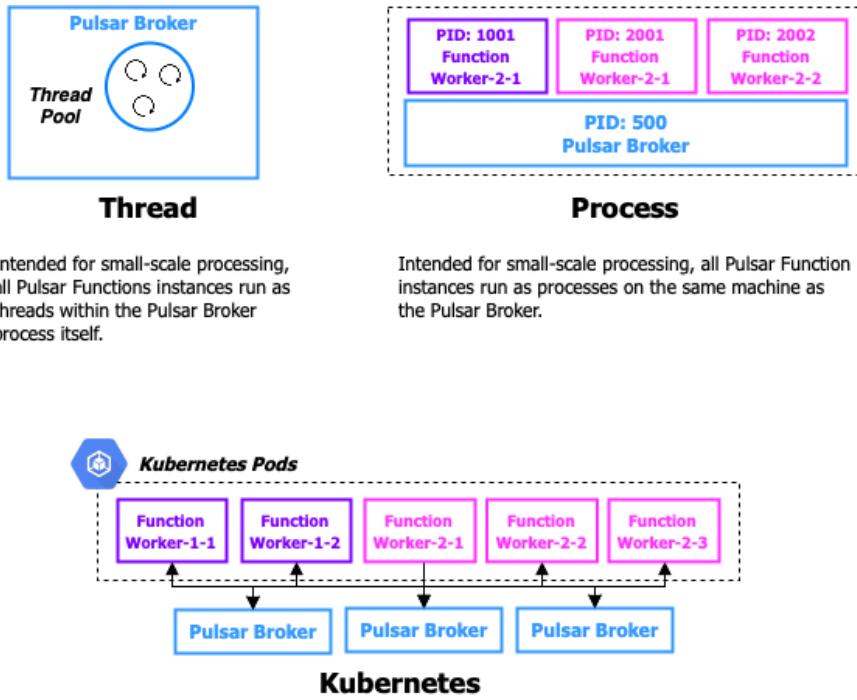


Figure 4.10: A Pulsar Function can run as either a thread, process, or K8s StatefulSet inside the Pulsar Function Workers.

CLUSTER MODE

In this mode, users ‘submit’ their functions to a running pulsar cluster and Pulsar will take care of distributing them across the Function Workers for execution. The Pulsar Functions framework supports the following runtime options when running in cluster mode; thread, process, and Kubernetes as shown in Figure 4.10 and these refers to how the Function code itself is executed inside a **Function Worker**, which is the runtime environment used to host Pulsar Functions.

In Pulsar, there are two options for where the Function Workers can run. They can either run inside the Pulsar Brokers themselves which simplifies the deployment. The other option is to start the Function Worker processes on their own separate nodes to provide better resource isolation between them and your Brokers.

The benefit of running the Function Workers on the broker node itself as either a thread or a separate process includes a smaller hardware footprint as you won’t need as many nodes in your environment, and a reduction in network latency between the Function processes and the Broker that is serving the messages. However, running the Function

Workers on separate nodes provides better resource isolation and insulates the Pulsar Broker process from being inadvertently killed by a Function Worker crashing and bringing the Broker down with it.

4.5.5 Pulsar Function Data Flow

Before I conclude this chapter, I wanted to take a moment to document the flow of an individual message through a Pulsar Function and tie the various stages back to the configuration parameters supplied when you first create or update a Function, so that you have a better understanding of how to configure your Functions. The data flow inside a Pulsar function is depicted in the state machine shown Figure 4.11 along with the configuration properties that control the Function behavior.

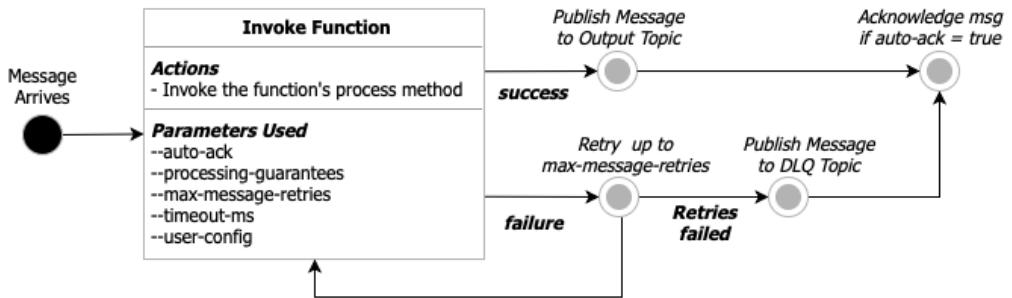


Figure 4.11: The basic message flow for a Pulsar Function running inside of a Pulsar Worker

When a message arrives on any of the Functions configured input topics, the Function's process method is called with the message contents as an input parameter. If the Function can successfully process the message without encountering any runtime exceptions, then the value returned by the method call is published to the configured output topic, unless it is a Void function, in which case nothing is published.

However, if the Function encounters a runtime exception, then the message is retried up to the value configured in the max-message-retries parameter. If all these attempts fail, then the message is routed to the configured dead-letter-queue topic (if any) so that it can be retained for future examination. In either case, the message is acknowledged as consumed by the Pulsar Function if the auto-ack flag was configured to true, allowing the next message to be processed.

4.6 Summary

- Pulsar Functions are a serverless computing framework that runs on top of Apache Pulsar that allows you to define functions that get executed when a new message arrives in Topic
- Pulsar Functions can be written in several popular languages including Python, Go, and Java, but throughout this book we will focus on Java

- Pulsar Function can be configured, submitted, and monitored from the Pulsar command line interface
- If you have experience with Docker, you can quickly create a local test environment for deploying and testing Pulsar Functions.

5

Pulsar IO Connectors

This chapter covers

- Introduction to the Pulsar IO framework
- Configuring, deploying, and monitoring Pulsar IO connectors
- Writing your own Pulsar IO connector in Java

Messaging systems are much more useful when you can easily use them to move data into and out of other external systems such as databases, local and distributed filesystems, or other messaging systems. Consider the scenario where you want to ingest log data from external sources such as applications, platforms, and cloud-based services and publish it to a search engine for analysis. This could easily be accomplished with a pair of Pulsar IO connectors; the first would be a Pulsar source that collects the application logs, and the second would be a Pulsar Sink that writes the formatted records to Elastic Search.

Pulsar provides a collection of pre-built connectors that can be used to interact with external systems, such as Apache Cassandra, ElasticSearch, and HDFS just to name a few. The Pulsar IO framework is also extensible which allows you to develop your own connectors to support new or legacy systems as needed.

5.1 What are Pulsar IO Connectors?

The Pulsar IO connector framework provides developers, data engineers, and operators an easy way to move data into and out of the Pulsar messaging platform without having to write any code or become experts in both Pulsar and the external system. From an implementation perspective, Pulsar IO connectors are just specialized Pulsar Functions purpose-built to interface with external systems through an extensible API interface.

Compare this to a scenario in which you had to implement the logic for interacting with an external system, such as MongoDB inside a Java class that uses the Pulsar Java client. Not only would you have to become familiar with the client interface for MongoDB and

Pulsar, but you would also have the operational burden of deploying and monitoring a separate process that is now a critical part of your application stack. Pulsar IO seeks to make the movement of data into and out of Pulsar less cumbersome.

Pulsar IO connectors come in two types: **Sources** that ingest data from an external system into Pulsar; and **Sinks** that feed data from Pulsar into an external system. Figure 5.1 illustrates the relationship between sources, sinks, and Pulsar.

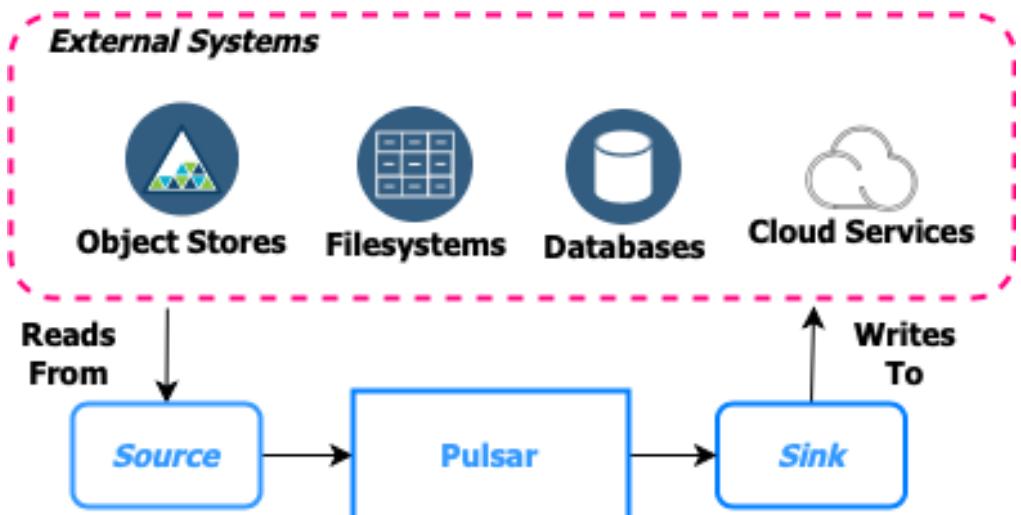


Figure 5.1 Sources consume data from external systems, while sinks write data to external systems

5.1.1 Sink Connectors

While the Pulsar IO framework already provides a collection of built-in connectors for some of the most popular data systems, it was designed with extensibility in mind. Allowing users to add new connectors as new systems and APIs are developed. The programming model behind Pulsar IO Connectors is very straightforward, which greatly simplifies the development process. Pulsar IO Sink Connectors can receive messages from one or more **input** topics. Every time a message is published to any of the input topics, the Pulsar Sink's write method is called.

The implementation of the write method is responsible for determining how to process the incoming message contents and properties before in order to write data to the source system. The Pulsar Sink shown in Figure 5.2 can use the message contents to determine which database table to insert the record into and then construct and execute the appropriate SQL command to do so.

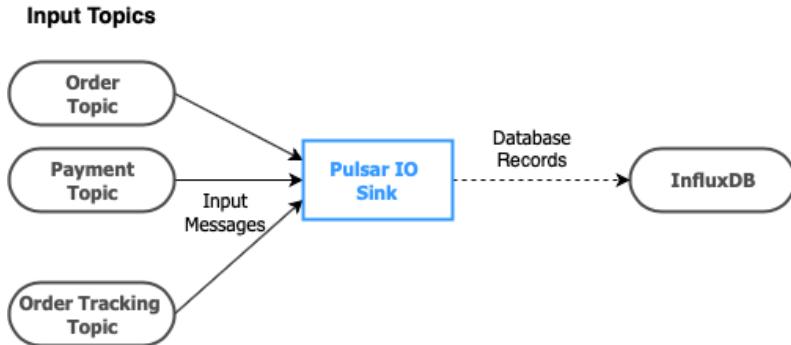


Figure 5.2 Overview of the Pulsar IO Sink Connector programming model.

The easiest way to create a custom sink connector is to write a Java class that implements the `org.apache.pulsar.io.core.Sink` interface shown in Listing 5.1. The first method defined in the interface is the `open` method, which is called just once when the sink connector is created and can be used to initialize all the necessary resources, e.g., for a database connector you can create the JDBC client. The `open` method also provides a single input parameter named "config" from which you can retrieve all the connector specific settings, e.g., the database connection URL, username, and password.

In addition to the passed-in `config` object, the Pulsar runtime also provides a `SinkContext` for the connector that provides access to runtime resources much like the `Context` object does in the Pulsar Function's API.

Listing 5.1 The Pulsar Sink Interface

```

package org.apache.pulsar.io.core;

public interface Sink<T> extends AutoCloseable {
    /**
     * Open connector with configuration
     *
     * @param config initialization config
     * @param sinkContext
     * @throws Exception IO type exceptions when opening a connector
     */
    void open(final Map<String, Object> config,
             SinkContext sinkContext) throws Exception;

    /**
     * Write a message to Sink
     * @param record record to write to sink
     * @throws Exception
     */
    void write(Record<T> record) throws Exception;
}

```

The other method defined in the interface is the `write` method, which is responsible for consuming messages from the Sink's configured source Pulsar topic and writing the data to the external source system. The "write" method receives an object that implements the `org.apache.pulsar.functions.api.Record` interface that provides information that can be used when processing the incoming message.

It is worth pointing out that the Sink interface extends the `AutoCloseable` interface, which includes a `close` method definition that can be used to release any resources, such as database connections or open file writers before the connector is stopped.

Listing 5.2 The Record Interface

```
package org.apache.pulsar.functions.api

public interface Record<T> {

    default Optional<String> getTopicName() {      #A
        return Optional.empty();
    }

    default Optional<String> getKey() {      #B
        return Optional.empty();
    }

    T getValue();      #C

    default Optional<Long> getEventTime() {      #D
        return Optional.empty();
    }

    default Optional<String> getPartitionId() {      #E
        return Optional.empty();
    }

    default Optional<Long> getRecordSequence() {      #F
        return Optional.empty();
    }

    default Map<String, String> getProperties() {      #G
        return Collections.emptyMap();
    }

    default void ack() {      #H
    }

    default void fail() {      #I
    }

    default Optional<String> getDestinationTopic() {      #J
        return Optional.empty();
    }
}
```

#A If the record originated from a topic, report the topic name.

#B Return a key if the key has one associated.

#C Retrieves the actual data of the record.

#D Retrieves the event time of the record from the source.

- #E If the record is originated from a partitioned source, return its partition id. The partition id will be used as part of the unique identifier by Pulsar IO runtime to do message deduplication and achieve exactly-once processing guarantee.
- #F If the record is originated from a sequential source, return its record sequence. The record sequence will be used as part of the unique identifier by Pulsar IO runtime to do message deduplication and achieve exactly-once processing guarantee.
- #G Retrieves user-defined properties attached to record.
- #H Acknowledge the source system that this record is fully processed.
- #I Indicate to the source system that this record has failed to be processed.
- #J To support message routing on a per message basis.

The implementation of the record should also provide two methods: ack and fail. These two methods will be used by Pulsar IO connector to acknowledge the records that have been processed and fail the records that have failed. Failure to acknowledge or fail messages within the source connector will result in the messages being retained, which will ultimately cause the connector to stop processing due to back-pressure.

5.1.2 Source Connectors

Pulsar IO Source Connectors are responsible for consuming data from external systems and publishing the data to the configured output topic. There are two distinct types of sources supported by the Pulsar IO framework, the first are those that operate on a pull-based model. As you can see from Figure 5.3, the Pulsar IO framework repeatedly calls the `Source connector's read()` method to pull data from the external source into Pulsar.

In this particular case, the logic inside the connector's `read` method would be responsible for querying the database, converting the result set into Pulsar messages, and publishing them to the output topic. This type of connector would be particularly useful when you have a legacy order entry application that only writes incoming orders into a MySQL database, and you want to expose these new orders to other systems for real-time processing and analysis.

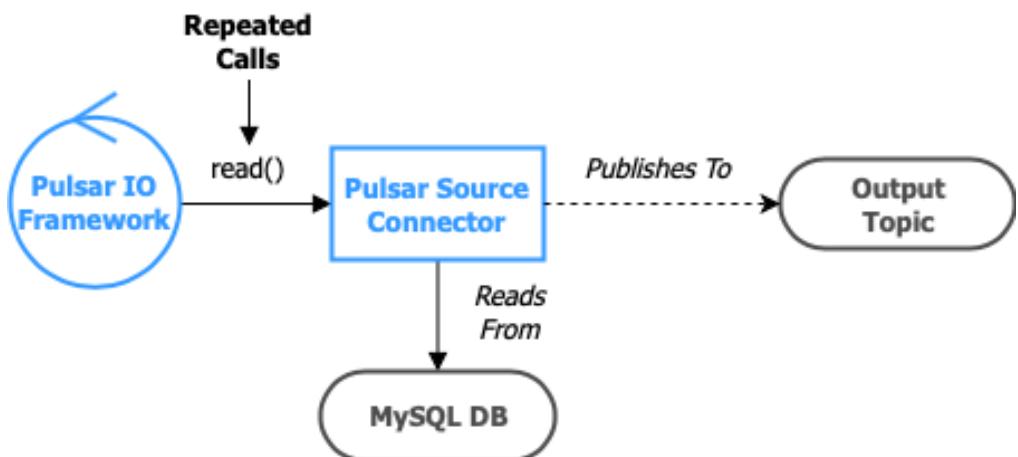


Figure 5.3 The Source Connector's `read` method is repeatedly called, which then pulls the information from the database and publishes it to Pulsar.

The easiest way to create a pull-based source connector is to write a Java class that implements the `org.apache.pulsar.io.core.Source` interface which is shown in Listing 5.3. The first method defined in this interface is the `open` method, which is called just once when the source connector is created and should be used to initialize all the necessary resources, such as a database client.

The `open` method specifies an input parameter named `config` of type `Map`, from which you can retrieve all the connector specific settings, such as the database connection URL, username, and password. This input parameter contains all of the values specified in the file specified by the `--source-config-file` parameter along with all the default configuration settings and values provided by the various switches used to create or update a function.

Listing 5.3 The Pulsar Source Interface

```
package org.apache.pulsar.io.core;

public interface Source<T> extends AutoCloseable {
    /**
     * Open source with configuration
     *
     * @param config initialization config
     * @param sourceContext
     * @throws Exception IO type exceptions when opening a connector
     */
    void open(final Map<String, Object> config,
              SourceContext sourceContext) throws Exception;

    /**
     * Reads the next message from source.
     * If source does not have any new messages, this call should block.
     * @return next message from source. The result should never be null
     * @throws Exception
     */
    Record<T> read() throws Exception;
}
```

In addition to the `config` parameter, the Pulsar runtime also provides a `SourceContext` for the connector. Much like the `Context` object defined in the Pulsar Function API, the `SourceContext` object provides access to runtime resources for tasks like collecting metrics, retrieving stateful property values, etc.

The other method defined in the interface is the `read` method, which is responsible for retrieving data from the external source system and publishing the data to the target Pulsar topic. The implementation of this method should be blocking on this method if there is no data to return and should never return null. The “`read`” method returns an object that implements the `org.apache.pulsar.functions.api.Record` interface that we saw earlier in Listing 5.2.

It is worth pointing out that the `Source` interface also extends the `AutoCloseable` interface that includes a `close` method definition, which should be used to release any resources such as database connections before the connector is stopped

5.1.3 PushSource Connectors

The second type of Source connectors are those that operate on a push-based model. These connectors continuously gather data and buffers it inside an internal blocking queue for eventual delivery to Pulsar. As you can see from Figure 5.4, PushSource connectors typically have a background thread running continuously that gathers information from the source system and buffers it inside an internal queue. When the Pulsar IO framework repeatedly calls the Source connector's `read()` method the data inside that internal queue is then published into Pulsar.

In this particular case, the connector has a background thread that is listening to all incoming traffic on a network socket and publishing it to the internal queue. This type of connector is particularly useful when you are consuming data from an external source that does not retain any information which can be periodically queried at any future point in time. A network socket is just such an example, in that if the thread wasn't connected and listening at all times, data sent over that network connection would be lost forever. Contrast this with the previous connector which queries a database. In that scenario, the connector can query the database after an order has been entered into the database and still retrieve the data because the database has retained it.

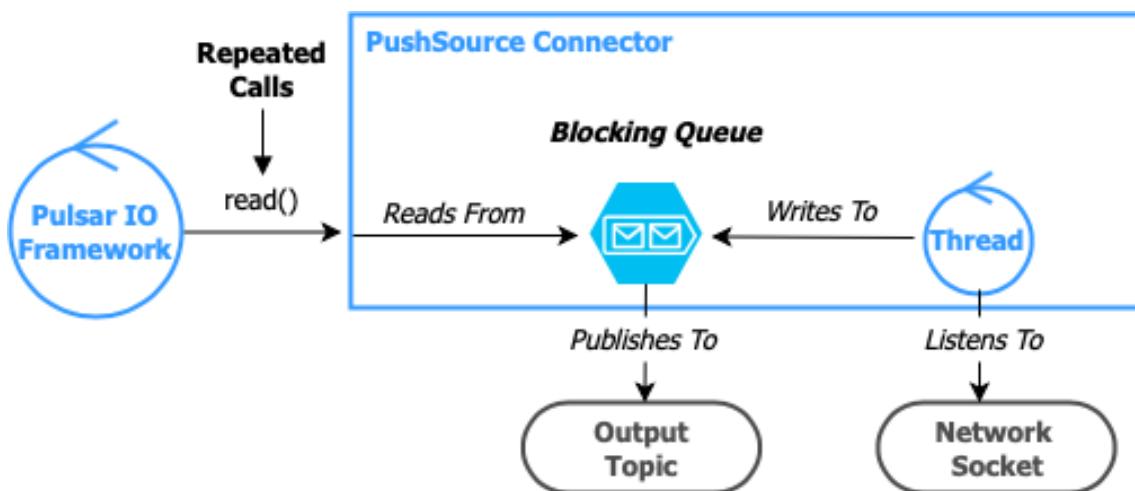


Figure 5.4: The PushSource Connector's has a background thread running that is listening to a network socket and pushing all the traffic it receives to a blocking queue. When the `read` method is called, the data is pulled from the blocking queue.

The easiest way to create a push-based source connector is to write a Java class that extends the abstract `org.apache.pulsar.io.core.PushSource` class shown in Listing 5.4. Since this class implements the Source interface, your class must provide an implementation of all the methods we discussed in the previous section.

Listing 5.4 The PushSource Class

```
package org.apache.pulsar.io.core;
/**
 * Pulsar's Push Source interface. PushSource read data from
 * external sources (database changes, twitter firehose, etc)
 * and publish to a Pulsar topic. The reason its called Push is
 * because PushSources get passed a consumer that they
 * invoke whenever they have data to be published to Pulsar.
 * The lifecycle of a PushSource is to open it passing any config needed
 * by it to initialize(like open network connection, authenticate, etc).
 * A consumer is then to it which is invoked by the source whenever
 * there is data to be published. Once all data has been read, one can use
 * close at the end of the session to do any cleanup
 */
public abstract class PushSource<T> implements Source<T> {

    private LinkedBlockingQueue<Record<T>> queue;
    private static final int DEFAULT_QUEUE_LENGTH = 1000;

    public PushSource() {
        this.queue = new LinkedBlockingQueue<>(this.getQueueLength());      #A
    }

    @Override
    public Record<T> read() throws Exception {
        return queue.take();      #B
    }

    /**
     * Attach a consumer function to this Source. This is invoked by the
     * implementation to pass messages whenever there is data to be
     * pushed to Pulsar.
     *
     * @param record next message from source which should be sent to
     *   a Pulsar topic
     */
    public void consume(Record<T> record) {
        try {
            queue.put(record);      #C
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
    }

    /**
     * Get length of the queue that records are push onto
     * Users can override this method to customize the queue length
     * @return queue length
     */
    public int getQueueLength() {
        return DEFAULT_QUEUE_LENGTH;      #D
    }
}
```

#A The BatchPushSource uses an internal blocking queue to buffer messages

#B Messages are read from the internal queue that blocks if no data is available

#C Incoming messages are stored in the internal queue that blocks if it is full

#D You must override this method if you want to increase the size of the internal queue

The key architectural feature of the PushSource is the `LinkedBlockingQueue` that is used to buffer messages before they are published to Pulsar. This queue allows you to have a continuously running process that listens for incoming data and publishes it to Pulsar. It is also worth noting that the internal queue can be constrained to the desired size, which allows you to limit the memory consumption of the PushSource connector. When the blocking queue reaches the configured size limit, no more records can be published to the queue. This will result in the background thread getting blocked, which could lead to data loss while the queue is full. Therefore, it is important to size the queue properly.

5.2 Developing Pulsar IO Connectors

In the previous section I introduced the Source and Sink interfaces provided by the Pulsar IO connector framework and provided a high-level discussion of what each method does. I will build upon that foundation in this section, as I walk you through the process of developing new connectors.

5.2.1 Developing a Sink Connector

I will start with a very basic Sink connector that receives an endless stream of String values and writes them to local temp file, as shown in Listing 5.5. While this Sink connector does have some limitations, specifically that you cannot write an endless stream of data to a single file, it does serve as a good example that can be used to demonstrate the process of developing a Sink connector.

Listing 5.5 Local File Pulsar IO Sink

```
import org.apache.pulsar.io.core.Sink;      #A

public class LocalFileSink implements Sink<String> {

    private String prefix, suffix;
    private BufferedWriter bw = null;
    private FileWriter fw = null;

    public void open(Map<String, Object> config,
                    SinkContext sinkContext) throws Exception {

        prefix = (String) config.getOrDefault("filenamePrefix", "test-out");
        suffix = (String) config.getOrDefault("filenameSuffix", ".tmp");      #B

        File file = File.createTempFile(prefix, suffix);      #C
        fw = new FileWriter(file.getAbsolutePath(), true);      #D
        bw = new BufferedWriter(fw);
    }

    public void write(Record<String> record) throws Exception {
        try {
            bw.write(record.getValue());      #E
            bw.flush();
            record.ack();      #F
        } catch (IOException e) {
            record.fail();      #G
        }
    }
}
```

```

        throw new RuntimeException(e);
    }

    public void close() throws Exception {      #H
        try {
            if (bw != null)
                bw.close();
            if (fw != null)
                fw.close();
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }
}

```

#A Import the Source interface

#B Retrieve the target filename prefix and suffix from the provided configuration properties.

#C Create the new file in the temporary directory.

#D Initialize the file and buffered writers.

#E Retrieve the value from the incoming record and write it to the open file.

#F Acknowledge that we processed the message successfully, so it can be purged.

#G Indicate that we weren't able to process the message so it can be retained and retried at a later time.

#H Close both of the open file streams to ensure the data is flushed to disk.

The `open` method of the connector retrieves the configuration properties provided by the user and creates the empty target file inside the temp directory of the host. Next the instance-level `FileWriter` and `BufferedWriter` variables are initialized to point to the newly created target file. While the `close` method will attempt to close both of the writers when the connector is stopped.

The Sink's `write` method is invoked whenever a new message arrives in any of the Sink's configured input topics. The method appends the record value to the target file via the `BufferedWriter`'s `write` method before acknowledging that the message has been successfully processed. In the unlikely event that we cannot write the Record's contents to the temp file, an `IOException` will be thrown and the Sink will fail the message before propagating the exception.

5.2.2 Developing a PushSource Connector

Next, let's write a custom push-based Source connector that scans a directory for new files and publishes the contents of these files line-by-line to Pulsar, as shown in Listing 5.6. We want this connector to periodically scan for new files and publish their contents as soon as they the file is written to the directory we are scanning. We can do this by extending the `PushSource` class, which is a specialized implementation of the source interface that is designed to use a background process to continuously produce Records.

Listing 5.6 A PushSource Connector

```
import org.apache.pulsar.io.core.PushSource;
import org.apache.pulsar.io.core.SourceContext;

public class DirectorySource extends PushSource<String> {
    private final ScheduledExecutorService scheduler =
Executors.newScheduledThreadPool(1);      #A

    private DirectoryConsumerThread scanner;

    private Logger log;

@Override
public void open(Map<String, Object> config, SourceContext context)
throws Exception {
    String in = (String) config.getOrDefault("inputDir", ".");
    String out = (String) config.getOrDefault("processedDir", ".");
    String freq = (String) config.getOrDefault("frequency", "10");

    scanner = new DirectoryConsumerThread(this, in, out, log);      #C
    scheduler.scheduleAtFixedRate(scanner, 0, Long.parseLong(freq), TimeUnit.MINUTES);      #D
    log.info(String.format("Scheduled to run every %s minutes", freq));
}

@Override
public void close() throws Exception {
    log.info("Closing connector");
    scheduler.shutdownNow();
}
}
```

#A An internal thread pool for running the background thread

#B Get the runtime settings from the configuration properties that are passed in.

#C Create the background thread, passing in a reference to the source connector and the configs

#D Start the background thread

The `open` method of the Source connector retrieves the configuration properties provided by the user that specifies the local directory to read the files from, and then launches a background thread of type `DirectoryConsumerThread` that is responsible for scanning the directory and reading each of the file's contents line by line. The background thread class shown in Listing 5.7 takes the Source connector instance as a parameter to its constructor method, which it then uses to pass the file contents to internal blocking queue inside the `threads` `process` method.

Listing 5.7 The DirectoryConsumerThread Process

```
import org.apache.pulsar.io.core.PushSource;

public class DirectoryConsumerThread extends Thread {
private final PushSource<String> source;      #A
private final String baseDir;
private final String processedDir;

public DirectoryConsumerThread(PushSource<String> source, String base, String processed,
Logger log) {
```

```

        this.source = source;
        this.baseDir = base;
        this.processedDir = processed;
        this.log = log;
    }

    public void run() {
        log.info("Scanning for files.....");
        File[] files = new File(baseDir).listFiles();
        for (int idx = 0; idx < files.length; idx++) {
            consumeFile(files[idx]);      #B
        }
    }

    private void consumeFile(File file) {
        log.info(String.format("Consuming file %s", file.getName()));
        try (Stream<String> lines = getLines(file)) {      #C
            AtomicInteger counter = new AtomicInteger(0);
            lines.forEach(line ->
                process(line, file.getPath(), counter.incrementAndGet()));      #D

                log.info(String.format("Processed %d lines from %s",
                    counter.get(), file.getName()));
                Files.move(file.toPath(),Paths.get(processedDir)
                    .resolve(file.toPath().getFileName()), REPLACE_EXISTING);      #E
                log.info(String.format("Moved file %s to %s",
                    file.toPath().toString(), processedDir));
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    private Stream<String> getLines(File file) throws IOException {      #F
        if (file == null) {
            return null;
        } else {
            return Files.lines(Paths.get(file.getAbsolutePath()),
                Charset.defaultCharset());
        }
    }

    private void process(String line, String src, int lineNumber) {
        source.consume(new FileRecord(line, src, lineNumber));      #G
    }
}

```

#A A reference to the PushSource connector
#B Process all of the files in the configured base directory
#C Split each file into individual lines.
#D Process each line in the file individually
#E When we are finished with a file move it to the processed directory
#F Splits the given file into a stream of individual lines.
#G Creates a new Record for each line of text in the file.

I also created a new Record type, `FileRecord` shown in Listing 5.8, for this PushSource connector. This allows me to retain some additional meta-data about the file content including the name of the source file, and line number. This type of meta-data can be for

down-stream processing of these records by other Pulsar Functions, as it would allow you to sort the records by file name or type, or ensure that you are processing the lines of a given file in sequence (based on the line number), etc.

Listing 5.8 The FileRecord class

```
import org.apache.pulsar.functions.api.Record;

public class FileRecord implements Record<String> {

    private static final String SOURCE = "Source";
    private static final String LINE = "Line-no";
    private String content;      #A
    private Map<String, String> props;     #B

    public FileRecord(String content, String src, int lineNumber) {
        this.content = content;
        this.props = new HashMap<String, String>();
        this.props.put(SOURCE, srcFileName);
        this.props.put(LINE, lineNumber + "");
    }

    @Override
    public Optional<String> getKey() {
        return Optional.ofNullable(props.get(SOURCE));      #C
    }

    @Override
    public Map<String, String> getProperties() {
        return props;      #D
    }

    @Override
    public String getValue() {
        return content;      #E
    }
}
```

#A The actual contents of the file for this particular line

#B The message properties

#C Use the source file as the key, for key-based subscriptions, etc.

#D The message properties expose the meta-data.

#E The message value is the raw file contents themselves

The thread's `process` method calls the PushSource's `consume` method, passing in the file contents. As we saw previously in Listing 5.4, the `consume()` method inside the PushSource simply writes the incoming data directly to the internal blocking queue. This decouples the reading of the file contents from the pulsar framework's call to the PushSource connector's `read` method.

The use of a background thread is a common design pattern for PushSource connectors that retrieve the data from the external system and then invoke the Source's `consume` method to push the data to the output topic.

5.3 Testing Pulsar IO Connectors

In this section I will walk you through the process of developing and testing a Pulsar Connector. Let's use with the `DirectorySource` connector shown in Listing 5.6 to demonstrate the software development lifecycle for a Pulsar Connector. This connector takes in a user provided directory and publishes the contents of all the files within the given directory line-by-line.

While this code is fairly simplistic, I will walk through the testing process you would typically use when developing a connector for production use. Since this is just plain Java code, we can leverage any of the existing unit testing frameworks such as Junit or TestNG to test the function logic.

5.3.1 Unit Testing

The first step would be to write a suite of unit tests that test some of the more common scenarios in order to validate that the logic is correct and produces accurate results for various sentences that we send it. Since this code uses the Pulsar SDK API, we will need to use a Mocking library, such as Mockito to mock the `SourceContext` object as shown in Listing 5.9.

Listing 5.9 DirectorySource Unit Tests

```
public class DirectorySourceTest {
    final static Path SOURCE_DIR =
        Paths.get(System.getProperty("java.io.tmpdir"), "source");
    final static Path PROCESSED_DIR =
        Paths.get(System.getProperty("java.io.tmpdir"), "processed");      #A

    private Path srcPath, processedPath;
    private DirectorySource spySource;      #B

    @Mock
    private SourceContext mockedContext;

    @Mock
    private Logger mockedLogger;

    @Captor
    private ArgumentCaptor<FileRecord> captor;      #C

    @Before
    public final void init() throws IOException {
        MockitoAnnotations.initMocks(this);
        when(mockedContext.getLogger()).thenReturn(mockedLogger);
        FileUtils.deleteDirectory(SOURCE_DIR.toFile());
        FileUtils.deleteDirectory(PROCESSED_DIR.toFile());      #D
        srcPath = Files.createDirectory(SOURCE_DIR,
            PosixFilePermissions.asFileAttribute(
                PosixFilePermissions.fromString("rwxrwxrwx")));
        processedPath = Files.createDirectory(PROCESSED_DIR,
            PosixFilePermissions.asFileAttribute(
                PosixFilePermissions.fromString("rwxrwxrwx")));      #E
        spySource = spy(new DirectorySource());      #F
    }
}
```

```

@Test
public final void oneLineTest() throws Exception {
    Files.copy(getFile("single-line.txt"), Paths.get(srcPath.toString(),
        "single-line.txt"), COPY_ATTRIBUTES);      #G
    Map<String, Object> configs = new HashMap<String, Object>();
    configs.put("inputDir", srcPath.toFile().getAbsolutePath());
    configs.put("processedDir", processedPath.toFile().getAbsolutePath());

    spySource.open(configs, mockedContext);      #H
    Thread.sleep(3000);

    Mockito.verify(spySource).consume(captor.capture());      #I
    FileRecord captured = captor.getValue();      #J
    assertNotNull(captured);
    assertEquals("It was the best of times", captured.getValue());      #K
    assertEquals("1", captured.getProperties().get(FileRecord.LINE));
    assertTrue(captured.getProperties().get(FileRecord.SOURCE)
        .contains("single-line.txt"));      #L
}

@Test
public final void multiLineTest() throws Exception {
    Files.copy(getfile("example-1.txt"), Paths.get(srcPath.toString(),
        "example-1.txt"), COPY_ATTRIBUTES);
    Map<String, Object> configs = new HashMap<String, Object>();
    configs.put("inputDir", srcPath.toFile().getAbsolutePath());
    configs.put("processedDir", processedPath.toFile().getAbsolutePath());

    spySource.open(configs, mockedContext);      #M
    Thread.sleep(3000);

    Mockito.verify(spySource, times(113)).consume(captor.capture());      #N

    final AtomicInteger counter = new AtomicInteger(0);
    captor.getAllValues().forEach(rec -> {
        assertNotNull(rec.getValue());
        assertEquals(counter.incrementAndGet() + "",
            rec.getProperties().get(FileRecord.LINE));
        assertTrue(rec.getProperties().get(FileRecord.SOURCE)
            .contains("example-1.txt"));      #O
    });
}

private static Path getFile(String fileName) throws IOException {
    ...
}
}

#A Using the tmp folder for testing
#B We will spy on the DirectorySource connector
#C Class that captures all of the records written by the DirectorySource connector
#D Clear out the tmp folder before running each test
#E Create the source and processed folder used during the test
#F Instantiate the DirectorySource connector
#G Copy the test file into the source directory

```

```

#H Run the DirectorySource connector
#I Verify that a single record was published
#J Retrieve the published record for validation
#K Validate the record contents
#L Validate the record properties
#M Run the DirectorySource connector
#N Verify that the expected number of records were published
#O Validate each of the records values and properties

```

As you can see these unit tests cover the very basic functionality of Function and rely on the use of a Mock object for the Pulsar context object. This type of test suite is quite similar to one you would write to test any Java class that wasn't a Pulsar Function.

5.3.2 Integration Testing

After we are satisfied with our unit testing results, we will want to see how the Pulsar Function will perform on a Pulsar cluster. The easiest way to test a Pulsar Function is to start a Pulsar server and run the Pulsar Function locally using the LocalRunner helper class. In this mode, the function runs as a standalone process on the machine it is submitted from. This option is best when you are developing and testing your Connectors as it allows you to attach a debugger to the Connector process on the local machine.

In order to use the LocalRunner, you must first add a few dependencies to your maven project as shown in Listing 5.10, which brings in the LocalRunner class that is used to test the Function against a running Pulsar cluster.

Listing 5.10 Including the LocalRunner Dependencies

```

<dependencies>
    ...
    <dependency>
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-core</artifactId>
        <version>2.11.1</version>
    </dependency>
    <dependency>
        <groupId>org.apache.pulsar</groupId>
        <artifactId>pulsar-functions-local-runner-original</artifactId>
        <version>2.6.1</version>
    </dependency>
</dependencies>

```

Next we need to write a class to configure our and launch the LocalRunner, as shown in Listing 5.11. As you can see this code must first configure the Pulsar Connector to execute on the LocalRunner and specifies the address of the actual Pulsar cluster instance that will be used for the testing. The easiest way to gain access to a Pulsar cluster is to launch the Pulsar Docker container like we have done in previous by running the following command in a bash window; `docker run -d -p 6650:6650 -p 8080:8080 --name pulsar apachepulsar/pulsar-standalone`, which will start a Pulsar cluster in standalone mode inside the container.

Listing 5.11 Testing the DirectorySource with the LocalRunner

```
public class DirectorySourceLocalRunnerTest {
    final static String BROKER_URL = "pulsar://localhost:6650";
    final static String OUT = "persistent://public/default/directory-scan";
    final static Path SOURCE_DIR =
        Paths.get(System.getProperty("java.io.tmpdir"), "source");
    final static Path PROCESSED_DIR =
        Paths.get(System.getProperty("java.io.tmpdir"), "processed")      #A

    private static LocalRunner localRunner;
    private static Path srcPath, processedPath;

    public static void main(String[] args) throws Exception {
        init();
        startLocalRunner();
        shutdown();
    }

    private static void startLocalRunner() throws Exception {
        localRunner = LocalRunner.builder()
            .brokerServiceUrl(BROKER_URL)      #B
            .sourceConfig(getSourceConfig())    #C
            .build();
        localRunner.start(false);
    }

    private static void init() throws IOException {
        Files.deleteIfExists(SOURCE_DIR);
        Files.deleteIfExists(PROCESSED_DIR);
        srcPath = Files.createDirectory(SOURCE_DIR,
            PosixFilePermissions.asFileAttribute(
                PosixFilePermissions.fromString("rwxrwxrwx")));
        processedPath = Files.createDirectory(PROCESSED_DIR,
            PosixFilePermissions.asFileAttribute(
                PosixFilePermissions.fromString("rwxrwxrwx")));      #D

        Files.copy(getFile("example-1.txt"), Paths.get(srcPath.toString(),
            "example-1.txt"), COPY_ATTRIBUTES);      #E
    }

    private static void shutdown() throws Exception {      #F
        Thread.sleep(30000);
        localRunner.stop();
        System.exit(0);
    }

    private static SourceConfig getSourceConfig() {
        Map<String, Object> configs = new HashMap<String, Object>();
        configs.put("inputDir", srcPath.toFile().getAbsolutePath());
        configs.put("processedDir", processedPath.toFile().getAbsolutePath());

        return SourceConfig.builder()
            .className(DirectorySource.class.getName())      #G
            .configs(configs)      #H
            .name("directory-source")
            .tenant("public")
            .namespace("default")
            .topicName(OUT)        #I
    }
}
```

```

        .build();
    }

    private static Path getFile(String fileName) throws IOException {
        . . . #J
    }
}

```

#A Using the tmp folder for testing
#B Connect the LocalRunner to the Docker container
#C Deploy the DirectorySource connector
#D Create the source and processed folder used during the test
#E Copy the test file into the source directory
#F Stops the LocalRunner after 30 seconds
#G Specify the DirectorySource as the connector we want to run
#H Configure the DirectorySource connector
#I Specifies the output topic for the source connector
#J Reads the file from the project resources folder.

Typically, you would run the LocalRunner test from inside your IDE in order to attach a debugger and step through the function code in order to identify and resolve any errors you have encountered.

5.3.3 Packaging Pulsar IO Connectors

Since Pulsar IO Connectors specialized Pulsar Functions, they are expected to be self-contained software bundles. Thus, you will also need to package your connector along with all of its dependencies as either a "fat JAR" or a NAR file. NAR stands for NiFi Archive. It is a custom packaging mechanism used by Apache NiFi, that provides Java ClassLoader isolation. In order to have your Pulsar IO Connector packaged as a NAR file, all that is required is to include the `nifi-nar-maven-plugin` in your maven project for your connector as shown in Listing 5.12.

Listing 5.12 Creating a NAR package

```

<build>
    ...
    <plugin>
        <groupId>org.apache.nifi</groupId>
        <artifactId>nifi-nar-maven-plugin</artifactId>
        <version>1.2.0</version>
        <extensions>true</extensions>
        <executions>
            <execution>
                <phase>package</phase>
                <goals>
                    <goal>nar</goal>
                </goals>
            </execution>
        </executions>
    </plugin>
</build>

```

The build plugin in Listing 5.12 is used to generate a NAR file, which by default includes all of the project dependencies in the generated archive file. This is the preferred method for bundling and deploying Java-based Pulsar IO Connectors. With this plug-in added to your pom.xml file, all that you need to do is run the `mvn clean install` command in order to produce a NAR file that can be used to deploy your connector onto a production Pulsar cluster. Once you have packaged up your connector along with all of its dependencies inside a NAR file, the next step is to deploy the connector to a Pulsar cluster.

5.4 Deploying Pulsar IO Connectors

As specialized Pulsar Functions, IO Connectors utilize the same runtime environment that provides all the benefits of the Pulsar Functions framework including fault tolerance, parallelism, elasticity, load balancing, on-demand updates, and much more.

With respect to deployment options, you can either have the Pulsar IO Connector run on your local development machine, aka **localrun mode**, or inside the Function workers in the Pulsar cluster, aka **cluster mode**. In the previous section, we were using the LocalRunner to run our connector in localrun mode. In this section I will walk you through the process of running the `DirectorySource` Connector we developed in cluster mode.

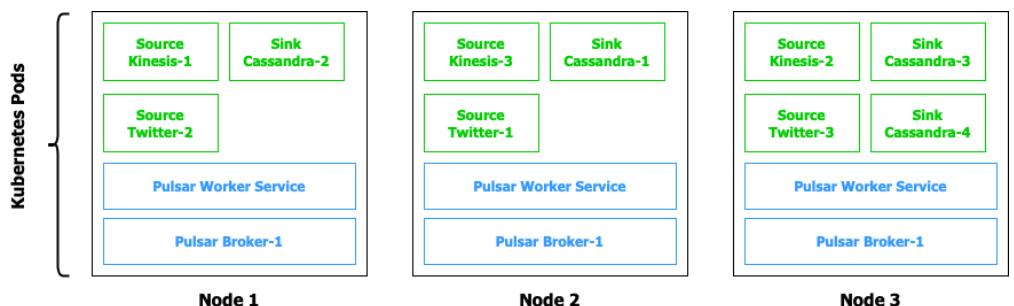


Figure 5.5: Pulsar IO Connector deployment on Kubernetes

Figure 5.5 shows a cluster mode deployment inside a Kubernetes environment, where each connector runs in its own container alongside other non-connector Function instances. In cluster mode, Pulsar IO Connectors leverage the fault-tolerance capability offered by the Pulsar Functions runtime scheduler to handle failures. If a connector is running on a machine that fails, Pulsar will automatically attempt to restart the task on one of the remaining running nodes in the cluster.

5.4.1 Creating and Deleting Connectors

If you haven't already done so, please run the `mvn clean install` command to create the NAR file for the `DirectorySource` connector. Also, you will want to stop any running Docker Pulsar containers, as you will need to start a new instance with some additional parameters,

as shown in Listing 5.13, that allow you to access the NAR file from inside the Pulsar Docker container.

Listing 5.13 Starting Pulsar Docker Container with Mounted Volumes

```
$ export GIT_PROJECT=<CLONE_DIR>/pulsar-in-action/chapter5      #A
$ docker run --name pulsar -id \
  -p 6650:6650 -p 8080:8080 \
  -v $GIT_PROJECT:/pulsar/dropbox    #B
  apachepulsar/pulsar-standalone
```

#A Set this to the directory where you cloned the books associated repo.

#B Makes the project directory accessible inside the Docker container

As you can see from listing 5.13, we have added a -v switch to the usual command we have been using to launch a Pulsar Docker container. That switch mounts the local directory where you cloned the source code for this chapter onto your machine to a folder named /pulsar/dropbox inside the Docker container itself. This is necessary in order to deploy the connector, since the NAR file has to be physically accessible by the Pulsar cluster in order for it to be deployed. We also will use this mounted directory to access the configuration file that must be provided when creating a connector.

As you may have noticed, we have always provided hard-coded values for the configuration property inside the unit an integration tests, but when deploying to production, we will want to specify those values in a more dynamic manner. This is where the configuration file comes in, as it allows us to specify connector-specific configurations along with other standard connector properties such as parallelism, etc.

Listing 5.14 Contents of the DirectorySource Connector Config File

```
tenant: public
namespace: default
name: directory-source

className: com.manning.pulsar.chapter5.source.DirectorySource
topicName: "persistent://public/default/directory-scan"
parallelism: 1
processingGuarantees: ATLEAST_ONCE

# Connector specific config
configs:
  inputDir: "/tmp/input"
  processedDir: "/tmp/processed"
  frequency: 10
```

As you can see in Listing 5.14, the configuration file we are going to use to deploy the connector contains values for source and processed directories, along with several properties used by the Pulsar IO framework to create the Pulsar IO connector. The configuration file is then provided to the bin/pulsar-admin source create command when you want to create a new source connector as shown in Listing 5.15.

Listing 5.15 The Output of the Create Command

```
docker exec -it pulsar mkdir -p /tmp/input
docker exec -it pulsar chmod a+w /tmp/input
docker exec -it pulsar mkdir -p /tmp/processed
docker exec -it pulsar chmod a+w /tmp/processed
docker exec -it pulsar cp /pulsar/dropbox/src/test/resources/example-1.txt /tmp/input    #A

docker exec -it pulsar /pulsar/bin/pulsar-admin source create \    #B
--archive /pulsar/dropbox/target/chapter5-0.0.1.nar \
--source-config-file /pulsar/dropbox/src/main/resources/config.yml    #D

"Created successfully"    #E

docker exec -it pulsar /pulsar/bin/pulsar-admin source list    #F
[
  "directory-source"
]
```

#A Create the input and output folders inside the container and copy over a test file
#B Using the source create command to create the connector
#C Specifies the NAR file that contains the source connector class
#D Specifies the configuration file to use
#E A response message indicating that the source was created
#F List the active source connectors to confirm it was created

When you create the connector, you should receive reassuring “Created successfully” message to indicate that the connector was launched successfully. If you did not receive the success message, you may be given the reason as to why that was the case. If not, then you will need to debug the error as described in the next section.

5.4.2 Debugging Deployed Connectors

If you encounter any errors or unexpected behavior inside a Pulsar IO connector that have been deployed, the best place to start is the log files on the Pulsar worker node where the Connector is running. By default, all the of the Connectors startup information and captured stderr output is written to a log file. The name of the file is based upon the Connector name and matches the following pattern in a production environment; logs/functions/tenant/namespace/function-name/function-name-instance-id.log. In standalone mode, which is what we are currently using the base directory is /tmp instead of /logs, with the rest of the path staying the same.

Let’s examine the log file for the DirectorySource connector that we created in the previous section, shown in Listing 5.16, and review some of the information that is available to you for debugging.

Listing 5.16 The first section of the DirectorySource log file

```
cat /tmp/functions/public/default/directory-source/directory-source-0.log    #A

20:53:30.671 [main] INFO org.apache.pulsar.functions.runtime.JavaInstanceStarter -
JavaInstance Server started, listening on 36857
```

```

20:53:30.676 [main] INFO org.apache.pulsar.functions.runtime.JavaInstanceStarter -
    Starting runtimeSpawner
20:53:30.678 [main] INFO org.apache.pulsar.functions.runtime.RuntimeSpawner -
    public/default/directory-source-0 RuntimeSpawner starting function
20:53:30.689 [main] INFO org.apache.pulsar.functions.runtime.thread.ThreadRuntime -
    ThreadContainer starting function with instance config InstanceConfig(instanceId=0,
        functionId=c368b93f-34e9-4bcf-801f-d097b1c0d173, functionVersion=247cbde2
    -b8b4-45bb-a3cb-8926c3b33217, functionDetails=tenant: "public" #B
    namespace: "default"
    name: "directory-source"
    className: "org.apache.pulsar.functions.api.utils.IdentityFunction"
    autoAck: true
    parallelism: 1
    source {
        className: "com.manning.pulsar.chapter5.source.DirectorySource" #C
        configs: #D
            "{\"processedDir\":\"/tmp/processed\", \"inputDir\":\"/tmp/input\", \"frequency\":\"2\"}"
    }
    typeClassName: "java.lang.String"
}
sink {
    topic: "persistent://public/default/directory-scan" #E
    typeClassName: "java.lang.String"
}
resources {
    cpu: 1.0
    ram: 1073741824
    disk: 10737418240
}
componentType: SOURCE
, maxBufferedTuples=1024, functionAuthenticationSpec=null, port=36857,
    clusterName=standalone, maxPendingAsyncRequests=10
00)

...
20:53:31.223 [public/default/directory-source-0] INFO
    org.apache.pulsar.functions.instance.JavaInstanceRunnable - Initialize function
    class loader for function directory-source at function cache manager,
    functionClassLoader: org.apache.pulsar.common.nar.NarClassLoader[/tmp/pulsar-
    nar/chapter5-0.0.1.nar-unpacked] #F

```

#A Examining the connector's log file
#B The connector configuration details section
#C The connector class name
#D The configuration map
#E The output topic
#F The NAR file and version we are using to deploy the Connector

The first section of the log file contains the basic information about the connector, such as tenant, namespace, name, parallelism, resources, and so on, which can be used to check whether the connector has been configured correctly or not. A little further down in the log file you should see a message indicating which artifact file that the Connector will be created from, which allows you to confirm that you are using the correct artifact file.

Listing 5.18 The Last Section of the DirectorySource log file

```
org.apache.pulsar.client.impl.ProducerStatsRecorderImpl - Starting
Pulsar producer perf with config: {    #A
    "topicName" : "persistent://public/default/directory-scan",
    "producerName" : null,
    "sendTimeoutMs" : 0,
    ...      #B
    "multiSchema" : true,
    "properties" : {
        "application" : "pulsar-source",
        "id" : "public/default/directory-source",
        "instance_id" : "0"
    }
}
20:53:33.704 [public/default/directory-source-0] INFO
    org.apache.pulsar.client.impl.ProducerStatsRecorderImpl - Pulsar client config: {
    #C
    "serviceUrl" : "pulsar://localhost:6650",
    "authPluginClassName" : null,
    "authParams" : null,
    ...      #D
    "proxyProtocol" : null
}
20:53:33.726 [public/default/directory-source-0] INFO
    org.apache.pulsar.client.impl.ProducerImpl - [persistent://public/
default/directory-scan] [null] Creating producer on cnx [id: 0bcd9978b, L:/127.0.0.1:44010
- R:localhost/127.0.0.1:6650]
20:53:33.886 [pulsar-client-io-1-1] INFO org.apache.pulsar.client.impl.ProducerImpl -
[persistent://public/default/direc
tory-scan] [standalone-0-0] Created producer on cnx [id: 0bcd9978b, L:/127.0.0.1:44010 -
R:localhost/127.0.0.1:6650]
20:53:33.983 [public/default/directory-source-0] INFO function-directory-source -
Scheduled to run every 2 minutes    #E
20:53:33.985 [pool-6-thread-1] INFO function-directory-source - Scanning for files....
20:53:33.987 [pool-6-thread-1] INFO function-directory-source - Processing file example-
1.txt
20:53:33.987 [pool-6-thread-1] INFO function-directory-source - Consuming file example-
1.txt
20:53:34.385 [pool-6-thread-1] INFO function-directory-source - Processed 113 lines from
example-1.txt
20:53:34.385 [pool-6-thread-1] INFO function-directory-source - Moved file
/tmp/input/example-2.txt to /tmp/processed
```

#A The Pulsar Producer for the Source Connector

#B Additional source connector properties

#C The Pulsar client configuration, including security settings

#D Additional Pulsar client configuration properties

#E Log messages from the DirectorySource Connector.

The next section of the log file, shown in Listing 5.18, contains some information about the Pulsar producers and consumers that are created on behalf of the Connector and will be used to publish and consume data from the configured input and output topics. Any connectivity issues with either of these will result in errors at this point. All the log statements added to your code will follow this section and allow you to monitor the progress of your Connector or see any of the exceptions that were raised.

When you are finished with the connector and don't want it to run any more, you can use the `bin/pulsar-admin source delete` command to stop all the running instances of the connector. The only parameters you need to provide are the connectors tenant, namespace, and name in order to uniquely identify the connector you wish to delete, e.g., in order to delete the source that we just created you would simply execute the following command;

```
bin/pulsar-admin source delete --tenant public --namespace default --name directory-source
```

5.5 Pulsar's Built-In Connectors

Pulsar provides a wide variety of existing sources and sinks that are collectively referred to as "built-in connectors" that you can use to get started using the Pulsar IO connector framework without having to write any code. Pulsar releases all the built-in connectors as individual archives. All that is required to use these connectors is a copy the built-in connector's archive (NAR) file on your Pulsar cluster and a simple YAML or JSON configuration file that specifies the runtime parameters used to connect to the external system. If you are running Pulsar in standalone mode, as we are by using the `pulsar-standalone` docker image then these built-in connectors individual archives are already included as part of the distribution.

Let's walk through a simple scenario that uses these built-in connectors to move data from Pulsar into MongoDB. While this example is a bit simplistic in nature, it will demonstrate how easy it is to use the connectors framework and help demonstrate some of the high-level steps required to deploy and use Pulsar IO connectors. The first step in this process will be to create an instance of MongoDB that we can interact with.

5.5.1 Launching the MongoDB Cluster

The following command will run the latest MongoDB container in detached mode for us. We are also mapping the container ports with host ports so that way we can access the database from our local machine if we wanted to. Once the container has launched, we will have a functional MongoDB deployment available for us to work with.

```
$ docker run -d \
-p 27017-27019:27017-27019 \
--name mongodb \
mongo
```

At this point we will have a MongoDB Docker container currently running in detached mode. Next, you will need to execute the `"mongo"` command in order to launch the MongoDB shell client. Once inside the shell we will need to create a new database and collection to store the data. Next, we will need to create a new database named `"pulsar_in_action"` and define a collection inside the database that we will use for storing the data using the commands shown in Listing 5.14.

Listing 5.14 Creating a Mongo Database Table

```
docker exec -it mongodb mongo      #A
MongoDB shell version v4.4.1      #B
...
>

>use pulsar_in_action;    #C
switched to db pulsar_in_action

> db.example.save({ firstname: "John", lastname: "Smith"})      #D
WriteResult({ "nInserted" : 1 })

> db.example.find({firstname: "John"})    #E
{ "_id" : ObjectId("5f7a53aedccb229a78960d2c"), "firstname" : "John", "lastname" : "Smith"
 }
```

#A Start the MongoDB interactive shell

#B Among the output you should see the MongoDB shell version

#C Creates a database with the name pulsar_in_action

#D Creates a collection named example inside the database and defines the schema

#E Query the database to confirm that the record was added successfully.

Now that we have a MongoDB cluster running locally, and a database created we can proceed with configuring a MongoDB sink connector so that it will read messages from a Pulsar topic and write the messages into a MongoDB table we created.

5.5.2 Link the Pulsar and MongoDB Containers

Since we are going to run the MongoDB Pulsar connector inside the Pulsar Docker container, there must be network connectivity between the two containers. The easiest way to accomplish this is Docker is by using the `--link` command line argument when launching the Pulsar container. However, since we already started the Pulsar container, we will first need to stop it and remove it before re-starting it with the `--link` switch. Therefore, you will need to execute all of the following commands shown in Listing 5.15 before proceeding.

Listing 5.15 Commands to Link Pulsar and MongoDB Containers

```
$ docker stop pulsar    #A

$ docker rm pulsar     #B

$ docker run -d \
  -p 6650:6650 -p 8080:8080 \
  -v $PWD/data:/pulsar/data \
  --name pulsar \
  --link mongodb \
  apachepulsar/pulsar-standalone    #C

$ docker exec -it pulsar bash    #D

apt-get update && apt-get install vim --fix-missing -y    #E
vim /pulsar/examples/mongodb-sink.yml #F
```

```

#A Stops the currently running Pulsar container
#B Deletes the old Pulsar container so we can create a new one with the same name
#C Links the mongodb container to the Pulsar container
#D Exec into the new Pulsar container
#E We need to install the vim text editor in the Pulsar container so we can edit the config file.
#F Launch the text editor inside the Pulsar container to create the configuration file.

```

By providing the name of the container running the MongoDB instance we wish to interact with in the `--link` switch, Docker creates a secure network channel between the two containers that allows the Pulsar container to talk to the MongoDB container via the link name, as we shall see when we configure the MongoDB Sink connector.

5.5.3 Configure and create the MongoDB Sink

Configuring Pulsar IO connectors is straightforward. All you need to do is to provide a yaml configuration file when you create the connectors. In order to run a MongoDB sink connector, you will need to prepare a yaml config file containing all the information that the Pulsar IO runtime needs to know in order to connect to the local MongoDB instance. First, you need to create a local file in the `examples` sub-directory named `mongodb-sink.yml` and edit it to have the following content shown in Listing 5.16.

Listing 5.16 The MongoDB Sink Connector Configuration File

```

tenant: "public"
namespace: "default"
name: "mongo-test-sink"
configs:
    mongoUri: "mongodb://mongodb:27017/admin"      #A
    database: "pulsar_in_action"      #B
    collection: "example"      #C
    batchSize: 1
    batchTimeMs: 1000

```

#A We can us the name specified with the `-link` switch here instead of a hostname or IP address.
#B We must specify the Mongo database we are writing to
#C We must specify the Mongo collection we are writing to

For more information on the MongoDB sink connector configuration, please refer to the [documentation](#). The Pulsar command line interface provides commands for running and managing Pulsar IO connectors, so you can run the command shown in Listing 5.17 from the Pulsar container command line to start the MongoDB sink connector.

Listing 5.17 Starting the MongoDB Sink Connector

```

/pulsar/bin/pulsar-admin sink create \
    --sink-type mongo \
    --sink-config-file /pulsar/examples/mongodb-sink.yml \
    --inputs test-mongo \
    "Created successfully"

```

#A Using the sink create command
#B Indicates we want to use the built-in Sink connector for MongoDB
#C Use the configuration file we created earlier

#D Specifies the input topic
#E A response message indicating that the sink was created.

Once the command is executed, Pulsar will create a sink connector named mongo-test-sink and the sink connector will be running as a Pulsar Function and write the messages produced in topic `test-mongo` to the MongoDB collection examples in the `pulsar_in_action` database. Now let's send some messages to the `test-mongo` topic to confirm that the connector is functioning as expected by executing the commands shown in Listing 5.18 from inside the docker container.

Listing 5.18 Sending messages to the Connector's Input Topic

```
/pulsar/bin/pulsar-client produce \  
-m "{\"firstname: \"Mary\", lastname: \"Smith\"}" \  
-s % \  
-n 10 \  
test-mongo
```

#A We are producing messages

#B The message contents, including escaped quotes

#C Defines a non-comma record separator character, otherwise the message contents would be split

#D Specifies we want to send the same message 10 times

#E The destination topic

You can now query the mongo DB instance to confirm that the MongoDB connector worked as expected. Return to the MongoDB shell that we opened earlier to create the database and run some different queries as shown in Listing 5.19 to confirm that the records were added to the MongoDB table as expected.

Listing 5.19 Querying the MongoDB table after the messages were sent

```
> db.example.find({lastname: "Smith"}) \  
{ "_id" : ObjectId("5f7a53aedccb229a78960d2c"), "firstname" : "John", "lastname" : "Smith"  
} \  
{ "_id" : ObjectId("5f7a68bbb94aa03489fa5ca9"), "firstname" : "Mary", "lastname" : "Smith"  
} \  
{ "_id" : ObjectId("5f7a68bbb94aa03489fa5caa"), "firstname" : "Mary", "lastname" : "Smith"  
} \  
{ "_id" : ObjectId("5f7a68bbb94aa03489fa5cab"), "firstname" : "Mary", "lastname" : "Smith"  
} \  
{ "_id" : ObjectId("5f7a68bbb94aa03489fa5cac"), "firstname" : "Mary", "lastname" : "Smith"  
} \  
{ "_id" : ObjectId("5f7a68bbb94aa03489fa5cad"), "firstname" : "Mary", "lastname" : "Smith"  
} \  
{ "_id" : ObjectId("5f7a68bbb94aa03489fa5cae"), "firstname" : "Mary", "lastname" : "Smith"  
} \  
{ "_id" : ObjectId("5f7a68bbb94aa03489fa5caf"), "firstname" : "Mary", "lastname" : "Smith"  
} \  
{ "_id" : ObjectId("5f7a68bbb94aa03489fa5cb0"), "firstname" : "Mary", "lastname" : "Smith"  
} \  
{ "_id" : ObjectId("5f7a68bbb94aa03489fa5cb1"), "firstname" : "Mary", "lastname" : "Smith"  
} \  
{ "_id" : ObjectId("5f7a68bbb94aa03489fa5cb2"), "firstname" : "Mary", "lastname" : "Smith"
```

```
#A Query by lastname field  
#B The original record we published  
#C Ten instances of the new record created from the messages we just sent.
```

This concludes our quick introduction to the built-in Pulsar IO connectors, and now you should have a better understanding of how to configure and deploy Pulsar connectors, and how the overall IO connectors framework works.

5.6 Administering Pulsar IO Connectors

The pulsar admin CLI tool provides a collection of commands that enables you to manage, monitor, and update Pulsar IO connectors. We will discuss some of these commands that were designed specifically for Pulsar IO connectors including how and when they should be used and what information they provide. It is worth noting that both the sink and source commands have the exact same sub-commands, so we will only focus on the sink command, but the information is applicable to source connectors as well.

5.6.1 Listing Connectors

The next command we will look at is the `pulsar-admin sink list` command, which will return a list of all the sinks currently running on the Pulsar cluster which is useful when you want to make sure that the connector you just created was accepted and is running. If you were to run this command after you deployed the `mongo-test-sink` connector, the expected output would be similar to what is shown in Listing 5.20.

Listing 5.20 Output of the list command inside the Docker Container

```
docker exec -it pulsar /pulsar/bin/pulsar-admin sink list  
[  
    "mongo-test-sink"  
]
```

Which shows that the `mongo-test-sink` was indeed created and is the only sink connector currently running in the Pulsar cluster. The `list` command is not to be confused with the `available-sources`, or `available-sinks` commands, which will return a list of all the built-in connectors that are supported by the Pulsar cluster. By default, the built-in connectors are included in the Pulsar standalone Docker container, so the output of the command should be as shown in Listing 5.21.

Listing 5.21 Output of the available-sinks command inside the Pulsar Docker container

```
docker exec -it pulsar /pulsar/bin/pulsar-admin sink available-sinks  
aerospike  
Aerospike database sink  
-----  
cassandra  
Writes data into Cassandra  
-----  
data-generator  
Test data generator source
```

```
-----  
elastic_search  
Writes data into Elastic Search  
-----  
flume  
flume source and sink connector  
-----  
hbase  
Writes data into hbase table  
-----  
hdfs2  
Writes data into HDFS 2.x  
-----  
hdfs3  
Writes data into HDFS 3.x  
-----  
influxdb  
Writes data into InfluxDB database  
-----  
jdbc-clickhouse  
JDBC sink for ClickHouse  
-----  
jdbc-mariadb  
JDBC sink for MariaDB  
-----  
jdbc-postgres  
JDBC sink for PostgreSQL  
-----  
jdbc-sqlite  
JDBC sink for SQLite  
-----  
kafka  
Kafka source and sink connector  
-----  
kinesis  
Kinesis connectors  
-----  
mongo  
MongoDB source and sink connector  
-----  
rabbitmq  
RabbitMQ source and sink connector  
-----  
redis  
Writes data into Redis  
-----  
solr  
Writes data into solr collection  
-----
```

The `available-sinks` command can also help you confirm that you have successfully installed a custom connector manually.

5.6.2 Monitoring Connectors

Another useful command for monitoring Pulsar IO connectors is the `pulsar-admin sink status` command, which returns runtime information about the specified connector such as

how many instances are running, and if there are any of the instances have encountered errors.

Listing 5.22 Output of the Sink status command

```
docker exec -it pulsar /pulsar/bin/pulsar-admin sink status \
--name mongo-test-sink
{
  "numInstances" : 1,      #A
  "numRunning" : 1,       #B
  "instances" : [ {       #C
    "instanceId" : 0,
    "status" : {
      "running" : true,   #D
      "error" : "",       #E
      "numRestarts" : 0,  #F
      "numReadFromPulsar" : 0,  #G
      "numSystemExceptions" : 0,
      "latestSystemExceptions" : [ ],
      "numSinkExceptions" : 0,
      "latestSinkExceptions" : [ ],
      "numWrittenToSink" : 0,
      "lastReceivedTime" : 0,  #H
      "workerId" : "c-standalone-fw-d513daf5b94e-8080"
    }
  } ]
}
```

#A The total number of instances of the connector that were requested

#B The total number of instances of the connector that are running

#C An array of information for each instance

#D The current status of the connector

#E Any applicable error messages

#F The number of times the connector attempted to restart. This number increases whenever the connector failed to start, and it is re-launched

#G The number of messages consumed from the Pulsar input topic by this instance

#H The last time an incoming message was consumed by this instance

As you can see from Listing 5.22, the `pulsar-admin sink status` command would be particularly useful for checking on the status of a connector immediately after you have deployed it to make sure it started properly. While the `pulsar-admin sink get` command can be used to return the configuration information about a Pulsar IO connector as shown, which is useful when you want to inspect the configuration settings of your connector to ensure that it is properly configured as shown in Listing 5.23.

Listing 5.23 Output of the Sink get command

```
docker exec -it pulsar /pulsar/bin/pulsar-admin sink get \
--name mongo-test-sink
{
  "tenant": "public",
  "namespace": "default",
  "name": "mongo-test-sink",      #A
  "className": "org.apache.pulsar.io.mongodb.MongoSink",      #B
  "inputSpecs": {
    "test-mongo": {      #C

```

```

        "isRegexPattern": false    #D
    },
},
"configs": {    #E
    "mongoUri": "mongodb://mongodb:27017/admin",
    "database": "pulsar_in_action",
    "collection": "example",
    "batchSize": "1.0",
    "batchTimeMs": "1000.0"
},
"parallelism": 1,    #F
"processingGuarantees": "ATLEAST_ONCE",
"retainOrdering": false,
"autoAck": true,
"archive": "builtin://mongo"
}

```

#A The tenant, namespace, and name of the connector

#B The Classname of the connector implementation

#C The input topics for the sink connector

#D Whether or not the sink is configured to consume from multiple topics based on some regular expression.

#E All of the user defined configuration properties we provided in the sink-config-file

#F All of the default property values for properties we did not specify.

What makes this command even more useful is the fact that the output of the command is a properly formatted JSON connector configuration that can be saved as a file, modified, and used to update the configuration of the running connector with the `update` command. This frees you from having to retain a copy of the configuration you deployed a specific connector with, as the data can easily be retrieved from the running connector with this command.

Listing 5.24 Updating the Mongo DB Connector

```
/pulsar/bin/pulsar-admin sink update \
    --sink-type mongo \
    --sink-config-file /pulsar/examples/mongodb-sink.yml \
    --inputs prod-mongo \
    --processing-guarantees EFFECTIVELY_ONCE
```

The `pulsar-admin sink update` command allows you to dynamically change the configuration parameters of an already submitted sink connector without having to delete and re-create it. The `update` command takes in a variety of command-line options, which are described in greater detail in the [Apache documentation](#), that allow you to change almost any of the connector's configurations including the archive file itself if you wanted to deploy a new version of the connector. This makes modifying, testing, and deploying a much more streamlined process. Listing 5.24 shows the how to update the mongo sink connector we deployed earlier to use a different Pulsar topic as the input source and change the processing guarantees.

That wraps up our quick introduction some of the commands available for monitoring and administering Pulsar IO connectors. My goal was to provide enough of a high-level overview of the capabilities provided by the framework itself that enable you to get started. I strongly recommend referring to the on-line documentation for details on the various switches and parameters for each of these commands.

5.7 Summary

- Pulsar IO Connectors are an extension of the Pulsar Functions framework specifically designed to interface with external systems such as databases.
- Pulsar IO Connectors come in two basic types: **sources** which pull data out of external systems into Pulsar; and **sinks** which publish data from Pulsar into external systems.
- Pulsar provides a set of built-in connectors that you can use to interact with several popular systems without having to write a single line of code.
- The Pulsar CLI tool allows you to administer Pulsar IO Connectors including creating, deleting, and updating connectors.

6

Pulsar Security

This chapter covers

- Encrypting data transmitted into and out of a Pulsar cluster
- Enabling client authentication using JSON Web tokens
- Encrypting data stored inside Apache Pulsar

This chapter covers how to secure your cluster in order to prevent unauthorized access to the data sent through Apache Pulsar. While the tasks I am going to cover are not important in a development environment, they are critically important for a production deployment to reduce the risk of unauthorized access to sensitive information, ensure data loss prevention, and protect your organization's public reputation. Modern systems and organizations utilize a combination of security controls and safeguards in order to provide multiple layers of defense that prevent access to data within the system. Particularly those that must maintain regulatory compliance with security regulations such as HIPPA, PCI-DSS, or GDPR just to name a few.

Pulsar integrates well with several existing security frameworks that allow you to leverage these tools to secure your Pulsar cluster at multiple levels in order to mitigate the risk of a lapse in one of the security mechanisms resulting in a total security failure. For instance, even if an unauthorized user were able to access your system with a compromised password, they would still need a valid encryption key in order to read the encrypted message data.

6.1 Transport Encryption

By default, the data transmitted between a Pulsar broker and a Pulsar client is sent in plain-text. This means that any sensitive data such as passwords, credit card numbers, and social security numbers contained within a message is susceptible to being intercepted by eavesdroppers as it is transmitted over the network. Therefore, the first layer of defense is

ensuring that the data transmitted between a Pulsar broker and a Pulsar client is encrypted BEFORE it is transmitted.

Pulsar allows you to configure all communication to use Transport Layer Security (TLS), which is a common cryptographic protocol that provides data encryption as it is transported across the network ONLY. This is why it is often referred to as encryption for “data-in-motion”, since the data is decrypted on the receiving end and therefore no longer encrypted once it reaches its destination.

ENABLING TLS ON PULSAR

Now that I have covered the basics of TLS wire encryption at a fairly high level, let’s focus on how we can use this technology to secure our communications between our Pulsar brokers and our clients. Since the Pulsar documentation does a fair job of outlining the steps required to enable TLS on Pulsar, I have decided that rather than republish those same steps here that I would capture all of those steps in a single script that can be used to automate the process inside a Docker based image.

I will then discuss the commands contained within the scripts in greater detail as they relate to the discussion that we had in the previous section so that you have a better understanding of why these steps are important, and how you might modify them to suit your needs in a true production environment. If you look inside the GitHub repo (<https://github.com/david-streamlio/pulsar-in-action>) associated with this book, you will find a Dockerfile similar to the one shown in Listing 6.1 under the *docker-image/pulsar-standalone* folder.

For those of you unfamiliar with Docker, A Dockerfile is a simple text file that contains a series of instructions that are executed sequentially by the Docker client when creating an image. They can be thought of as recipes or blueprints for building Docker images and provide a simple way to automate the image creation process. Now that we have a better understanding of Docker images, it is time to create our own. Our goal is to create an image that extends the capability of the pulsar-standalone image we used previously to include all of the security features I will be discussing throughout this chapter.

Listing 6.1 Dockerfile Contents

```
FROM apachepulsar/pulsar-standalone:latest      #A
ENV PULSAR_HOME=/pulsar

COPY conf/standalone.conf $PULSAR_HOME/conf/standalone.conf      #B
COPY conf/client.conf $PULSAR_HOME/conf/client.conf      #C

ADD manning $PULSAR_HOME/manning      #D

RUN chmod a+x $PULSAR_HOME/manning/security/*.sh \
    $PULSAR_HOME/manning/security/**/*.*.sh \
    $PULSAR_HOME/manning/security/authentication/**/*.*.sh      #E

#####
# Transport Encryption using TLS
#####
RUN ["/bin/bash", "-c",
    "/pulsar/manning/security/TLS-encryption/enable-tls.sh"]      #F
```

```
#A Use the existing pulsar-standalone image as a starting point to effectively extend its capabilities  
#B Overwrite the broker configuration with one that contains our updated security settings.  
#C Overwrite the client configuration with one that contains our updated client credentials.  
#D Copy the contents of the manning folder into the image at /pulsar/manning  
#E Give execute permission to all of the scripts we need to execute.  
#F Execute the specified script to generate the certificates required for TLS
```

I will start by specifying that I wish to use the non-secured `apache/pulsar:pulsar-standalone:latest` image as the base image by using the `FROM` keyword. For those of you that are familiar with object-oriented languages, this is effectively the same as inheriting from a base class. All of the base image's services, features, and configurations are automatically included in our image which ensures that our Docker images will also provide a complete Pulsar environment for testing purposes without having to replicate those commands in my Dockerfile.

After setting the `PULSAR_HOME` environment variable, I use the `COPY` keyword to replace both the Pulsar broker and client configuration files with ones that are properly configured to secure the Pulsar instance. This is an important step, as once a container based on this image is launched, it is impossible to change these settings and have them take effect. Next, we `ADD` the contents of the `manning` directory to the docker image and `RUN` a command to enable execute permission on all of the bash scripts that were added in the previous command so that we can execute them.

At the end of the Dockerfile are a series of bash scripts that get executed in order to generate the necessary security credentials, certificates, and keys required to secure a Pulsar cluster. Let's examine the first script, named `enable-tls.sh`, which performs all the steps necessary to enable TLS wire encryption on the Pulsar cluster.

The `pulsar-standalone` Docker image includes OpenSSL, which is an open-source library that provides several command-line tools for issuing and signing digital certificates. Since we are running in a development environment, we will use these tools to act as our own certificate authority to produce "self-signed" certificates rather than use an internationally trusted third-party CA (e.g., VeriSign, DigiCert) to sign our certificates. In a production environment you should **ALWAYS** rely on a third-party CA.

Acting as a certificate authority (CA) means dealing with cryptographic pairs of private keys and public certificates. The very first cryptographic pair we'll create is the root pair. This consists of the root key (`ca.key.pem`) and root certificate (`ca.cert.pem`). This pair forms the identity of your CA. Let's examine the first part of the `enable-tls.sh` script, which generates these. As we can see from Listing 6.2, the first command in the script generates the root key while the second command creates the public X.509 certificate using the root key that was generated in the previous step.

Listing 6.2 Portion of enable-tls.sh that creates CA

```
#!/bin/bash  
  
export CA_HOME=$(pwd)  
export CA_PASSWORD=secret-password      #A
```

```

mkdir certs crl newcerts private
chmod 700 private/
touch index.txt index.txt.attr
echo 1000 > serial

# Generate the certificate authority private key
openssl genrsa -aes256 \      #B
-passout pass:${CA_PASSWORD} \   #C
-out /pulsar/manning/security/cert-authority/private/ca.key.pem \
4096    #D

# Restrict Access to the certificate authority private key
chmod 400 /pulsar/manning/security/cert-authority/private/ca.key.pem    #E

# Create the X.509 certificate.
openssl req -config openssl.cnf \
-key /pulsar/manning/security/cert-authority/private/ca.key.pem \      #F
-new -x509 \    #G
-days 7300 \   #H
-sha256 \
-extensions v3_ca \
-out /pulsar/manning/security/cert-authority/certs/ca.cert.pem \
-subj '/C=US/ST=CA/L=Palo Alto/CN=gottaeat.com' \    #I
-passin pass:${CA_PASSWORD}    #J

```

#A We use an environment variable to set the password for the CA root key.

#B Encrypt the root key with AES 256-bit encryption

#C Generates the private key that is secured with a strong password.

#D Use 4096 bits for the root key

#E Anyone in possession of the root key and password can issue trusted certificates.

#F Generates the root certificate using the root key

#G Requests a new X.509 certificate

#H Give the root certificate a long expiry date.

#I Specify the organization for which the certificate is valid.

#J Allows us to provide the password for the root key from the command line without a prompt.

At this point the script has generated a password-protected private key (ca.key.pem), and root certificate (ca.cert.pem) for our internal CA. The script purposely generates the root certificate to a known location so that we can refer to it from inside the broker's configuration file, `/pulsar/conf/standalone.conf`. Specifically, we have preconfigured the `tlsTrustCertsFilePath` property to point to the location where the root certificate was generated. In a production environment where you are using a third-party CA, you would be provided with a certificate that can be used to authenticate the X.509 certificates and would update the property to point to that certificate instead.

Now that we have created a CA certificate, the next step is to generate a certificate for the Pulsar broker and sign it with our internal CA. When using a third-party CA, you would issue this request to them and wait for them to send you a certificate, however since we are acting as our own certificate authority, we can issue the certificate ourselves as shown in Listing 6.3.

Listing 6.3 Portion of enable-tls.sh That Generates the Pulsar Broker Certificate

```
export BROKER_PASSWORD=my-secret      #A

# Generate the Server Certificate private key
openssl genrsa -passout pass:${BROKER_PASSWORD} \\      #B
    -out /pulsar/manning/security/cert-authority/broker.key.pem \
        2048      #C

# Convert the key to PEM format
openssl pkcs8 -topk8 -inform PEM -outform PEM \\      #D
    -in /pulsar/manning/security/cert-authority/broker.key.pem \
    -out /pulsar/manning/security/cert-authority/broker.key-pk8.pem \
    -nocrypt

# Generate the server certificate request
openssl req -config /pulsar/manning/security/cert-authority/openssl.cnf \
    -new -sha256 \
    -key /pulsar/manning/security/cert-authority/broker.key.pem \
    -out /pulsar/manning/security/cert-authority/broker.csr.pem \
    -subj '/C=US/ST=CA/L=Palo Alto/O=IT/CN=pulsar.gottaeat.com' \\      #E
    -passin pass:${BROKER_PASSWORD}      #F

# Sign the server certificate with the CA
openssl ca -config /pulsar/manning/security/cert-authority/openssl.cnf \
    -extensions server_cert \\      #G
    -days 1000 -notext -md sha256 -batch \\      #H
    -in /pulsar/manning/security/cert-authority/broker.csr.pem \
    -out /pulsar/manning/security/cert-authority/broker.cert.pem \
    -passin pass:${CA_PASSWORD}      #I
```

#A We use an environment variable to set the password for the server certificate's private key.

#B Generates the private key that is secured with a strong password

#C Use 2048 bits for the private key

#D The broker expects the key to be in PKCS 8 format

#E Specify the organization and hostname for which the certificate is valid.

#F Allows us to provide the password for the key from the command line without a prompt.

#G Specify that this certificate is intended to be used by a server.

#H Give the server certificate a long expiry date.

#I We need to provide the password for the CA's private key since we are acting as the CA.

At this point, you have a broker certificate (`broker.cert.pem`), and its associated private key, (`broker.key-pk8.pem`) that you can use along with `ca.cert.pem` to configure TLS transport encryption for your broker and proxy nodes. Again, the script purposely generates the broker certificate to a known location so that we can refer to it from inside the broker's configuration file, `/pulsar/conf/standalone.conf`. Let's take a look at all the properties that were changed in order to enable TLS for Pulsar.

Listing 6.4 TLS Property Changes to the standalone.conf file

```
#### To Enable TLS wire encryption #####
tlsEnabled=true
brokerServicePortTls=6651
webServicePortTls=8443

# The Broker certificate and associated private key
```

```

tlsCertificateFilePath=/pulsar/manning/security/cert-authority/broker.cert.pem
tlsKeyFilePath=/pulsar/manning/security/cert-authority/broker.key-pk8.pem

# The CA certificate
tlsTrustCertsFilePath=/pulsar/manning/security/cert-authority/certs/ca.cert.pem

# Used for TLS negotiation to specify which ciphers we consider safe.
tlsProtocols=TLSv1.2,TLSv1.1
tls
Ciphers=TLS_DH_RSA_WITH_AES_256_GCM_SHA384,TLS_DH_RSA_WITH_AES_256_CBC_SHA

```

Since I have enabled TLS transport encryption, I also need to configure the command-line tools such as pulsar-admin, and pulsar-perf to communicate with the secure Pulsar broker by changing the following properties in the `$PULSAR_HOME/conf/client.conf` file, as shown in Listing 6.5

Listing 6.5 TLS Property Changes to the client.conf file

```

##### To Enable TLS wire encryption #####
# Use the TLS protocols and ports
webServiceUrl=https://pulsar.gottaeat.com:8443/
brokerServiceUrl=pulsar+ssl://pulsar.gottaeat.com:6651/

useTls=true
tlsAllowInsecureConnection=false
tlsEnableHostnameVerification=false
tlsTrustCertsFilePath=pulsar/manning/security/cert-authority/certs/ca.cert.pem

```

If you haven't already done so, change to the directory that contains the Dockerfile and run the following command, `docker build . -t pia/pulsar-standalone-secure:latest` to create the Docker image and tag it as shown in Listing 6.6.

Listing 6.6 Building the Docker Image from the Dockerfile

```

$ cd $REPO_HOME/pulsar-in-action/docker-images/pulsar-standalone      #A
$ docker build . -t pia/pulsar-standalone-secure:latest      #B
Sending build context to Docker daemon 3.345MB
Step 1/7 : FROM apachepulsar/pulsar-standalone:latest      #C
--> 3ed9bffff717
Step 2/7 : ENV PULSAR_HOME=/pulsar
--> Running in cf81f78f5754
Removing intermediate container cf81f78f5754
--> 48ea643513ff
Step 3/7 : COPY conf/standalone.conf $PULSAR_HOME/conf/standalone.conf
--> 6dcf0068eb40
Step 4/7 : COPY conf/client.conf $PULSAR_HOME/conf/client.conf
--> e0f6c81a10c4
Step 5/7 : ADD manning $PULSAR_HOME/manning      #D
--> e253e7c6ed8e
Step 6/7 : RUN chmod a+x $PULSAR_HOME/manning/security/*.sh
          $PULSAR_HOME/manning/security/*/*.sh
          $PULSAR_HOME/manning/security/authentication/*/*.sh
--> Running in 42d33f3e738b
Removing intermediate container 42d33f3e738b
--> ddccc85c75f4
Step 7/7 : RUN ["/bin/bash", "-c", "/pulsar/manning/security/TLS-encryption/enable-tls.sh"]

```

```

#E
---> Running in 5f26f9626a25
Generating RSA private key, 4096 bit long modulus
.....+++++
.....+++++
.....+++++
e is 65537 (0x010001)
Generating RSA private key, 2048 bit long modulus
.....+++++
.....+++++
e is 65537 (0x010001)
Using configuration from /pulsar/manning/security/cert-authority/openssl.cnf
Check that the request matches the signature
Signature ok
Certificate Details: #F
    Serial Number: 4096 (0x1000)
    Validity
        Not Before: Jan 13 00:55:03 2020 GMT
        Not After : Oct  9 00:55:03 2022 GMT      #G
    Subject:
        countryName          = US
        stateOrProvinceName = CA
        organizationName     = gottaeat.com
        commonName           = pulsar.gottaeat.com
    X509v3 extensions:
        X509v3 Basic Constraints:
            CA:FALSE
        Netscape Cert Type:
            SSL Server      #H
        Netscape Comment:
            OpenSSL Generated Server Certificate
    X509v3 Subject Key Identifier:
        DF:75:74:90:34:C6:0D:F0:9B:E7:CA:07:0A:37:B8:6F:D7:DF:52:0A
    X509v3 Authority Key Identifier:
        keyid:92:F0:6D:0F:18:D4:3C:1E:88:B1:33:3A:9D:04:29:C0:FC:81:29:02
        DirName:/C=US/ST=CA/L=Palo Alto/O=gottaeat.com
        serial:93:FD:42:06:D8:E9:C3:89

        X509v3 Key Usage: critical
            Digital Signature, Key Encipherment
        X509v3 Extended Key Usage:
            TLS Web Server Authentication
Certificate is to be certified until Oct  9 00:55:03 2022 GMT (1000 days)

Write out database with 1 new entries
Data Base Updated
Removing intermediate container 5f26f9626a25
--> 0e7995c14208
Successfully built 0e7995c14208
Successfully tagged pia/pulsar-standalone-secure:latest      #I

```

#A Change to the directory that contains the Dockerfile

#B Command to build the Docker image from the Dockerfile and tag it.

#C Pulls down the apachepulsar/pulsar-standalone:latest image from the public repository

#D Copies the entire contents of the manning directory into the Docker image

#E Executes the enable-tls.sh script

#F The details of the server certificate that is generated by the enable-tls.sh script.

```
#G The expiration date of the certificate  
#H Indicates that the certificate generated can be used as a server certificate.  
#! Success message including the tag used to reference the image.
```

Once the `enable-tls.sh` script has been executed and the properties all configured to point to the correct values, the pulsar standalone image will only accept connections over a secure, TLS channel. You can verify this by using the sequence of commands shown in Listing 6.7 to launch a container with the new Docker image. Notice that I am using the `-volume` switch to create a logical mount point between my laptop's `$HOME` directory and a directory inside the docker container itself. This allows me to publish the TLS client credentials that only exist inside the container to a location on my machine where I can access them.

Next, I need to ssh into the container and run the `publish-credentials.sh` script inside the newly launched bash session in order to make these credentials available to me in the `$(HOME)/exchange` folder on my local machine.

Listing 6.7 Publishing the TLS Credentials

```
$ docker run -id --name pulsar -p:6651:6651 -p 8443:8443\      #A  
  -volume=${HOME}/exchange:/pulsar/manning/dropbox \      #B  
  -t pia/pulsar-standalone-secure:latest      #C  
$ docker exec -it pulsar bash  
$ /pulsar/manning/security/publish-credentials.sh      #D  
$ exit      #E
```

```
#A Use the SSL ports of 6651 and 8443  
#B Using the volume switch to allow me to copy files from the container to my local machine  
#C Specify the image that we just built.  
#D Run the script that copies the generated credentials to my local machine.  
#E Create the default namespace to use for testing.  
#F Exit the container
```

Next, I will attempt to connect to it over the TLS secured port (6651) by using the following Java program, which is available in the GitHub repo associated with this book.

Listing 6.8 Using the TLS Credentials to Connect to Pulsar

```
import org.apache.pulsar.client.api.*;  
public class TlsClient {  
    public static void main(String[] args) throws PulsarClientException {  
        final String HOME = "/Users/david/exchange";  
        final String TOPIC = "persistent://public/default/test-topic";  
  
        PulsarClient client = PulsarClient.builder()  
            .serviceUrl("pulsar://localhost:6651/")  
            .tlsTrustCertsFilePath(HOME + "/ca.cert.pem")  
            .build();  
  
        Producer<byte[]> producer =  
            client.newProducer().topic(TOPIC).create();  
  
        for (int idx = 0; idx < 100; idx++) {  
            producer.send("Hello TLS".getBytes());  
        }  
    }  
}
```

```
        System.exit(0);
    }
}

#A Specify the pulsar+ssl protocol, failure to do so will result in a connection failure.
#B The location of the file containing the trusted TLS certificates.
```

That concludes the configuration of TLS wire encryption on the Pulsar Broker. From this point forward, all communication with the Broker will be over SSL and all traffic will be encrypted to prevent unauthorized access to the data contained inside the messages that are published to and consumed from the Pulsar Broker. You can experiment some more with this container, but once you are finished you should be sure to run `docker stop pulsar && docker rm pulsar` in order to stop the container and remove it. Both of these steps are necessary in order rebuild the Docker image in the next section to enable support for Authentication in the Pulsar standalone image.

6.2 Authentication

An authentication service provides a way for a user to confirm their identity by providing some credentials, such as a username and password, to validate you are who you claim to be. Pulsar supports a pluggable authentication mechanism which clients can use to authenticate themselves. Currently, Pulsar supports different authentication providers. In this section, I will walk through the steps required to configure both TLS authentication and JSON web token authentication.

6.2.1 TLS Authentication

In TLS Client Authentication, the client uses a certificate to authenticate itself. Obtaining a certificate requires interaction with a Certification Authority (CA) that will issue a certificate that can be trusted by the Pulsar Broker. For a client certificate to pass a server's validation process, the digital signature found on it should have been signed by a CA recognized by the server. Therefore, I have used the same CA that issued the server certificate in the previous section to generate the client certificates to ensure that the client certificates are "trusted".

With TLS client authentication, the client generates a keypair for authentication purpose and retains the private key of the keypair in a secure location. The client then issues a certificate request to a trusted CA and receives back an X.509 digital certificate.

These client certificates typically contain pertinent information like a digital signature, expiration date, name of client, name of CA (Certificate Authority), revocation status, SSL/TLS version number, serial number, common name, and possibly more, all structured using the X.509 standard. Pulsar uses the common name field of the certificate to map the client to a specific role, which is used to determine what actions the client is authorized to perform.

When a client attempts to connect to a Pulsar Broken that has TLS authentication enabled, it can submit a client certificate for authentication as part of the TLS handshake. Upon receiving the certificate, the Pulsar Broker uses it to identify the certificate's source and determine whether the client should be granted access.

Don't confuse client certificates with the server certificate we used to enable TLS wire encryption. Both are X.509 digital certificates but they are two different things. A server certificate is sent from the Pulsar Broker to the client at the start of a session and is used by the client to authenticate the server. A client certificate, on the other hand, is sent from the client to the Broker at the start of a session and is used by the server to authenticate the client.

In order to enable TLS-based authentication, I append a command to the Dockerfile from the previous section that executes another script named `gen-client-certs.sh`, that generates TLS client certificates that can be used for authentication as shown in Listing 6.9

Listing 6.9 Updated Dockerfile Contents

```
FROM apachepulsar/pulsar-standalone:latest
ENV PULSAR_HOME=/pulsar

...      #A

RUN ["/bin/bash", "-c",
     "/pulsar/manning/security/TLS-encryption/enable-tls.sh"]

RUN ["/bin/bash", "-c",
     "/pulsar/manning/security/authentication/tls/gen-client-certs.sh"]      #B
```

#A Same content as shown in Figure 6.1

#B Execute the specified script to generate TLS client certs for role-based authentication.

Let's take a look at the `gen-client-certs.sh` script in Listing 6.10, to see exactly what steps are required in order to generate TLS client certificates that can be used to authenticate with the Pulsar broker.

Listing 6.10 Contents of the gen-client-certs.sh File

```
#!/bin/bash

cd /pulsar/manning/security/cert-authority
export CA_HOME=$(pwd)
export CA_PASSWORD=secret-password

function generate_client_cert() {

    local CLIENT_ID=$1      #A
    local CLIENT_ROLE=$2      #B
    local CLIENT_PASSWORD=$3      #C

    # Generate the Client Certificate private key
    openssl genrsa -passout pass:${CLIENT_PASSWORD} \      #D
        -out /pulsar/manning/security/authentication/tls/${CLIENT_ID}.key.pem \
        2048

    # Convert the key to PEM format
    openssl pkcs8 -topk8 -inform PEM -outform PEM -nocrypt \
        -in /pulsar/manning/security/authentication/tls/${CLIENT_ID}.key.pem \
        -out /pulsar/manning/security/authentication/tls/${CLIENT_ID}-pk8.pem
```

```

# Generate the client certificate request
openssl req -config /pulsar/manning/security/cert-authority/openssl.cnf \
-key /pulsar/manning/security/authentication/tls/${CLIENT_ID}.key.pem
-out /pulsar/manning/security/authentication/tls/${CLIENT_ID}.csr.pem \
-subj "/C=US/ST=CA/L=Palo Alto/O=gottaeat.com/CN=${CLIENT_ROLE}" \
-new -sha256 \
-passin pass:${CLIENT_PASSWORD}      #F

# Sign the server certificate with the CA
openssl ca -config /pulsar/manning/security/cert-authority/openssl.cnf \
-extensions usr_cert \
-days 100 -notext -md sha256 -batch \
-in /pulsar/manning/security/authentication/tls/${CLIENT_ID}.csr.pem \
-out /pulsar/manning/security/authentication/tls/${CLIENT_ID}.cert.pem \
-passin pass:${CA_PASSWORD}      #H

# Remove the client key and certificate request once we are finished
rm -f /pulsar/manning/security/authentication/tls/${CLIENT_ID}.csr.pem
rm -f /pulsar/manning/security/authentication/tls/${CLIENT_ID}.key.pem
}

# Create a certificate for Adam with admin role-level access
generate_client_cert admin admin admin-secret

# Create a certificate for the web app with webapp role-level access
generate_client_cert webapp-service webapp webapp-secret

# Create a certificate for Peggy who with payment-role level access
generate_client_cert peggy payments payment-secret

# Create a certificate for David who needs driver-role level access
generate_client_cert david driver davids-secret

#A Sets the local variable CLIENT_ID to the first parameter passed to the function call
#B Sets the local variable CLIENT_ROLE to the second parameter passed to the function call
#C Sets the local variable CLIENT_PASSWORD to the third parameter passed to the function call
#D Use the CLIENT_PASSWORD to secure the private key.
#E Pulsar uses the value associated with the common name (CN) to determine the client's role.
#F We need to pass in the password associated with the client key used to generate the CSR.
#G Specify that we want a client certificate
#H We need to provide the CA password in order to approve and sign the client's certificate request.

```

In a production environment, the clients would use their own private keys to generate the certificate requests and only send over the CSR files. Since I am automating the process, I have taken the liberty of generating these as part of the script for a small set of users, each with a different role.

At this point there are now several pairs of private keys and client certificates that have been generated and signed by our CA that can be used to authenticate to the Pulsar standalone instance. The properties shown in Listing 6.11 have also been added to the standalone.conf file in order to enable TLS-based authentication on the Broker.

Listing 6.11 TLS Authentication Property Changes to the standalone.conf file

```
#### To enable TLS authentication
authenticationEnabled=true      #A
authenticationProviders=org.apache.pulsar.broker.authentication.AuthenticationProviderTls
```

```
#B
```

```
#A Turn on authentication  
#B Specifies the TLS Authentication Provider
```

Changes are also required to the client.conf file as shown in Listing 6.12, which grants all of the pulsar command-line tools admin level permissions including the ability to define authorization rules at the Pulsar namespace level.

Listing 6.12 TLS Authentication Property Changes to the client.conf file

```
## TLS Authentication ##  
authPlugin=org.apache.pulsar.client.impl.auth.AuthenticationTls      #A  
authParams=tlsCertFile:/pulsar/manning/security/authentication/tls/admin.cert.pem,tlsKeyFil  
e:/pulsar/manning/security/authentication/tls/admin-pk8.pem      #B
```

```
#A Tells the client to use TLS authentication when connecting to Pulsar  
#B Specifies the client certificate and associated private key to use.
```

Let's rebuild the image to include the changes necessary to generate the TLS client certificates and enable TLS-based authentication that was discussed in this section by executing the commands shown in Listing 6.13.

Listing 6.13 Building the Docker Image from the Dockerfile

```
$ cd ${REPO_HOME}/pulsar-in-action/docker-images/pulsar-standalone      #A  
$ docker build . -t pia/pulsar-standalone-secure:latest      #B  
Sending build context to Docker daemon 3.345MB  
Step 1/8 : FROM apache/pulsar/pulsar-standalone:latest      #C  
--> 3ed9bffff717  
...      #D  
Step 8/8 : RUN ["/bin/bash", "-c", "/pulsar/manning/security/authentication/tls/gen-client-  
certs.sh"]      #E  
--> Running in 5beaabb5865  
Generating RSA private key, 2048 bit long modulus      #F  
.....+++++  
.....+++++  
e is 65537 (0x010001)  
Using configuration from /pulsar/manning/security/cert-authority/openssl.cnf  
Check that the request matches the signature  
Signature ok  
Certificate Details:  
    Serial Number: 4097 (0x1001)      #G  
    Validity  
        Not Before: Jan 13 02:47:54 2020 GMT  
        Not After : Apr 22 02:47:54 2020 GMT  
    Subject:  
        countryName          = US  
        stateOrProvinceName = CA  
        organizationName    = gottaeat.com  
        commonName          = admin      #H  
....  
Write out database with 1 new entries  
Data Base Updated  
Generating RSA private key, 2048 bit long modulus
```

```
...    #I  
Successfully built 6ef6eddd675e  
Successfully tagged pia/pulsar-standalone-secure:latest    #J
```

```
#A Change to the directory that contains the Dockerfile  
#B Command to build the Docker image from the Dockerfile and tag it  
#C Notice that there are now 8 steps instead of 7  
#D Execution of Steps 2 through 7.  
#E Executes the gen-client-certs.sh script  
#F The following stanza will be repeated 5 times, once for each client cert generated.  
#G The certificate serial number will be different for each client.  
#H The common name will be different for each client and is used to associate it to a particular role.  
#I Four more occurrences of the stanza should appear. One for each client certificate generated.  
#J Success message including the tag used to reference the image.
```

Next, I need to follow the steps shown in Listing 6.7 again, in order launch a new container based on the updated Docker image, ssh into the container, and run the `publish-credentials.sh` script inside the newly launched bash session in order to make these client certificates available to me in the `${HOME} /exchange` folder on my local machine. For instance, the certificate file `admin.cert.pem`, and the associated private key file `admin-pk8.pem` can now be used together to authenticate to the Pulsar standalone instance. Next, I will attempt to use TLS authentication by using the Java program shown in Listing 6.14, which is available in the GitHub repo associated with this book.

Listing 6.14 Authenticating with TLS Client Certificates

```
import org.apache.pulsar.client.api.*;  
  
public class TlsAuthClient {  
    public static void main(String[] args) throws PulsarClientException {  
        final String AUTH = "org.apache.pulsar.client.impl.auth.AuthenticationTls";  
        final String HOME = "/Users/david/exchange";  
        final String TOPIC = "persistent://public/default/test-topic";  
  
        PulsarClient client = PulsarClient.builder()  
            .serviceUrl("pulsar+ssl://localhost:6651/")      #A  
            .tlsTrustCertsFilePath(HOME + "/ca.cert.pem")      #B  
            .authentication(AUTH,      #C  
                "tlsCertFile:" + HOME + "/admin.cert.pem," +   #D  
                "tlsKeyFile:" + HOME + "/admin-pk8.pem")      #E  
            .build();  
  
        Producer<byte[]> producer =  
            client.newProducer().topic(TOPIC).create();  
  
        for (int idx = 0; idx < 100; idx++) {  
            producer.send("Hello TLS Auth".getBytes());  
        }  
        System.exit(0);  
    }  
}
```

#A Use the pulsar+ssl protocol, failure to do so will result in a connection failure.

```
#B The location of the file containing the trusted TLS certificates  
#C Use TLS Authentication  
#D The client certificate location  
#E The private key associated with the client certificate.
```

TLS Client Authentication is useful in cases where a server is keeping track of hundreds of thousands or millions of clients, as in IoT, or in a mobile app with millions of installs exchanging secure information. For example, a manufacturing company with 100s of thousands of IoT devices can issue a unique client certificate to each device, and then limit connections to only their devices by having Pulsar only accept connections where the client presents a valid client certificate signed by the company's certificate authority.

Or in the case of a mobile application, where you want to prevent your customers' sensitive data such as credit card information from getting stolen by someone spoofing your mobile app, you can issue a unique certificate to every app installation, and use them to validate that the request is coming from an approved version of your mobile application and not a spoofed version.

6.2.2 JSON Web Token Authentication

Pulsar supports authenticating clients using security tokens that are based on JSON Web Tokens (JWT), which is a standardized format used to create JSON-based access tokens that assert one or more claims. Claims are factual statements or assertions. The two claims that can be found in all JWT are: `iss` (issuer) that identifies the party that issued the JWT and `sub` that identifies the subject or party that the JWT carries information about.

In Pulsar, JSON Web Tokens are used to authenticate a Pulsar client and associate it with some "role" which will then be used to determine what actions it is authorized to perform such as publish or consume from a given topic. Typically, the administrator of the Pulsar cluster would generate a JWT that has a `sub` claim associated with a specific role, e.g., `admin`, that identifies the owner of the token. The admin would then provide the JWT to a client over a secure communication channel and the client could then use that token to authenticate with Pulsar and assigned to the admin role.

The JWT standard defines the structure of a JSON Web Token as consisting of the following three parts; the header which contains basic information, the payload which contains the claims, and the signature which can be used to validate the token. Data from each of these parts are encoded separately and concatenated together using periods to delineate the fields. The resulting strings looks something shown Figure 6.1 which can be transmitted easily over HTTP.



Figure 6.1 An encoded JSON Web Token and its corresponding parts.

Since JWTs are based on an open standard, the contents of a JWT are readable to anyone in possession of it. This means that anyone can attempt to gain access to your system with a JSON Web token that they generate themselves, or by altering an intercepted JWT and modifying one or more of the claims to impersonate a validated user. This is where the signature comes into play, as it provides a means to establish the authenticity of the entire JWT itself.

The signature is calculated by encoding the header and payload using Base64url Encoding and concatenating the two together along with a secret. That string is then passed through the cryptographic hashing algorithm specified in the header. With JWTs, there are two cryptographic schemes you can use; the first is referred to as a shared secret scheme in which both the party that generates the signature and the party that verifies it must know the secret. Since both parties are in possession of the secret, each party can verify the authenticity of a JWT by calculating the signature for themselves and validate that the value they computed matches the value in the JWT's signature section. Any discrepancy indicates that the token has been tampered with.

This scheme is useful when you want or need both parties to be able to exchange information back and forth in a secure manner. The second scheme uses an asymmetric public/private-key pair in which the party with the public key can validate the authenticity of any JWT it receives, and the party with the private key can generate valid JSON Web Tokens.

There are several online tools available that allow you to encode and decode JWTs, and Figure 6.2 shows the side-by-side output from one such tool that was used to decode two different JWTs. The output on the left is from a token that is using the shared secret scheme. As you can see, the contents of the token can be easily read by the tool without any additional credentials. However, the secret key is required in order to validate the token's signature to confirm it hasn't been tampered with.

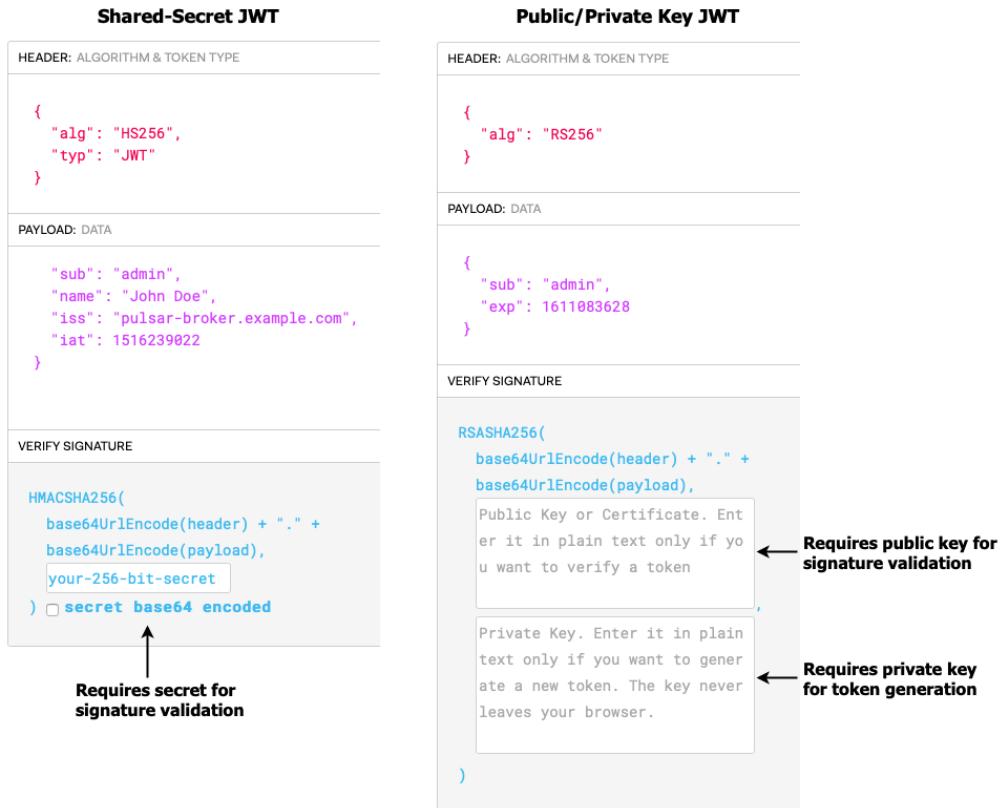


Figure 6.2: Verification Methods for Shared-Secret vs. Public/Private Keypair signed JWTs

The output on the right is from a JWT using the asymmetric scheme, whose contents can also be easily read without any additional credentials. However, the public key is required to verify the token's signature. The tool also requires a private key only if you wish to use it to generate a new JWT. Hopefully this helps clarify the differences between the two, and why both the JWT token and an associated key is required to configure Pulsar Brokers and clients to use JWT for authentication.

Lastly, it is worth mentioning that anyone in possession of a valid JWT can use it to authenticate with Pulsar. Therefore, adequate precautions should be taken to prevent an issued JWT from falling into unwanted hands. This includes only sending JWTs tokens over an TLS encrypted connection and storing the contents of a JWT in a safe location on the client machine.

In order to enable JWT authentication, I append a command to the Dockerfile from the previous section as shown in Listing 6.15 that executes another script named `gen-tokens.sh`, that generates JSON Web Tokens that can be used for authentication.

Listing 6.15 Updated Dockerfile Contents

```
FROM apachepulsar/pulsar-standalone:latest
ENV PULSAR_HOME=/pulsar

...      #A

RUN ["/bin/bash", "-c",
     "/pulsar/manning/security/authentication/jwt/gen-tokens.sh"]    #B
```

#A Same content as shown in Figure 6.9

#B Execute the specified script to generate some JWT tokens for role-based authentication.

Let's take a look at the `gen-token.sh` script in Listing 6.16, to see exactly what steps are required in order to generate the JSON Web Tokens that can be used to authenticate with the Pulsar broker.

Listing 6.16 Contents of the gen-token.sh File

```
#!/bin/bash

# Create a public/private keypair.
/pulsar/bin/pulsar tokens create-key-pair \    #A
  --output-private-key \
  /pulsar/manning/security/authentication/jwt/my-private.key \
  --output-public-key \
  /pulsar/manning/security/authentication/jwt/my-public.key

# Create the token for the admin role
/pulsar/bin/pulsar tokens create --expiry-time 1y \
  --private-key \    #B
  file:///pulsar/manning/security/authentication/jwt/my-private.key \
  --subject admin > /pulsar/manning/security/authentication/jwt/admin-token.txt

...      #C
```

#A We are going to use Asymmetric encryption, so we need to create a key pair.

#B Create a token and give it the subject of "admin"

#C Repeats token creation process for additional roles

After this script is executed, there will be several JWT tokens that have been generated and stored as text files on the Pulsar Broker. These tokens along with the public key will need to be distributed to the Pulsar clients so they can be used to authenticate to the Pulsar standalone instance. The properties shown in Listing 6.17 have also been modified inside the `standalone.conf` file in order to enable JWT-based authentication.

Listing 6.17 JWT Authentication Property Changes to the standalone.conf file

```
#### JWT Authentication #####
authenticationEnabled=true    #A
authorizationEnabled=true    #B
authenticationProviders=org.apache.pulsar.broker.authentication.AuthenticationProviderTls,o
  rg.apache.pulsar.broker.authentication.AuthenticationProviderToken    #C
tokenPublicKey=file:///pulsar/manning/security/authentication/jwt/my-public.key    #D
```

```
#A This is required for any authentication mechanism in Pulsar  
#B Enable authorization  
#C Append the JWT Token Provider to the list of Authentication Providers  
#D The location of the public key of the JWT keypair which is used for signature validation
```

If you want to use JWT authentication, then you will need to make changes shown in Listing 6.18 to the client.conf file. Since we have previously enabled TLS authentication for the client, we will need to comment out those properties for now since the client can only use one authentication method at a time.

Listing 6.18 JWT Authentication Property Changes to the client.conf file

```
##### JWT Authentication #####  
authPlugin=org.apache.pulsar.client.impl.auth.AuthenticationToken  
authParams=file:///pulsar/manning/security/authentication/jwt/admin-token.txt  
  
##### TLS Authentication #####  
#authPlugin=org.apache.pulsar.client.impl.auth.AuthenticationTls      #A  
#authParams=tlsCertFile:/pulsar/manning/security/authentication/tls/admin.cert.pem,tlsKeyFi  
le:/pulsar/manning/security/authentication/tls/admin-pk8.pem
```

#A Comment out both of the TLS authentication properties

Now let's rebuild the image to include the changes necessary to generate the JSON Web Tokens and enable JWT-based authentication that was discussed in this section by executing the commands shown in Listing 6.19. Be sure to have stopped and removed any previous running instances of the Pulsar image before doing so.

Listing 6.19 Building the Docker Image from the Dockerfile

```
$ cd ${REPO_HOME}/pulsar-in-action/docker-images/pulsar-standalone    #A  
$ docker build . -t pia/pulsar-standalone-secure:latest    #B  
Sending build context to Docker daemon 3.345MB  
Step 1/9 : FROM apachepulsar/pulsar-standalone:latest    #C  
--> 3ed9bffff717  
...  
Step 8/9 : RUN ["/bin/bash", "-c", "/pulsar/manning/security/authentication/jwt/gen-  
tokens.sh"]    #D  
Step 9/9 : CMD ["/pulsar/bin/pulsar", "standalone"]  
--> Using cache  
--> a229c8eed874  
Successfully built a229c8eed874  
Successfully tagged pia/pulsar-standalone-secure:latest    #E
```

#A Change to the directory that contains the Dockerfile
#B Command to build the Docker image from the Dockerfile and tag it
#C Notice that there are now 9 steps instead of 8
#D Executes the gen-token.sh script
#E Success message including the tag used to reference the image.

Next, we need to follow the steps shown in Listing 6.7 again, in order to launch a new container based on the updated Docker image, ssh into the container, and run the publish-credentials.sh script inside the newly launched bash session in order to make these JSON Web Tokens available to me in the \${HOME}/exchange folder on my local machine. Next, we

will attempt to use JWT authentication by using the Java program shown in Listing 6.20, which is available in the GitHub repo associated with this book.

Listing 6.20 Authenticating with JWT

```
import org.apache.pulsar.client.api.*;

public class JwtAuthClient {
    public static void main(String[] args) throws PulsarClientException {
        final String HOME = "/Users/david/exchange";
        final String TOPIC = "persistent://public/default/test-topic";

        PulsarClient client = PulsarClient.builder()
            .serviceUrl("pulsar+ssl://localhost:6651/")
            .tlsTrustCertsFilePath(HOME + "/ca.cert.pem")
            .authentication(
                AuthenticationFactory.token(() -> {
                    try {
                        return new String(Files.readAllBytes(
                            Paths.get(HOME + "/admin-token.txt")));
                    } catch (IOException e) {
                        return "";
                    }
                })).build();

        Producer<byte[]> producer =
            client.newProducer().topic(TOPIC).create();

        for (int idx = 0; idx < 100; idx++) {
            producer.send("Hello JWT Auth".getBytes());
        }
        System.exit(0);
    }
}
```

That concludes the configuration of Authentication on the Pulsar Broker. From this point forward, all Pulsar clients will be required to present valid credentials in order to be authenticated. Thus far we have only implemented two of the four different authentication mechanisms supported by Pulsar, TLS and JWT. Additional authentication methods can be enabled by following the steps outlined in Pulsar's online documentation.

6.3 Authorization

Authorization occurs only after a client has been successfully authenticated by Pulsar's configured authentication provider and determines whether or not you have sufficient permission to access a given topic. If you only enable authentication, any authenticated user has the ability to perform ANY action on ANY namespace or topic within the cluster.

6.3.1 Roles

In Pulsar, roles are used to define a collection of privileges, such as permission to access a particular set of topics, that are assigned to groups of users. Apache Pulsar uses the configured authentication providers to establish the identity of a client and then assign a *role*

token to that client. These role tokens are then used by the authorization mechanism to determine what actions the clients are allowed to do.

Role tokens are analogous to physical keys that are used to open a lock. Many people might have a copy of the key and the lock doesn't care who you are, only that you have the right key. Within Pulsar the hierarchy of roles falls into three categories, each with their own capabilities and their intended uses. As you can see from Figure 6.3 the role hierarchy closely mirrors Pulsar's cluster/tenant/namespace hierarchy when it comes to structuring data. Let us examine each of these roles in greater detail.

SUPER USERS

Just as the name implies, super users can perform any action at any level of the Pulsar cluster, however it is considered a best practice to delegate the administration of a tenant and its underlying namespaces to another person who will be responsible for the administration of that particular tenant's policies. Therefore, super users' primary activity is the creation of tenant admins, which can be accomplished from the Pulsar Admin CLI by executing a command similar to the one shown in Listing 6.21:

Listing 6.21 Creating a Pulsar Tenant

```
$PULSAR_HOME/bin/pulsar-admin tenants create tenant-name \#A
  --admin-roles tenant-admin-role \#B
  --allowed-clusters standalone    #C
```

#A Specifies the tenant's name

#B Specifies the tenant admin role name for the tenant

#C Specifies the clusters where the tenant will exist

This allows the super user to focus on cluster-wide administrative tasks such as cluster creation, broker management, and configuring tenant-level resource quotas to ensure all the tenants receive their fair share of the cluster's resources.

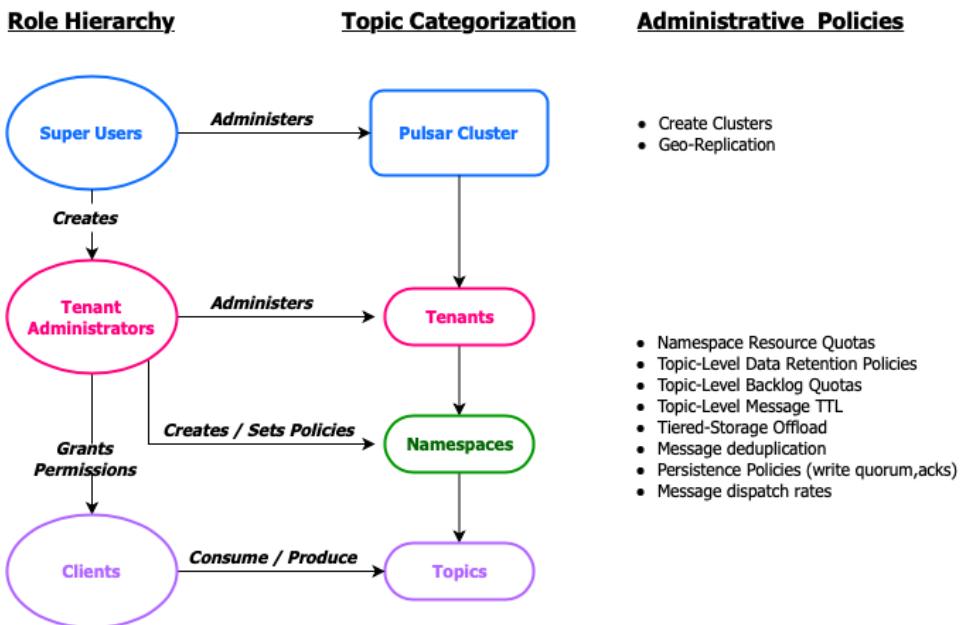


Figure 6.3: The Pulsar Role Hierarchy and Their Corresponding Administrative Tasks.

Typically, the administration of a tenant is given to a user that is responsible for designing the layout of the topics required for the tenant, such as a department head, project manager, or application team leader.

TENANT ADMINISTRATORS

The primary focus of the tenant administrator is the design and implementation of namespaces within their tenant. As I discussed earlier, namespaces are nothing more than a logical collection of topics that have the same policy requirements. Namespace-level policies apply to each topic within a given namespace, and allow you to configure things such as *data retention policies*, *backlog quotas*, *tiered-storage offload*, etc.

CLIENTS

Lastly are the client roles, which can be any application or program who has been granted permission to consume from or produce to topics with the Pulsar cluster. These permissions are granted by the tenant admins at either the namespace level, meaning that they apply to all the topics in a particular namespace, or on a per-topic basis. Listing 6.22 shows an example of both scenarios.

6.3.2 An Example Scenario

Let's imagine that you are working for a food delivery company that allows customers to order food from a variety of participating restaurants and have it delivered directly to you.

Rather than employ their own fleet of drivers, your company has decided to enlist private individuals to deliver the food in order to keep costs down.

The company will use three separate mobile applications to conduct business, a publicly available one for customers, a restricted one for all participating restaurateurs, and a restricted one for all authorized delivery drivers.

ORDER PLACEMENT USE CASE

Next, let's walk through a very basic use case for your company, the placement of and order by a customer all the way through to when it is assigned to a driver for delivery using a microservices based application that uses Apache Pulsar for communicating via messages. We will focus on how you might structure your namespaces under the "restaurateurs" and "driver" tenants along with what permissions you would want to grant. The "microservices" tenant will be used by multiple tenants, so it is created and managed by the application team. Figure 6.4 shows the overall message flow in the order entry use case.

In this use case a customer uses the company's mobile application to select some food items, enter the desired delivery address, and their payment information and submitting the request. The order information is published to the persistent://microservices/orders/new topic by the mobile application.

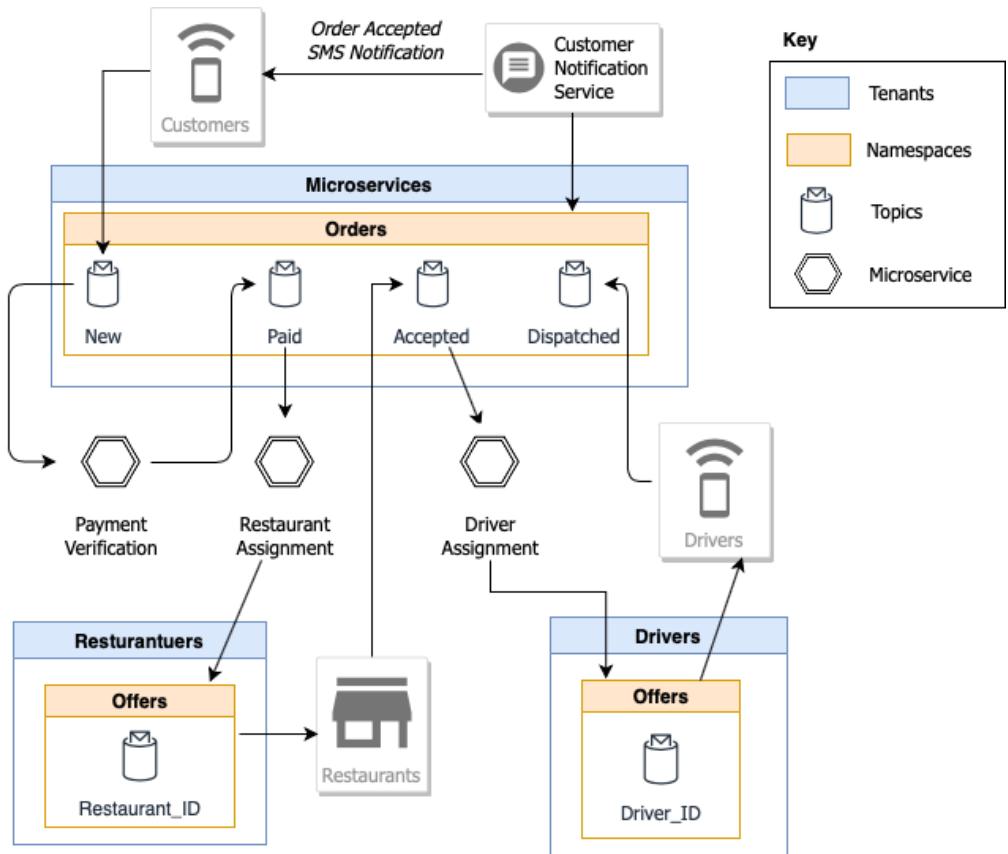


Figure 6.4 Message flow during the Order Placement Use Case

The Payment Verification microservice consumes the inbound order and communicates with the credit card payment system to secure payment for the items before placing the order in the persistent://microservices/orders/paid topic to indicate that payment has been captured for the items. The service also replaces the raw credit card information with an internal tracking id that be used at a later time to access the payment.

The Restaurant Assignment microservice is responsible for finding a restaurant that can fulfill the order by consuming from the persistent://microservices/orders/paid topic and determining a sub-set of candidate restaurants based on the food items and delivery location. It then sends out an offer to each of these restaurants by publishing the order details to a specific restaurant's topic that only they can access, e.g. persistent://restaurateurs/offers/restaurant_4827 to ensure that the order is only fulfilled by one restaurant. If the offer is rejected, or after a certain amount of time expires, the offer is rescinded and sent to the next restaurant in the list until it is fulfilled. Once a restaurant

accepts the order via their mobile application, a message containing the restaurant details, including location is published to the persistent://microservices/orders/accepted topic.

Once the order has been accepted, the Driver Assignment microservice will consume the message from the “accepted” topic and will attempt to locate a driver that can deliver the order to the customer. It selects a list of potential drivers based on their location, planned routes, etc. and publishes a message to a topic that only the driver can access, e.g. persistent://drivers/offers/driver_5063 to ensure that the order is only dispatched to a single driver. If the offer is rejected, or after a certain amount of time elapses, the offer is rescinded and sent to the next driver in the list until it is accepted. If a driver accepts the delivery, a message containing the driver details is published to the persistent://microservices/orders/dispatched topic to indicate that the order is scheduled for delivery.

All of the messages stored inside the topics of the persistent://microservices/orders namespace contain both a customer_id and order_id data element. A Customer Notification Service is subscribed to all of the topics shown and when it receives a new record in any of these topics, it parses out those fields and uses the information to send out a notification to the corresponding customer’s mobile app so they can track the progress of their order.

TENANT CREATION

As the Pulsar super user, your first task would be to create all three of the tenants and their associated tenant admin role. So, let’s review the steps required to create the necessary tenants for this use case that are shown in Listing 6.22

Listing 6.22 Creating the Pulsar Tenants

```
$PULSAR_HOME/bin/pulsar-admin tenants create microservices \
--admin-roles microservices-admin-role

$PULSAR_HOME/bin/pulsar-admin tenants create drivers \
--admin-roles drivers-admin-role

$PULSAR_HOME/bin/pulsar-admin tenants create restaurateurs \
--admin-roles restaurateurs -admin-role
```

Since we are delegating the administration of these tenants to someone else within our organization, I will take a moment to outline the characteristics of the individuals who should serve in this capacity on a tenant-by-tenant basis:

- Since the microservices tenant is used to share information across multiple services and applications, a good candidate for the corresponding tenant admin should be an IT professional responsible for architectural decisions across departments, such as an enterprise architect.
- For the restaurateurs’ tenant, a good candidate would be the person(s) in charge of the acquisition of new restaurateurs, as they will be the ones responsible for soliciting, vetting, and managing these partners. They can include the creation of restaurateurs’ private topics as part of the on-boarding process, etc.
- Similarly, a good candidate for administering the driver’s topic would be the person(s) in charge of the acquisition of new drivers.

AUTHORIZATION POLICIES

Based on the requirements for this use case, we would devise an overall security design similar to the one shown in Figure 6.5, which defines the following security requirements;

- The application that requires access
- The authentication method it will use
- The topics and namespaces it will access and how.
- The roleID we will assign to these permissions.

Application	Authentication	RoleID	Authorization Policies
 Driver Assignment Microservice	 JWT	DA_service	microservices/orders/accepted (consume) drivers/offers/driver_* (publish)
 Restaurant Assignment Microservice	 JWT	RA_service	microservices/orders/paid (consume) restaurants/offers/restaurant_* (publish)
 Payment Verification Microservice	 JWT	PV_service	microservices/orders/new (consume) microservices/orders/paid (produce)
 Customer Notification Microservice	 JWT	CN_service	microservices/orders/* (consume)
 Customer Mobile Application	 TLS Cert	Customer	microservices/orders/new (publish)
 Driver Mobile Application	 TLS Cert	Driver_<id>	drivers/offers/driver_ID (consume) microservices/orders/dispatched (produce)
 Restaurant Mobile Application	 TLS Cert	Restaurant_<id>	microservices/orders/accepted (produce) restaurants/offers/restaurant_ID (consume)

Figure 6.5: Application-level security requirements for the order placement use case.

For example, we know that the Driver Assignment microservice will use a JSON Web Token to authenticate with Pulsar and be assigned the “DA_service” role token which will grant consume permission on the persistent://microservices/orders/accepted topic and produce permission on all the topics in the persistent://drivers/offers namespace.

Therefore, the commands shown in Listing 6.23 would need to be executed in order to enable the security policies for the DA_service role, and similar commands would be required for the other services as well. It is worth mentioning that even though there may be multiple instances of the Driver Assignment Service running, they can all use the same RoleID. This is NOT the case for some of the mobile applications, since we will need to know exactly which user is accessing the system in order to enforce our security policies.

Listing 6.23 Enabling Security for the Driver Assignment Microservice

```
$PULSAR_HOME/bin/pulsar-admin namespaces create microservice/orders    #A  
$PULSAR_HOME/bin/pulsar tokens create \    #B  
  --private-key file:///path/to/private-secret.key \  
  --subject DA_service > DA_service-token.txt    #C  
$PULSAR_HOME/bin/pulsar-admin namespaces grant-permission \  
  drivers/offers --actions produce --role DA_service    #D  
$PULSAR_HOME/bin/pulsar-admin topics grant-permission \  
  persistent://microservices/orders/accepted \  
  --actions produce \  
  --role DA_service    #E
```

#A Create the microservices/order namespace, this only needs to be done once.

#B Create a JWT for the DA_service role.

#C Save the token in a text file to share with the service deployment team.

#D Grant the DA_service role permission to publish to any topic in the drivers/offers namespace

#E Grant the DA_service role permission to consume from a single topic.

The JWT should be saved to a file and shared with the team that will be deploying the Driver Assignment service, so that they can make it access to the service at runtime by either bundling it with the service or placing it in a secure location such as a Kubernetes secret. The mobile applications, in contrast, will use custom client certificates that are generated after the driver or restaurateur has been successfully on-boarded and issued a unique ID, as the client certificate that is generated will be associated with that specific ID in order to uniquely identify the user when they connect, e.g., Driver_4950 would be issued a client certificate associated with a common name of "driver_4950" and would be granted a role token for that role. We can see this in Listing 6.24 which shows the steps that would be taken by the driver-admin-role in order to add a new driver to the Pulsar cluster.

Listing 6.24 On-Boarding a New Driver

```
openssl req -config file:///path/to/openssl.cnf \  
  -key file:///path/to/driver-4950.key.pem -new -sha256    #A  
  -out file:///path/to/driver-4950.csr.pem \  
  -subj "/C=US/ST=CA/L=Palo Alto/O=gottaeat.com/CN=driver_4950"    #B  
  -passin pass:${CLIENT_PASSWORD}  
  
openssl ca -config file:///path/to/openssl.cnf \  
  -extensions usr_cert \  
  -days 100 -notext -md sha256 -batch \  
  -in file:///path/to/driver-4950.csr.pem    #C  
  -out file:///path/to/driver-4950.cert.pem    #D  
  -passin pass:${CA_PASSWORD}  
  
$PULSAR_HOME/bin/pulsar-admin topics grant-permission \  
  persistent://drivers/offers/driver_4950    #E  
  --actions consume \  
  --role driver_4950    #F  
  
$PULSAR_HOME/bin/pulsar-admin topics grant-permission \  
  persistent://microservices/orders/dispatched    #G  
  --actions produce \  
  --role driver_4950    #H
```

```
--role driver_4950 #H  
  
#A Use the driver-provided private key to generate a certificate request  
#B Use the Common Name (CN) field to specify the new roleID for this driver  
#C Use the CSR to generate a TLS client certificate for the new driver  
#D The name of the client certificate file, which will be bundled with the mobile app download.  
#E Grant the driver_4950 role permission to consume from a dedicated topic  
#F Grant the driver role permission to produce to the general "dispatched" topic.
```

In Listing 6.24, we see that a TLS client certificate is generated specifically for the new driver, based on their ID. This certificate is then bundled together with the Driver mobile application code to ensure that it is always used to authenticate with Pulsar when connecting to the cluster. The driver will then be sent a link they can use to download and install it on their smart phone.

6.4 Message Encryption

The new orders topic will contain sensitive credit card information that must be not be accessible to anyone other than the Payment Verification service. Even though we have configured TLS wire encryption to secure the information during transit into the Pulsar cluster, we also need to ensure that the data is stored in an encrypted format as well. Fortunately, Pulsar provides methods that allow you to encrypt message contents on the producer side before sending them to the Broker. These message contents will remain encrypted until a consumer with the correct private key consumes them.

In order to send and receive encrypted messages, you will first need to create an asymmetric key pair. A script named gen-rsa-key.sh is included in the Docker image we have been using that can be used to generate a public/private key pair. Listing 6.25 shows the contents of the script, which is located in the /pulsar/manning/security/message-encryption folder.

Listing 6.25 Contents of gen-rsa-keys.sh

```
# Generate the private key  
openssl ecparam -name secp521r1 \  
    -genkey \  
    -param_enc explicit \  
    -out /pulsar/manning/security/encryption/ecdsa_privkey.pem  
  
# Generate the public key  
openssl ec -pubout \  
    -outform pem \  
    -in /pulsar/manning/security/encryption/ecdsa_privkey.pem \  
    -out /pulsar/manning/security/encryption/ecdsa_pubkey.pem
```

You should give the public key to the producer application which in this particular case is the client mobile application. While the private key should only be shared with the Payment Verification service since it will need to consume the encrypted messages that are published to the persistent://microservices/orders/new topic. Listing 6.26 shows sample code of how to use the public key with a message producer to encrypt the data before it is sent.

Listing 6.26 Encrypted Producer Configuration

```
String pubKeyFile = "path to ecdsa_pubkey.pem";      #A
CryptoKeyReader cryptoReader
    = new RawFileKeyReader(pubKeyFile, null);      #B

Producer<String> producer = client
    .newProducer(Schema.STRING)
    .cryptoKeyReader(cryptoReader)      #C
    .cryptoFailureAction(ProducerCryptoFailureAction.FAIL)      #D
    .addEncryptionKey("new-order-key")      #E
    .topic("persistent://microservices/orders/new")
    .create();
```

#A The client must have access to the public key

#B Initialize the crypto reader to use the public key

#C Configure the producer to use the public key to encrypt the messages via the CryptoReader

#D Tells the producer to throw an exception if the message cannot be encrypted.

#E Provides a name for the encryption key.

Listing 6.27 shows sample code of how to configure the consumer inside the Payment Verification service to use the private key to decrypt the messages it consumes from the persistent://microservices/orders/new topic.

Listing 6.27 Configuring a Pulsar Client to Read from an Encrypted Topic

```
String privKeyFile = "path to ecdsa_privkey.pem";      #A
CryptoKeyReader cryptoReader
    = new RawFileKeyReader(null, privKeyFile);      #B

ConsumerBuilder<String> builder = client
    .newConsumer(Schema.STRING)
    .consumerName("payment-verification-service")
    .cryptoKeyReader(cryptoReader)      #C
    .cryptoFailureAction(ConsumerCryptoFailureAction.DISCARD)      #D
    .topic("persistent://microservices/orders/new")
    .subscriptionName("my-sub");
```

#A The client must have access to the public key

#B Initialize the crypto reader to use the public key

#C Configure the consumer to use the private key to decrypt the messages via the CryptoReader

#D Tells the consumer to discard the message if it cannot be decrypted.

Now that I've covered the steps required to configure the message producers and consumers to support message encryption. I want to drill down into the details of how it is implemented internally within Pulsar, and some of the key design decisions that you should be aware of in order to better understand how to leverage it in your applications. Figure 6.6 shows the steps taken on the producer side when message encryption is enabled. When the producers' send method is called with the raw message bytes, the producer first makes an internal call to the Pulsar client library to get the current symmetric AES encryption key. I say current, because these keys are automatically rotated every 4 hours in order to limit the impact of someone gaining unauthorized access to this key. By rotating the key used to encrypt the data, we limit the data exposed to a 4-hour window in the event of a compromised key rather than the entire history of the topic.

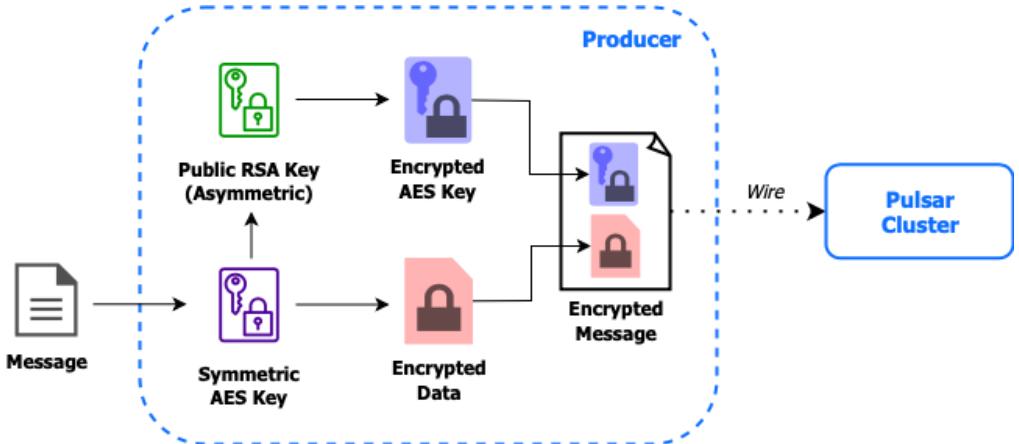


Figure 6.6: Message Encryption on the Pulsar Producer

The current AES key is then used to encrypt the raw message bytes of the message, which are then placed in the outbound message's payload. Next, the ***public key*** of the RSA asymmetric key pair we generated earlier is used to encrypt the AES key itself and the resulting encrypted AES Key is placed in the outbound message's header properties.

By encrypting the AES Key with the public key of an asymmetric key pair, we can rest assured that only a consumer with the corresponding private key of the key pair can decrypt (and thus read) the AES key that was used to encrypt the message payload. At this point all of the data within the outbound message is encrypted; the message payload with the AES Key, and the message header that contains the AES key that itself has been encrypted with a public RSA key. The message is then sent over the wire to the Pulsar cluster for delivery to its intended consumers.

Upon receipt of an encrypted message, a consumer performs the steps shown in Figure 6.7 in order to decrypt the message and access the raw data. First the consumer reads the message headers and looks for the "encryption key" property which contains the encrypted AES Key. It then uses the private RSA key from the asymmetric key pair to decrypt the contents of the "encryption key" property to produce the AES Key. Now that the consumer is in possession of the AES Key it can then decrypt the message contents with it due to the fact that it is a symmetrical key, meaning it is used to both encrypt and decrypt data.

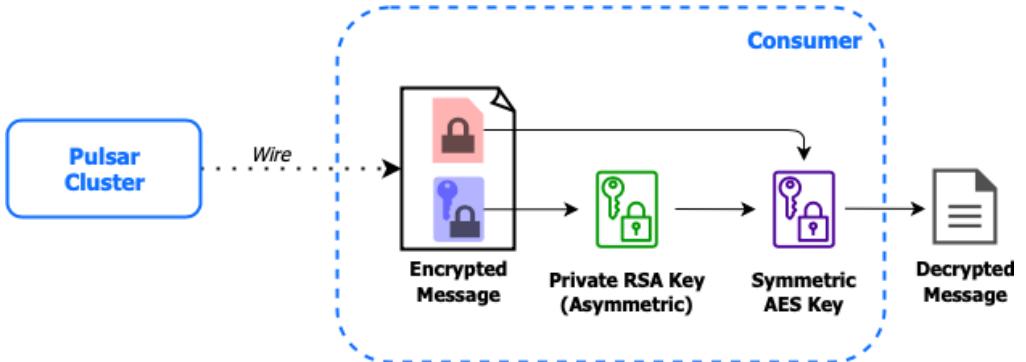


Figure 6.7: Message Decryption on the Pulsar Consumer.

By choosing to store the AES Key used to encrypt the message with the encrypted message contents themselves alleviates Pulsar from the responsibility of having to store and manage these encryption keys internally. However, if you lose or delete the RSA private key used to decrypt the RSA Key, your message is irretrievably lost and cannot be decrypted by Pulsar. Therefore, it is a best practice to store and manage the RSA key pairs in a third-party key management service, such as Amazon KMS.

Now that I brought up the possibility of a lost RSA Key, you might be wondering what exactly happens in such a scenario? Obviously, you never want this to occur, but it can happen, so you need to consider how you want your application to respond and what options you have on the both the producer and consumer side.

First let's consider the producer-side, there are really only two options here; you can choose to continue sending the messages without encrypting them. This might be a good option if the value of the data exceeds the risk of having it stored in an unencrypted format for a limited amount of time. For instance, in the order placement scenario we discussed earlier, it would be better to allow the customer orders to be sent than to shut down the entire business due to a missing RSA key. Since the data is being sent over a secure TLS connection, the risk is minimal.

The other option is for the producer to fail the messages and throw an exception, which is a good option when risk of the exposing the data outweighs the potential business value of having it delivered in a timely manner. This would typically be some sort of non-customer-facing application such as a backend payment processing system, etc. that does not have a strict business SLA. You can specify which action you want your producer to take by using the `setCryptoFailureAction()` method in the producer configuration as was shown in Listing 6.26.

If consumption fails due to decryption failure on the consumer side, then you have three options; first, you can elect to deliver the encrypted contents to the consuming application. However, it will then be the consuming applications responsibility to decrypt the message. This option is useful when your consumer's logic is NOT based on the message contents, such as routing of the message to downstream consumers based on the information in the message headers.

The second option is to fail the message which will make the Pulsar broker redeliver it. This option is useful if you are a shared subscription type so that there are multiple consumers on the same subscription. When you fail the message, you are hoping that the decryption issue is isolated to just the current consumer, such as a missing private key on that consumer's host machine, and if the message is redelivered to another consumer, it will be able to decrypt the message and consume it successfully. Do NOT use this option with an exclusive subscription type, as this will result in the message getting re-delivered indefinitely to the same consumer, who cannot process it.

The last option is to have the consumer discard the message and is useful for an exclusive subscription consumer that needs access to the message contents in order to perform its logic. As we mentioned earlier, failing the message will result in an infinite number of message redelivers and the consumer will be "stuck" re-processing the message. Such a scenario would halt message consumption on the subscription and steady increase in the message backlog for the entire topic. Therefore, discarding the message is the only way to prevent this from occurring, but at the cost of message loss.

6.5 Summary

- Pulsar supports TLS wire encryption which ensures that all data transferred between clients and the Pulsar Broker is encrypted.
- Pulsar supports TLS authentication with client certificates, which allows you to distribute these credentials to only trusted users and limit cluster access to the only those in possession of a valid client certificate.
- Pulsar allows you to use JSON Web tokens to authenticate users and map them to a specific role.
- Once authenticated, a user is granted a role token that is used to determine which resources within the Pulsar cluster that the user is authorized to read from and write to.
- Pulsar supports message-level encryption to provide security for data stored local disk in the bookies. This prevents authorized access to any sensitive data that may be inside those messages.

7

Schema Registry

This chapter covers

- Using the Pulsar schema to simplify your microservice development.
- Understanding the different schema compatibility types.
- Using the LocalRunner class to run and debug your functions inside your IDE.
- Evolving a schema without impacting existing consumers.

Traditional databases employ a process referred to as Schema-on-Write, where the table's columns, rows, and types are all defined before any data can be written into the table. This ensures that the data conforms to a predetermined specification and the consuming clients can access the schema information directly from the database itself which enables them to determine the basic structure of the records they are processing.

Apache Pulsar messages are stored as unstructured byte arrays and the structure is applied to this data only when it's read. This approach is referred to as Schema-on-Read and was first popularized by Hadoop and NoSQL databases. While the Schema-on-Read approach makes it easier to ingest and process new and dynamic data sources on the fly, it does have some drawbacks including the lack of a meta-store that clients can access to determine the schema for the Pulsar topic they are consuming from.

Pulsar clients just see a stream of individual records that can be of any type and need an efficient way to determine how to interpret each arriving record. This is where the Pulsar Schema Registry comes into play. It is a critical component of the Pulsar technology stack that tracks the schema of all the topics inside of Pulsar.

As we saw in the last chapter, the GottaEat development team has decided to embrace the microservice architectural style in which applications are comprised of a collection of loosely coupled, independently developed services. In such an architecture, different microservices will need to collaborate on the same data, and in order to do that they will need to know the basic structure of the event including the fields and their associated types,

etc. Otherwise, the event consumers will not be able to perform any meaningful calculations on the event data. In this chapter I will demonstrate how Pulsar's Schema Registry can be used to greatly simplify the sharing of this information across the application teams at GottaEat.

7.1 Microservice Communication

When you build microservices architectures, one of the concerns you need to address is that of interservice communication. There are different options for interservice communication, each with their own respective strengths and weaknesses. Typically, these various types of communication can be classified across two different decision factors. The first factor is whether the communication between the services is synchronous or asynchronous and the second factor is whether the communication is intended for a single receiver or multiple receivers as shown in Figure 7.1

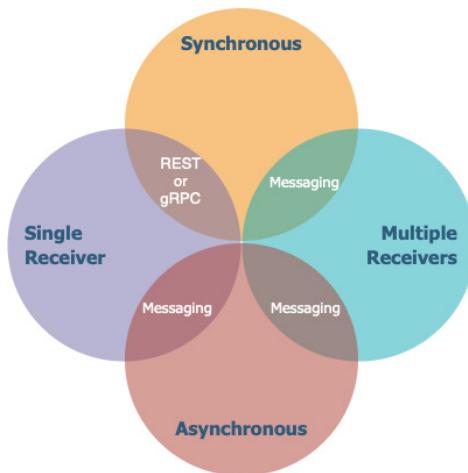


Figure 7.1: Microservice Communication Factors and Appropriate Communication Protocols

Protocols such as HTTP/REST or gRPC are most commonly used as a synchronous request/response-based interservice communication mechanism. With synchronous communication, one service sends a request and waits for a response from another service. The calling service's execution is blocked and cannot continue its task until it receives a response. This communication pattern is similar to procedure calls in traditional application programming in which a specific method is made to an external service with a specific set of parameters.

With asynchronous communication, one service sends a message and does not need to wait for a response. If you want an asynchronous publish/subscribe-based communication mechanism, messaging systems such as Pulsar are a perfect fit. A messaging system is also required to support publish/subscribe interservice communication between a single service

that has multiple message receivers. It is not practical to have a service block until all of the intended recipients have acknowledged the message as some of them may be offline, etc. A much better approach is to have the messaging system retain the messages.

These factors and communication mechanisms are good to know so you have clarity on the possible communication mechanisms you can use, but they're not the most important concerns when building microservices. What is important is being able to integrate your microservices while maintaining the independence of microservices, and in order to do so requires the establishment of a contract between the collaborating services regardless of the communication mechanism you chose.

7.1.1 Microservice APIs

As with any software development project, clear requirements help development teams create the right software. and having well-defined service contracts up-front allows microservice developers to write code without having to make assumptions about the data that will be provided to them or the expected output for any particular method within their service. In this section, I will cover how each of the communication styles supports service contracts.

REST AND gRPC PROTOCOLS

When evaluating the synchronous request/response-based interservice communication mechanisms, one of the biggest differences between REST and gRPC is the format of the payload. The conceptual model used by gRPC is to have services with clear interface definitions and structured messages for requests and responses. This model translates directly to programming language concepts like interfaces, functions, methods, and data structures. It also allows gRPC to automatically generate client libraries for you. These libraries can then be shared between the microservice development teams and act as a formal contract between them.

While the REST paradigm doesn't mandate any structure, message payloads typically use a loosely-typed data serialization system such as JSON. Consequently, REST APIs don't have a formal mechanism for specifying the format of the messages passed between services. The data is passed back and forth as raw bytes, which must be deserialized by the receiving service(s). However, these message payloads have an implied structure that the payload must adhere to. Therefore, any changes to the message payloads must be coordinated between the teams that are developing the services in order to ensure that any changes made by one team do not impact the others. One such example would be the removal of a field that is required by other services. Making such a change would violate the informal contract between the services and cause issues for the consuming services that depend on that particular field to perform their processing logic.

This problem isn't just limited to the REST protocol either, but rather is a side-effect of the loosely-typed data communication protocol being used. Consequently, this issue also exists with message-based interservice communication that uses a loosely-typed data serialization system.

MESSAGING PROTOCOL

As we have seen in Figure 7.1, most of the inter-service communication patterns can only be supported by a message-based communication protocol. Within Pulsar each message consists of two distinct parts; the message payload which is stored as raw bytes, and a collection of user defined properties that are stored as key/value pairs. Storing the message payload as raw bytes provided maximum flexibility but the trade-off is each message consumer is required to transform these bytes into a format that the consuming applications are expecting as shown in Figure 7.2.

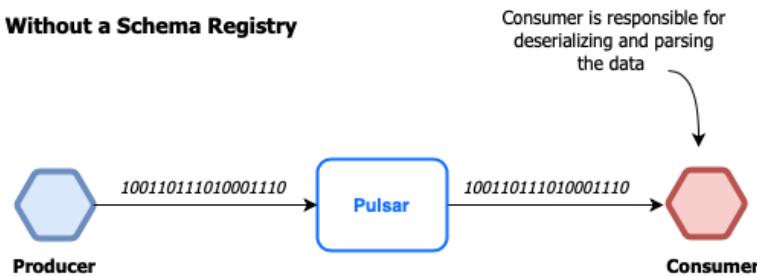


Figure 7.2: Pulsar takes raw bytes as input and delivers raw bytes as output.

Different microservices will need to communicate via messages, and in order to do so both the producer and consumer will need to agree upon the basic structure of the messages they are exchanging, including the fields and their associated types. The metadata that defines the structure of these messages is commonly referred to as message schemas. They provide a formal definition of how the raw message bytes should be translated into a more formal type structure, e.g., how the 0s and 1s stored inside the message payload map to a programming-language object type.

Message schemas are the closest we come to a formal contract between the services that generate the messages and the services that consume them. It is useful to think about message schemas as APIs. Applications depend on APIs and expect any changes made to APIs are still compatible and applications can still run.

7.1.2 The Need for a Schema Registry

A schema registry provides a central location for storing information about the schemas used within your organization which, in turn greatly simplifies the sharing of this information across application teams. It serves as a single source of truth for all the message schemas used across all your services and development teams, which makes it easier for them to collaborate. Having an external repository of message schemas helps answer the following questions for any given topic:

- If I am consuming messages, how do I parse and interpret the data?
- If I am producing messages, what is the expected format?
- Do all of the messages have the same format, or has the schema changed?

- If the schema has changed, what are the different message formats inside the topic?

Having a central schema registry along with the consistent use of schemas across the organization makes data consumption and discovery much easier. If you define a standard schema for a common business entity such as a customer, product, or order that almost all applications will use, then all message producing applications will be able to generate messages in the latest format. Similarly, consuming applications won't need to perform any transformations on the data in order to make it conform to a different format.

From a data discovery perspective, having the structure of the messages clearly defined in the schema registry allows data scientists to understand the structure of the data better without having to ask the development teams.

7.2 The Pulsar Schema Registry

The Pulsar Schema Registry enables message producers and consumers on Pulsar topics to coordinate on the structure of the topic's data through the Pulsar broker itself, without needing an additional serving layer for your metadata. Other messaging systems such as Kafka require a separate standalone schema registry component.

7.2.1 Architecture

By default, Pulsar uses the Apache BookKeeper table service for schema storage since it provides durable, replicated storage that ensures that the schema data is not lost. It also provides the added benefit of a convenient key/value API. Since Pulsar schemas are applied and enforced at the topic level, the topic name is used as the key and the values are represented by a data structure known as `SchemaInfo` that consists of the fields shown in Listing 7.1.

Listing 7.1 A Pulsar SchemaInfo Example

```
{
  "name": "my-namespace/my-topic",    #A
  "type": "STRING",      #B
  "schema": "",          #C
  "properties": {}       #D
}
```

#A A unique name, which should match the topic name that the schema is associated with.

#B Can either be one of the predefined schema types such as `STRING` or “struct” if you are using a generic serialization library such as [Apache Avro](#) or `JSON`

#C If you are using a supported serialization type such as Avro, etc., then this will contain the raw schema data.

#D A collection of user defined properties.

As you can see in Figure 7.3, Pulsar clients rely on the Schema Registry to get the schema associated with the topic they are connected to and invoke the associated serializer/de-serializer to transform the bytes into the appropriate type. This alleviates the consumer code from the responsibility of having to do the transformation and allows them to focus on the business logic.

With a Schema Registry

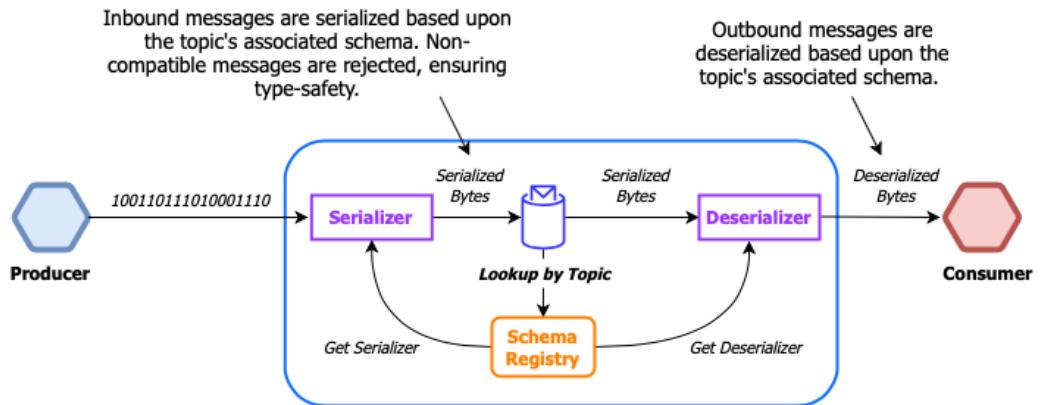


Figure 7.3: The Pulsar Schema Registry is used to serialize the bytes before they are published to a topic, and to deserialize them before they are delivered to the consumers.

The Schema registry uses the values of the *type* and *schema* fields to determine how to serialize/de-serialize the bytes contained inside the message body. The Pulsar Schema Registry supports a variety of schema types, which can be categorized as either *primitive* types or *complex* types. The *type* field is used to specify which of these categories the topic schema falls into.

PRIMITIVE TYPES

Currently, Pulsar provides several primitive schema types such as BOOLEAN, BYTES, FLOAT, STRING, and TIMESTAMP just to name a few. If the *type* field contains the name of one of these predefined schema types, then the message bytes will be automatically serialized/de-serialized into the corresponding programming language-specific type as shown in Table 7.1.

Table 7.1 Pulsar Primitive Types

BOOLEAN	A single binary value; 0 = false, 1 = true
INT8	A 8-bit signed Integer
INT16	A 16-bit signed Integer
INT32	A 32-bit signed Integer
INT64	A 64-bit signed Integer
FLOAT	A 32-bit, single precision floating point number (IEEE 754)
DOUBLE	A 64-bit, double precision floating point number (IEEE 754)
BYTES	A sequence of 8-bit unsigned bytes
STRING	A Unicode character sequence
TIMESTAMP	The number of milliseconds since Jan 1, 1970 stored as an INT64 value

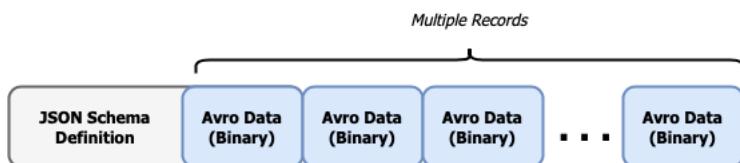
For primitive types, Pulsar does not require or use any data in the `schema` field because the schema is already implied, and thus is not necessary.

COMPLEX TYPES

When your messages require a more complex type, you should use one of generic serialization libraries supported by Pulsar such as Avro, JSON, or Protobuf. This would be denoted with an empty string in the `type` field of the `SchemaInfo` object associated with the topic, and the `schema` field will contain a UTF-8 encoded JSON string of the schema definition. Let's consider how this is applied to an Apache Avro schema definition to better illustrate how the Schema Registry simplifies the process.

Apache Avro is a data serialization format that supports the definition of complex data types via language-independent schema definitions in JSON. Avro data is serialized into a compact binary data format that can only be read by using the schema that was used when writing it. Since Avro requires that readers have access to the original writer schema in order to deserialize the binary data, the associated schema is typically stored with it at the beginning of the file. Avro was originally intended to be used for storing files with a large number of records of the same type, which allowed you to store the schema once and reuse it as you iterated through the records.

Traditional Avro Data File



Messages With Avro Data

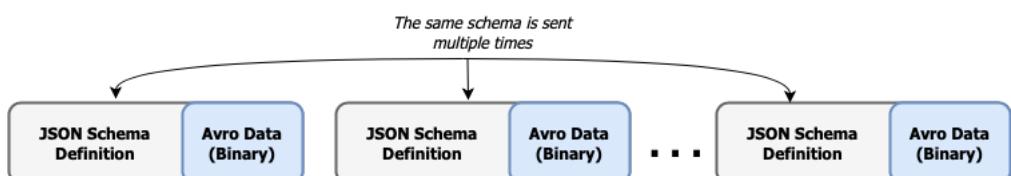


Figure 7.4: Avro messages would require the associated schema to be included with every message to ensure you could parse the binary message contents.

However, in a messaging use case, the schema would have to be sent with every single message as shown in Figure 7.4. Including the schema inside every Pulsar message would be inefficient in terms of memory, network bandwidth and disk space. This is where Pulsar's Schema Registry comes in, when you register a typed producer or consumer that uses an

Avro schema, the JSON representation of the Avro schema is stored inside the `schema` field of the associated `SchemaInfo` object. The Avro schema definition is then provided to the Avro serializers and deserializers used by the topic producers and consumers by the Schema Registry as shown in Figure 7.3. This eliminates the need to include it with every single message. Furthermore, the corresponding serializer or deserializer is cached inside the producers/consumers so it can be used on all subsequent messages until a different schema version is encountered.

7.2.2 Schema Versioning

Every `SchemaInfo` object stored with a topic has a version associated with it. When a producer publishes a message using a given `SchemaInfo`, the message is tagged with the associated schema version as shown in Figure 7.5. Storing the schema version with the message allows the consumer to use the version to look up the schema in the Schema Registry and use it to deserialize the message.

Messages With Avro Data and Schema Version

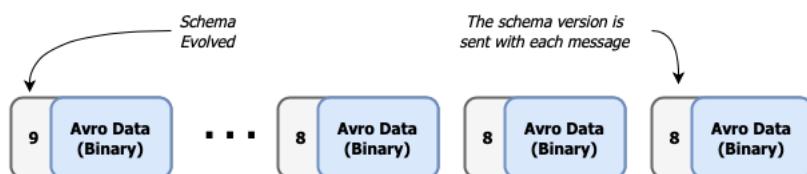


Figure 7.5: When using the Schema Registry, Avro messages would contain a schema version rather than the entire schema description. Consumers will use the schema version to retrieve the proper deserializer based on the version.

This is a simple and efficient way to associate the Avro schema with each Avro message without having to attach the schema's JSON description. The schemas are versioned in increasing order; e.g., v1, v2, ..., etc., and when the first typed consumer or producer with a schema connects to a topic, that schema is tagged as version 1. Once the initial schema is loaded, the brokers are provided the schema info for the topics they are serving and retain a copy of it locally for schema enforcement, etc.

Within a messaging system such as Pulsar, messages may be retained for an indefinite period of time. Consequently, some consumers will need to process these messages with older versions of the schema. Therefore, the Schema Registry retains a versioned history of all the schemas used in Pulsar in order to serve these consumers of historical messages.

7.2.3 Schema Compatibility

As you may recall, I started this chapter with a discussion of the importance of maintaining compatibility of messages used by microservices, even as requirements and applications evolve. In Pulsar every producer and consumer are free to use their own schema, so it is quite possible that a topic could contain messages that conform to different

schema versions as shown in Figure 7.5 where the topic contains messages with schema version 8 and messages with schema version 9.

It is important to point out that the Schema Registry does NOT ensure that every producer and consumer are using the exact same schema, but rather that the schemas they are using are compatible with one another. Consider the scenario where a development team changes the schema of the messages it is producing by adding/removing a field or changing one of the existing field types from say String to Timestamp. In order to maintain the implicit producer-consumer contract and avoid accidentally breaking consumer microservices, we need to ensure that the producers are publishing messages that contain all the information the consumers require. Otherwise, we run the risk of introducing messages that will “break” existing applications because you have removed a field that is required by these applications.

When you configure the schema registry to validate schema compatibility, then the Pulsar Schema Registry will perform a compatibility check when the producer connects to the topic. If the change will not “break” the consumers and cause exceptions, then the change is considered compatible and the producer is allowed to connect to the topic and produce messages with the new schema type. This approach helps prevent disparate development teams from introducing changes that “break” existing applications that are already consuming from Pulsar topics.

PRODUCER SCHEMA COMPATIBILITY VERIFICATION

Every time a typed producer connects to a topic as shown in Figure 7.6, it will transmit a copy of the schema it is using to the broker. A `SchemaInfo` object is created based on the passed in schema and passed to the Schema Registry. If the schema is already associated with the topic, then the producer is allowed to connect and can then proceed to publish messages using the specified schema.

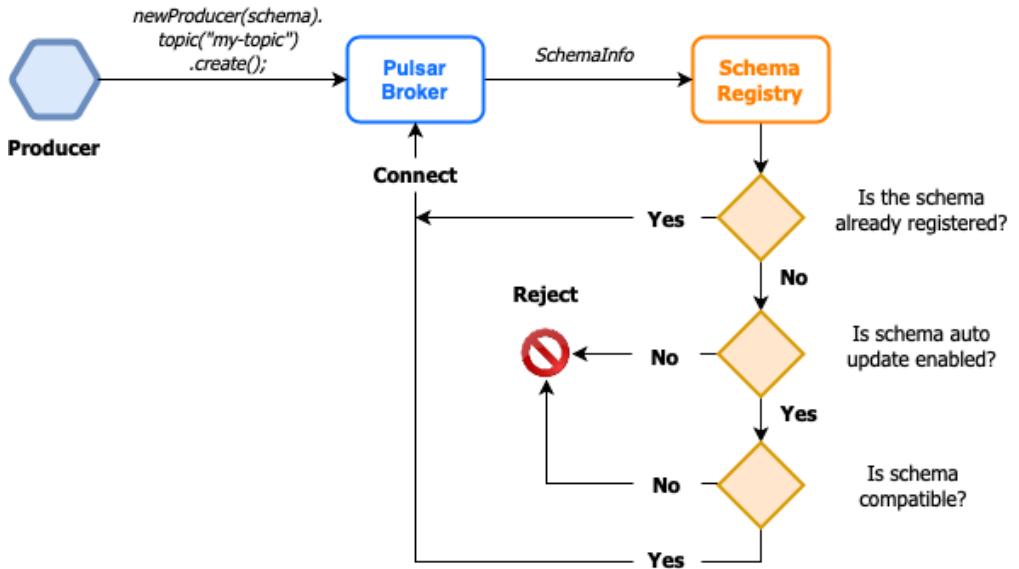


Figure 7.6: The logical flow of the schema validation checks for a typed producer.

If the schema is not already registered with the topic, then the Schema Registry checks the `AutoUpdate` strategy setting for the associated namespace to determine if the producer is permitted to register a new schema version on the topic or not. At this point the producer will be rejected if the policy prohibits the registration of new schema versions. If schema updates are permitted then the compatibility strategy check is performed, and if it passes, the schema is registered with the Schema Registry, and the producer is allowed to connect with the new schema version. If the schema is determined to be incompatible, then the producer is rejected.

CONSUMER SCHEMA COMPATIBILITY VERIFICATION

Every time a typed consumer connects to a topic as shown in Figure 7.7, it will transmit a copy of the schema it is using to the broker. A `SchemaInfo` object is created based on the passed in schema and passed to the Schema Registry.

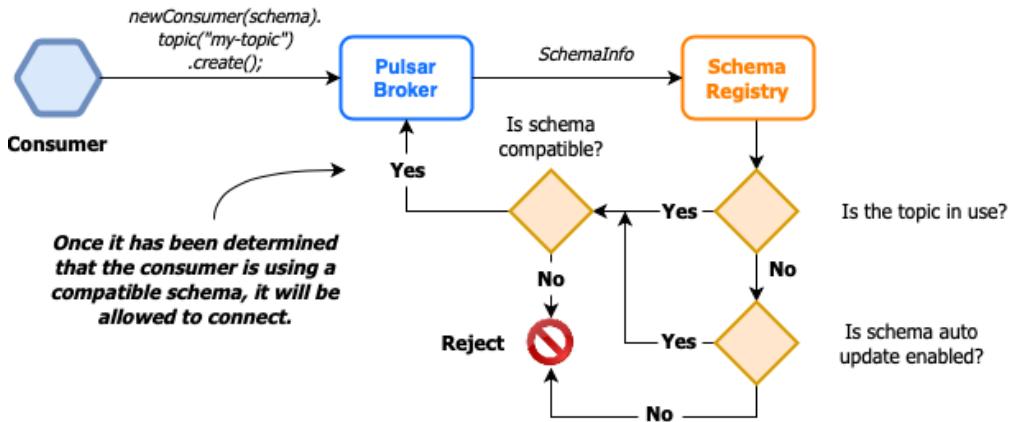


Figure 7.7: The logical flow of the schema validation checks for a typed consumer.

If the topic doesn't have any active producers or consumers, any registered schemas, or any existing data, then it is considered "not in use". In the absence of all of the aforementioned items, the Schema Registry checks the `AutoUpdate` strategy setting for the associated namespace to determine if the consumer is permitted to register a new schema version on the topic or not. The consumer will be rejected if it is prohibited from registering its schema, otherwise the compatibility strategy check is performed, and if it passes, the schema is registered with the Schema Registry, and the consumer is allowed to connect with the new schema version. If the schema is determined to be incompatible, then the consumer is rejected. The compatibility check is also performed if the topic is "in use".

7.2.4 Schema Compatibility Check Strategies

The Pulsar Schema Registry supports six different compatibility strategies which can be configured on a per topic basis. It is important to point out that all of the compatibility checks are from the consumer's perspective, even when it comes to the compatibility checks for producers as the goal is to prevent the introduction of messages that the existing consumers cannot process. The Schema Registry will use the underlying serialization library's (e.g., Avro) compatibility rules to determine whether the new schema is compatible with the current schema or not. The Pulsar Schema Registry's default compatibility type is `BACKWARD`, which is described in greater detail, along with the others, in the following sections.

BACKWARD COMPATIBILITY

Backward compatibility means that the consumer can use a newer version of the schema and still be able to read messages from a producer that is using the previous version. Consider the scenario where both the producer and consumer start out using the same version of the Avro schema, v1. One of the teams responsible for developing one of the consumers decide to add a new field called "status" to the schema in order to support a new customer loyalty

program that has various status tiers such as silver, gold, and platinum, as shown in Figure 7.8. In this case, the new schema version, v2 would be considered backwards compatible since it specifies a default value for the newly added field.

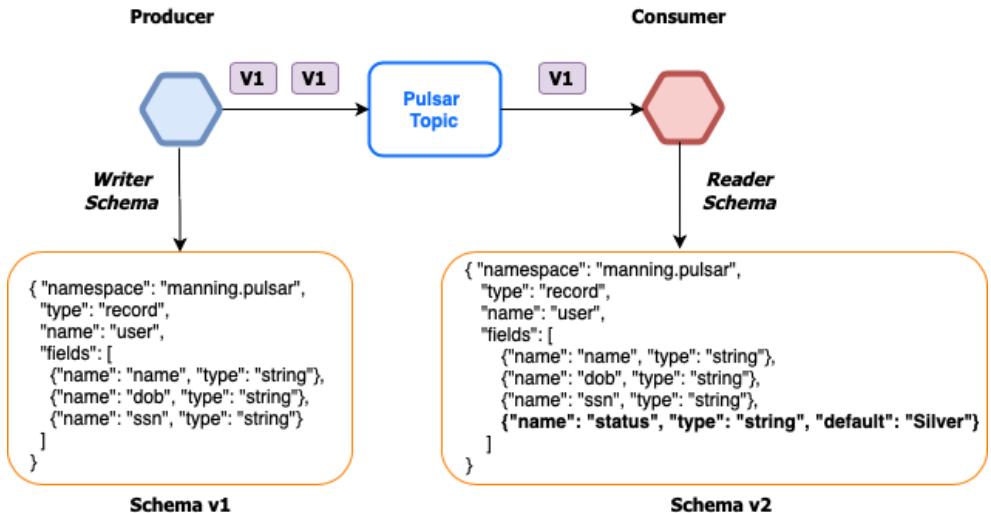


Figure 7.8: Backwards compatible changes are those that allow a consumer that is using a newer version of the schema to still be able to process messages that are written using the previous version of the schema.

This allows data written with v1 of the schema to be read by the consumer since the default value specified in the newer schema will be used for the missing field when deserializing the messages serialized with v1. Thereby treating all members as “silver” status in the consuming application.

To support this type of use case, you can use the BACKWARD schema compatibility strategy. However, that only supports the case where you have consumers that are one schema version ahead of your producers. If you want to support producers that are more than one schema version behind of your consumers, then you can use the BACKWARD_TRANSITIVE compatibility strategy instead.

Let’s expand upon the use case from the previous example, and now we have a new microservice added to the application that is responsible for determining a customer’s loyalty status and is producing messages that contain the “status” field.

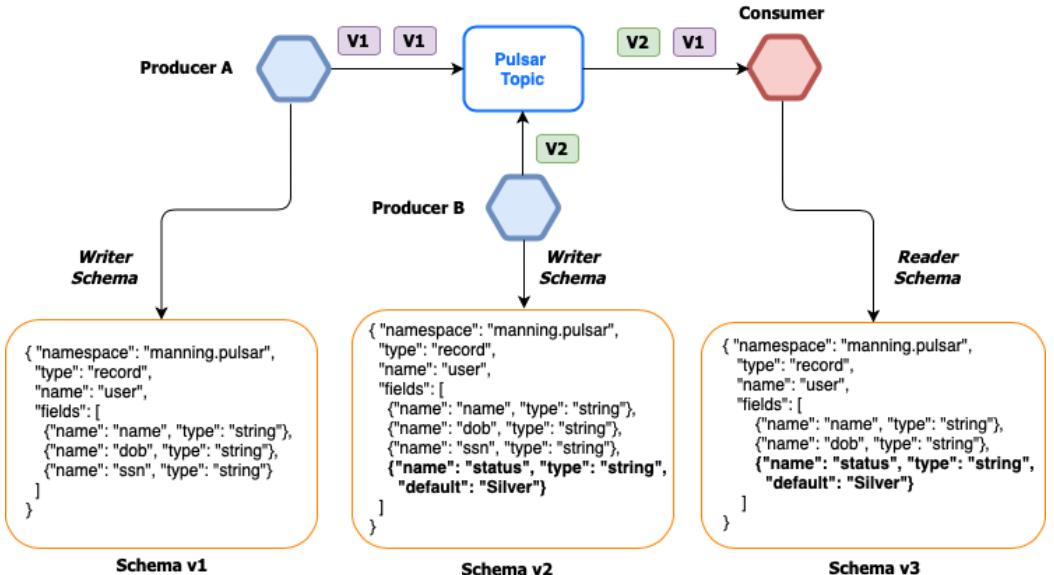


Figure 7.9: Backwards transitive compatible changes are those that allow a consumer that is using a newer version of the schema to still be able to process messages that are written using any of the previous version of the schema.

Also, the microservice that originally introduced the need for the “status” field has undergone a security review and it has been determined that having the customer’s social security number poses too big of a security risk, so it has been removed. As you can see from Figure 7.9, we still have the original producer that is using schema v1, the new microservice that is using v2 of the schema that includes the “status” field, and a consumer that is using a third schema version that has removed the “ssn” field.

In order to be considered backwards transitive compatible, the new consumer schema version v3, would have to be able to process messages from both active producers, e.g. schema versions v1 and v2.

Since v3 specifies a default value for the “status” field, data written with v1 of the schema can be read by the consumer since the default value specified in v3 of the schema will be used for the missing field when deserializing the messages serialized with v1. Similarly, since the messages serialized with v2 of the schema contain all of the fields required in v3, the consumer can simply ignore the extra “ssn” field in these messages.

FORWARD COMPATIBILITY

Forward compatibility means that the consumer can use an older version of the schema and still be able to read messages from a producer that is using a new version. Consider the scenario where both the producer and consumer start out using the same version of the Protobuf schema, v1. One of the teams responsible for developing one of the producers

decide to add a new field called “age” to the schema in order to support a new marketing program that is based on different age groups, as shown in Figure 7.10.

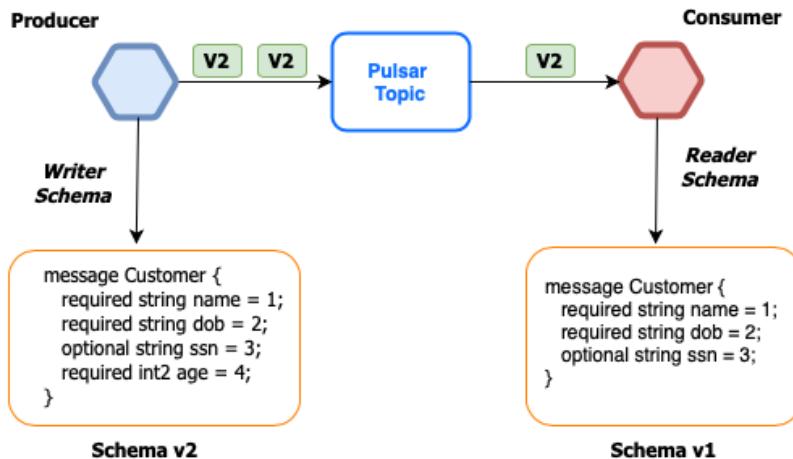


Figure 7.10: Forward compatible changes are those that allow a consumer that is using an older version of the schema to still be able to process messages that are written using a newer version of the schema.

In this case, the new schema version, v2 would be considered forward compatible since it simply added a new field that wasn't in the previous version. This allows data written with v2 of the schema to be read by the consumer since the newly added field will be ignored when deserializing the messages serialized with v2. The consumer can continue processing the messages since it does not have any dependency on the newly added “age” field.

To support this type of use case, you can use the FORWARD schema compatibility strategy. However, that only supports the case where you have consumers that are one schema version behind of the message producers. If you want to support producers that are more than one schema version ahead of your consumers, then you can use the FORWARD_TRANSITIVE compatibility strategy instead.

Let's expand upon the use case from the previous example, and now we have a new microservice added to the application that is responsible for determining a customer's demographic based on the “age” field and returns a message containing a brand-new field “demo” and eliminates the optional “ssn” field as shown in Figure 7.11

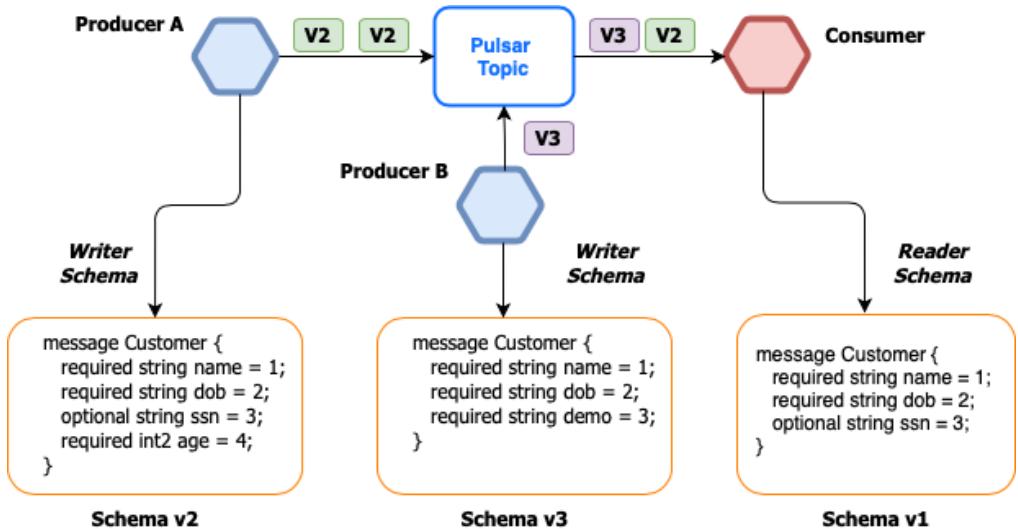


Figure 7.11: Forward transitive compatible changes are those that allow a consumer that is using an older version of the schema to still be able to process messages that are written using any of the newer versions of the schema.

In order to be considered forward transitive compatible, the consumer schema version v1, would have to be able to process messages from both active producers, e.g., schema versions v2 and v3.

Since v1 specifies that the “ssn” field is optional, data written with v3 of the schema can be read by the consumer since this field is not required, and the consumer must be prepared to handle null values for this field. Also, since the messages serialized with v2 and v3 both contain additional field that are not specified in v1, the consumer can simply ignore these extra fields in the messages as they are not required by the consumer.

FULL COMPATIBILITY

Full compatibility means the schema is both backward and forward compatible. Data serialized using an older version of the schema can be deserialized with the new schema, and data serialized using a newer version of the schema can also be deserialized with the previous version of the schema.

In some data formats, such as JSON, there are no full-compatible changes. Every modification is either only forward or only backward compatible. But in other data formats, like Avro or Protobuf, where you can define fields with default values, adding or removing a field with a default value is considered a fully compatible change. To support this type of use case, you can use either the FULL or FULL_TRANSITIVE schema compatibility strategies.

7.3 Using the Schema Registry

Let's consider a scenario in which you work for a food delivery company that allows customers to find and order food from any participating restaurant in your company's network and have it delivered to any location they choose. Orders can be placed via your company website or mobile application. Delivery of the food order is handled via a network of independent drivers who are required to download a different mobile application onto their smart phones. The drivers will receive notifications of food orders that are available for delivery within their area and have the option of accepting or declining the orders.

Once an order has been accepted by the driver it is added to their itinerary for the evening, and the driver will receive directions to the restaurant for pickup and the customer location for delivery inside the driver's mobile application. Participating restaurateurs are notified of incoming orders and responsible for reviewing the incoming orders and providing a time window for pickup. This information allows the system to better schedule the drivers in order to prevent them from arriving too early (and wasting their time) or too late (and the food is cold or late).

As the lead architect on this project, you have decided that a microservices architecture is best suited to meet the needs of the business. It also allows you to model the problem as an event-driven problem, which lends itself nicely to message-based communication between independent microservices. To this end you have sketched out the high-level design for the order entry use case shown in Figure 7.12 and need to determine how to implement this design using Pulsar. The overall flow of the use case is as follows:

1. Customers submit their order using the company website or mobile application, and they are published to the *customer order* topic.
2. There will be an order validation service that subscribes to the *customer order* topic and validates the order, including taking the provided payment information, such as the credit card number or gift card, and getting confirmation of payment.
3. Orders that are validated get published to the *validated order* topic and are consumed by both the customer notification service, e.g., sends an SMS message to the customer confirming the order was placed on the mobile app; and the restaurant notification service that publishes the order into the individual *restaurant order* topic associated with the order, e.g., `persistent://restaurants/orders/<restaurant-id>`.

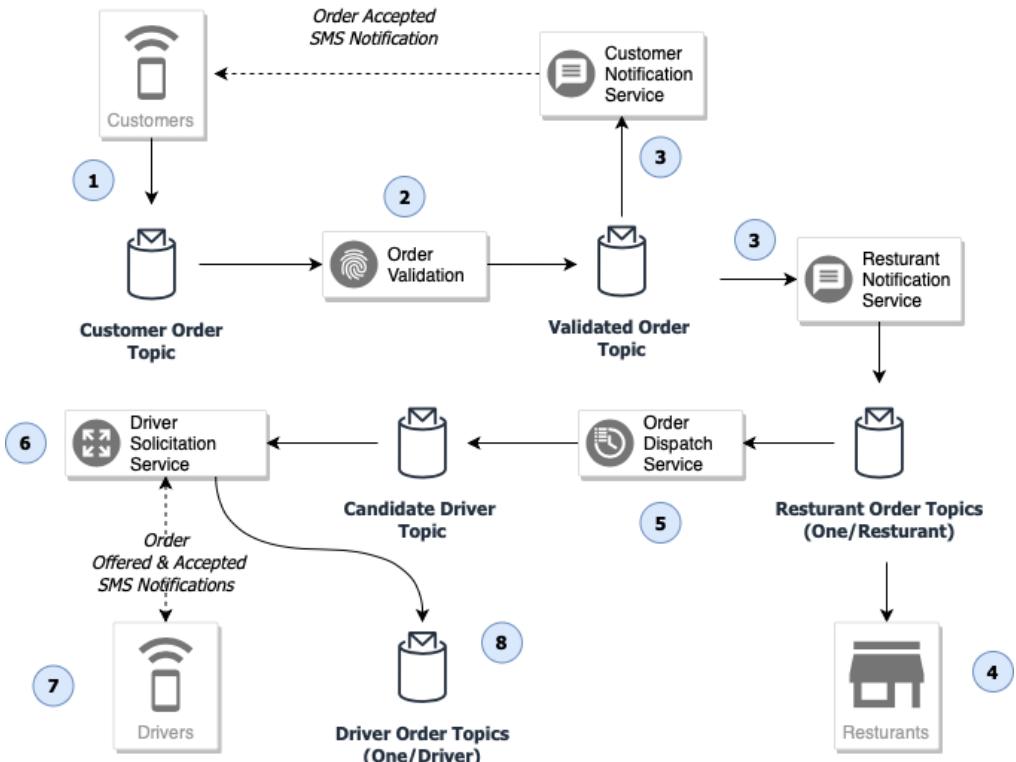


Figure 7.12 Order entry use case.

4. The restaurants review the incoming orders from their topic and updates the status of the order from "NEW" to "ACCEPTED" and also provide a pickup time window of when they feel the food will be ready.
5. The order dispatcher service is responsible for assigning the accepted orders to drivers and uses Pulsar's regex subscription capability to consume from ALL of the individual *restaurant order* topics, e.g., `persistent://restaurants/orders/*` and filters on a status of "ACCEPTED". It uses that information along with the driver's existing list of deliveries to select a handful of candidate drivers to offer the order to and publishes this list to the *candidate driver's* topic.
6. The driver solicitation service consumes from the *candidate driver's* topic and pushes a notification to each of the drivers in the list offering them the order. When one of the drivers accepts the order and notification is sent back to the solicitation service which in turn publishes the order to the driver's individual order topic, i.e., `persistent://drivers/orders/<drivers-id>`

Additional use cases are required to handle the routing of the driver, the notification of the customer about the order status, etc. But for now, I will focus on the order entry use case and how we Pulsar's Schema Registry will simplify the development of these microservices. Let's examine the structure of the GitHub project associated with this chapter of the book. For this section, ***please refer to the code in the 0.0.1 branch***. As you can see, it is a multi-module maven project that contains three submodules that I will discuss in the upcoming sections.

7.3.1 Modelling the Food Order Event in Avro

The first module, *domain-schema*, contains all of the Avro schema definitions for the GottaEat order entry use case. Avro schemas can be defined in either plain JSON or Avro IDL text files, but the schema files need to live somewhere, and this module serves that purpose.

As you can see from the contents of the *domain-schema* module, I have created an Avro IDL schema definition file named `src/main/resources/avro/order/food-order.avdl`, which contains the schema definition shown in Listing 7.1. This file represents the initial data model for the Food Order object that will be used across multiple services, and it will be used to generate the Java classes that will be used by all of the consuming Java-based microservices.

Listing 7.1 food-order.avdl

```
@namespace("com.gottaeat.domain.order")      #A
protocol OrderProtocol {
    import idl "../common/common.avdl";      #B
    import idl "../payment/payment-commons.avdl";
    import idl "../restaurant/restaurant.avdl";

    record FoodOrder {      #C
        long order_id;
        long customer_id;
        long restaurant_id;
        string time_placed;
        OrderStatus order_status;
        array<OrderDetail> details;      #D
        com.gottaeat.domain.common.Address delivery_location;      #E
        com.gottaeat.domain.payment.CreditCard payment_method;
        float total = 0.0;
    }

    record OrderDetail {
        int quantity;
        float total;
        com.gottaeat.domain.restaurant.MenuItem food_item;
    }

    enum OrderStatus {
        NEW, ACCEPTED, READY, DISPATCHED, DELIVERED
    }
}
```

#A The namespace for these types, which corresponds with the Java package name.

#B We import Avro type definitions from other files, which enables compositional schemas

```
#C The FoodOrder record definition
#D Each FoodOrder can contain one or more food items in it.
#E Using a type defined inside one of the included schema definitions.
```

We will use the Avro plugin to automatically generate Java classes based on the schema definitions in our project by adding the configuration shown in Listing 7.2 to the plugins section of the Maven pom.xml file.

Listing 7.2 Configuring the Avro Maven Plugin

```
<plugin>
  <groupId>org.apache.avro</groupId>
  <artifactId>avro-maven-plugin</artifactId>
  <version>1.9.1</version>
  <executions>
    <execution>
      <phase>generate-sources</phase>      #A
      <goals>
        <goal>idl-protocol</goal>      #B
      </goals>
      <configuration>
        <sourceDirectory>
          ${project.basedir}/src/main/resources/avro/order      #C
        </sourceDirectory>
        <outputDirectory>
          ${project.basedir}/src/main/java      #D
        </outputDirectory>
      </configuration>
    </execution>
  </executions>
</plugin>
```

```
#A We want to generate Java source files
#B The definitions are in the IDL format
#C Use the directory containing the food-order.avdl as the source directory
#D Where to output the generated source files
```

With the Avro schemas defined and the Maven plugin configured, we can execute the command shown in Listing 7.3 to generate the Java classes into the project's source folder as specified in Listing 7.2. This command will generate the Java classes source files, compile them, and jar them into the domain-schema-0.0.1.jar JAR file before finally publishing that JAR file to your local maven repository.

Listing 7.3 Generating the Java classes from the Avro schema

```
$ cd ./domain-schema
$ mvn install
[INFO] Scanning for projects...
[INFO] -----< com.gottaeat:domain-schema >-----
[INFO] Building domain-schema 0.0.1
[INFO] -----[ jar ]-----
[INFO]
[INFO] --- avro-maven-plugin:1.9.1:idl-protocol (default) @ domain-schema ---
[INFO]
[INFO] --- maven-compiler-plugin:3.8.1:compile (default-compile)
[INFO] Compiling 8 source files to domain-schema/target/classes
```

```
[INFO]
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ domain-schema ---
[INFO] Building jar: /domain-schema/target/domain-schema-0.0.1.jar
[INFO]
[INFO] --- maven-install-plugin:2.4:install (default-install)
[INFO] Installing /domain-schema/target/domain-schema-0.0.1.jar to ..
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
```

You will find the generated classes inside their respective sub-folders under the project's source folder as shown in Listing 7.4. These files are too lengthy to reproduce here, but if you open them with a text editor, you will see that the files contain POJOs that have been auto-generated by Avro and contain all the field definitions contained in the schema definitions along with methods to serialize the object to/from Avro's binary format.

Listing 7.4 Listing all of the generated Java classes

```
ls -R src/main/java/*
gottaeat

src/main/java/com/gottaeat:
domain

src/main/java/com/gottaeat/domain:
common    order          payment        restaurant

src/main/java/com/gottaeat/domain/common:
Address.java

src/main/java/com/gottaeat/domain/order:
FoodOrder.java  OrderDetail.java  OrderProtocol.java  OrderStatus.java

src/main/java/com/gottaeat/domain/payment:
CardType.java  CreditCard.java

src/main/java/com/gottaeat/domain/restaurant:
MenuItem.java
```

At this point I have a domain model in Java for our food order event that can be used by the other microservices in the project.

7.3.2 Producing Food Order Events

Rather than create a dependency on the customer mobile application that is being developed by a different team, we have decided to use this tool to generate load for testing purposes. The *customer-mobile-app-simulator* module contains an IO Connector intended to simulate the mobile application that customers will use to place food orders. The connector is defined inside the `CustomerSimulatorSource` class as shown in Listing 7.5.

Listing 7.5 The CustomerSimulatorSource IO Connector

```
import org.apache.pulsar.io.core.Source;
import org.apache.pulsar.io.core.SourceContext;
public class CustomerSimulatorSource implements Source<FoodOrder> {      #A

    private DataGenerator<FoodOrder> generator = new FoodOrderGenerator();      #B

    @Override
    public void close() throws Exception {
    }

    @Override
    public void open(Map<String, Object> map, SourceContext ctx)
        throws Exception {
    }

    @Override
    public Record<FoodOrder> read() throws Exception {
        Thread.sleep(500);      #C
        return new CustomerRecord<FoodOrder>(generator.generate());      #D
    }

    static private class CustomerRecord<V> implements Record<FoodOrder> {      #E
        ...
    }

    ...      #F
}
```

#A Implements the Source interface which defines the 3 overridden methods

#B The class that produces random food orders

#C Pause a half second between orders

#D Published a newly generated food order to the output topic

#E A wrapper class for sending the FoodOrder objects.

#F Where the LocalRunner code lives.

As you may recall from Chapter 5, the Source connector's read method is invoked by Pulsar's internal function framework repeatedly and the return value gets published to the configured output topic. In this case, the return value is a random food order based on the Avro schemas in the domain-schema module generated by another class inside the project named FoodOrderGenerator. I have decided to use the LocalRunner for debugging the CustomerSimulatorSource class as shown in Listing 7.6.

Listing 7.6 Using LocalRunner to debug the CustomerSimulatorSource IO Connector

```
...
public static void main(String[] args) throws Exception {
    SourceConfig sourceConfig = SourceConfig.builder()
        .className(CustomerSimulatorSource.class.getName())      #A
        .name("mobile-app-simulator")
        .topicName("persistent://orders/inbound/food-orders")      #B
        .schemaType("avro")
        .build();

    // Assumes you started docker container with --volume=${HOME}/exchange
```

```

String credentials_path = System.getProperty("user.home") +
    File.separator + "exchange" + File.separator;

LocalRunner localRunner = LocalRunner.builder()
    .brokerServiceUrl("pulsar+ssl://localhost:6651")      #C
    .clientAuthPlugin("org.apache.pulsar.client.impl.auth.AuthenticationTls")
    .clientAuthParams("tlsCertFile:" + credentials_path +
        "admin.cert.pem,tlsKeyFile:" + credentials_path + "admin-pk8.pem")
    .tlsTrustCertFilePath(credentials_path + "ca.cert.pem")   #D
    .sourceConfig(sourceConfig)
    .build();

localRunner.start(false);      #E
Thread.sleep(30 * 1000);
localRunner.stop();          #F
}

```

#A Specify the connector class that we want to run.
#B The topic that the connector will publish messages to.
#C Specify the URL of the Pulsar broker we are going to interact with.
#D Specify the TLS authentication credentials needs to connect to Pulsar
#E Starts the local runner
#F Stops the local runner

As you can see from Listing 7.6, I have configured the `LocalRunner` to connect to a locally running instance of the `pulsar-standalone-secure` Docker image that I created in Chapter 5. This is why there are several security related configuration settings in the source code that rely on the security credentials generated for that container such as the TLS client certificate and trust store.

7.3.3 Consuming the Food Order Events

Lastly, let's look at the `order-validation-service` module, which contains a single Pulsar function named `OrderValidationService`, that for the purposes of this chapter will be just a skeletal implementation of the microservice shown in Figure 7.12 that will accept the incoming food orders and validate them for correctness and confirmation of payment, etc. Over time additional logic will be added, but for now the function will simply write all of the food orders it receives to `stdout` before forwarding to the configured output topic.

Listing 7.7 The OrderValidationService Function

```

import org.apache.pulsar.functions.api.Context;
import org.apache.pulsar.functions.api.Function;
import com.gottaeat.domain.order.FoodOrder;

public class OrderValidationService implements Function<FoodOrder, FoodOrder> {      #A

    @Override
    public FoodOrder process(FoodOrder order, Context ctx) throws Exception {
        System.out.println(order.toString());
        return order;
    }
    ...
}

```

This class also includes a main method that contains a LocalRunner configuration so that we can debug this class locally as shown in Listing 7.8

Listing 7.8 The OrderValidationService LocalRunner Code

```
public static void main(String[] args) throws Exception {

    Map<String, ConsumerConfig> inputSpecs =
        new HashMap<String, ConsumerConfig> ();
    inputSpecs.put("persistent://orders/inbound/food-orders",
                  ConsumerConfig.builder().schemaType("avro").build());      #A

    FunctionConfig functionConfig =
        FunctionConfig.builder()
            .className(OrderValidationService.class.getName())      #B
            .inputs(Collections.singleton(
                "persistent://orders/inbound/food-orders"))      #C
            .inputSpecs(inputSpecs)
            .name("order-validation")
            .output("persistent://orders/inbound/valid-food-orders")   #D
            .outputSchemaType("avro")
            .runtime(FunctionConfig.Runtime.JAVA)
            .build();

    // Assumes you started docker container with --volume=${HOME}/exchange
    String credentials_path = System.getProperty("user.home") +
        File.separator + "exchange" + File.separator;

    LocalRunner localRunner = LocalRunner.builder()
        .brokerServiceUrl("pulsar+ssl://localhost:6651")      #E
        .clientAuthPlugin("org.apache.pulsar.client.impl.auth.AuthenticationTls")
        .clientAuthParams("tlsCertFile:" + credentials_path +
            "admin.cert.pem,tlsKeyFile:" + credentials_path + "admin-pk8.pem")
        .tlsTrustCertFilePath(credentials_path + "ca.cert.pem")      #F
        .functionConfig(functionConfig)
        .build();

    localRunner.start(false);      #G
    Thread.sleep(30 * 1000);
    localRunner.stop();          #H
}
```

#A We will be consuming Avro messages.

#B The function class that we want to run.

#C The topic that the function will consume messages from

#D The topic that the function will publish messages to.

#E Specify the URL of the Pulsar broker we are going to interact with.

#F Specify the TLS authentication credentials needs to connect to Pulsar

#G Starts the local runner

#H Stops the local runner

As you can see from Listing 7.8, I have configured the LocalRunner to connect to the same Pulsar instance. This is why there are several security related configuration settings in the source code that rely on the security credentials generated inside that container such as the TLS client certificate and trust store.

7.3.4 Complete Example

Now that we have walked through the code inside each of the Maven modules, it is time to walk through an end-to-end demonstration of the interaction between the `CustomerSimulatorSource` and the `OrderValidationService`. It is important to note that for this first demonstration both of these classes will be using the exact same version of the schema, e.g., the `domain-schema-0.0.1.jar`. Consequently, both the producer's and the consumer's schemas will be compatible with one another.

First, we need to have a running instance of the `pia/pulsar-standalone-secure` Docker image to serve as the Pulsar cluster we will use for testing. Therefore, you will need to execute the commands shown in Listing 7.9 in order to start an instance (if you don't already have one running), publish the security credentials needed by both of the `LocalRunner` instances, and create the topics that will be used.

Listing 7.9 Preparing the Pulsar Cluster

```
$ docker run -id --name pulsar --hostname pulsar.gottaeat.com -p:6651:6651 -p 8443:8443 -p 80:80 --volume=${HOME}/exchange:/pulsar/manning/dropbox -t pia/pulsar-standalone-secure:latest #A

$ docker exec -it pulsar bash #B

root@pulsar:/# /pulsar/manning/security/publish-credentials.sh #C
root@pulsar:/# /pulsar/bin/pulsar-admin tenants create orders #D
root@pulsar:/# /pulsar/bin/pulsar-admin namespaces create orders/inbound
root@pulsar:/# /pulsar/bin/pulsar-client consume -n 0 -s my-sub
    persistent://orders/inbound/food-orders #E
```

#A Launches the Pulsar standalone image and maps a volume for sharing the security credentials

#A Launches the F5 Big-IP Standalone Image and map
#B SSH into the Docker container you just launched

#C Publish all of the security credentials to the \${HOME}/exchange directory on your local machine.

#D Create the tenant and namespace we are going to use

#E Start a consumer on the specified topic

The last line in Listing 7.9 starts a consumer on the topic that we have configured the `CustomerSimulatorSource` to publish the `FoodOrder` events. This will automatically create the topic and allow us to confirm that the messages are getting published. Leave this command shell open so that we can monitor the messages as they come in and switch to your local IDE that you are using to review the code from the GitHub project associated with this chapter of the book. Navigate to the `customer-mobile-app-simulator` module and run the `CustomerSimulatorSource` as a Java application, which will execute the `LocalRunner` code shown in Listing 7.6.

If everything goes as expected, then you should start seeing Avro messages appear in the command shell window, as the messages are getting delivered to the consumer. When you look at the example of the expected output shown below, you will see that payloads contain a mix of readable text and binary data.

----- got message -----
◆?◆?◆?T◆?◆?◆?20200310◆AFrench Fries
Large@◆?@Fountain Drink
Small◆?709 W 18th StChicagoIL

```
66012&5555 6666 7777 8888  
66011123♦A
```

This is because the consumer we launched from the command line does not have a schema associated with it. Consequently, the raw bytes of the message which in this case are encoded in Avro's binary format are not deserialized before being delivered to the consumer and are treated as raw bytes instead. This should give insight into the actual content of the messages that are being transmitted from producer to consumer.

Next, let's switch back to your IDE, and navigate to the `order-validation-service` module and run the `OrderValidationService` as a Java application, which will execute the `LocalRunner` code shown in Listing 7.8 to start the consumer. You should see messages printed to stdout that contain food order data, but in JSON format now instead of the Avro binary data we were seeing in the schema-less consumer window.

```
{"order_id": 4, "customer_id": 471, "restaurant_id": 0, "time_placed": "2020-03-14T09:16:13.821", "order_status": "NEW", "details": [{"quantity": 10, "total": 69.899994, "food_item": {"item_id": 3, "item_name": "Fajita", "item_description": "Chicken", "price": 6.99}}], "delivery_location": {"street": "3422 Central Park Ave", "city": "Chicago", "state": "IL", "zip": "66013"}, "payment_method": {"card_type": "VISA", "account_number": "9999 0000 1111 2222", "billing_zip": "66013", "ccv": "555"}, "total": 69.899994}  
  
{"order_id": 5, "customer_id": 152, "restaurant_id": 1, "time_placed": "2020-03-14T09:16:14.327", "order_status": "NEW", "details": [{"quantity": 6, "total": 12.299999, "food_item": {"item_id": 1, "item_name": "Cheeseburger", "item_description": "Single", "price": 2.05}}, {"quantity": 8, "total": 31.6, "food_item": {"item_id": 2, "item_name": "Cheeseburger", "item_description": "Double", "price": 3.95}}], "delivery_location": {"street": "123 Main St", "city": "Chicago", "state": "IL", "zip": "66011"}, "payment_method": {"card_type": "VISA", "account_number": "9999 0000 1111 2222", "billing_zip": "66013", "ccv": "555"}, "total": 43.9}
```

This is because the function has a schema associated with it, which means that the Pulsar framework is automatically serializing the raw message bytes into the appropriate Java class based on the Avro schema definition.

One of the best features of Avro that makes it a great solution for message-based microservice communication, is its support for schema evolution. When a service that is writing messages updates its schema, the services consuming the messages can continue processing them without requiring any coding changes provided that the producer's new schema is compatible with the older version being used by the consumers.

Currently, both our producer and consumer are using the same version of the domain-schema JAR file, i.e. 0.0.1 and thus are also using the exact same schema version. While this is the expected behavior, it does not effectively demonstrate Pulsar's schema evolution capabilities. I will demonstrate this capability in the next section by walking through the project code on the 0.0.2 branch of the associated GitHub project. Therefore, you will need to switch to the 0.0.2 branch before walking through the examples and most importantly leave your Docker container running as it is.

7.4 Evolving the Schema

During our weekly meeting with the customer mobile application team, we are informed that their initial testing had revealed a gap in their requirements. Specifically, the current food order schema does not support the ability for customers to customize their food orders to their particular tastes. Currently, a customer cannot specify that they want “no onions” on their hamburgers or “extra guacamole” on their burritos. Consequently, the schema will have to be revised, and an additional field named *customizations* will be added to the original “menu item” type as shown in Listing 7.10.

Listing 7.10 Evolving the Schema

```
@namespace("com.gottaeat.domain.restaurant")  
  
protocol RestaurantsProtocol {  
  
    record MenuItem {  
        long item_id;  
        string item_name;  
        string item_description;  
        array<string> customizations = [""];      #A  
        float price;  
    }  
}
```

#A The newly added field to support customizations of individual food items with a default value

After making this change to the schema inside the domain-schema module, you should also update the artifact version in the pom.xml file to 0.0.2 so that we can differentiate between the two versions. Once these changes have been made to the source code, you should execute the command shown in Listing 7.11 to generate the Java classes source files, compile them, and jar them into the domain-schema-0.0.2.jar JAR file.

Listing 7.11 Generating the Java classes from the updated Avro schema

```
$ cd ./domain-schema  
$ mvn install
```

Make sure that the version being built is 0.0.2. If you fail to update the version number in the pom.xml file and leave it as 0.0.1, you will overwrite the existing jar that has the old schema version in it and your results will differ from the ones shown. If you happen to inadvertently overwrite the domain-schema-0.0.1.jar you can remove the newly added field and rebuild the jar file. Then add the field back, change the version number to 0.0.2 and rebuild it again. You can easily verify that the Java classes you generated were based on the new schema version by looking for the *customizations* field in the src/main/java/com/gottaeat/domain/restaurant/MenuItem.java class.

Next, I will update the version number of the domain-schema dependency in the *customer-mobile-app-simulator* module to use the updated schema as shown in Listing 7.12.

Listing 7.12 Update the version of the domain-schema dependency

```
<dependency>
  <groupId>com.gottaeat</groupId>
  <artifactId>domain-schema</artifactId>
  <version>0.0.2</version>
</dependency>
```

The FoodGenerator was updated in the 0.0.2 branch to include customizations to the food orders so it will require the newer version of the jar that was built in Listing 7.11. If you experienced any compile errors, then you most likely were still referencing the older version. You can now refresh the maven dependencies to ensure that you are using the 0.0.2 version of the domain-schema jar and run the `CustomerSimulatorSource`'s LocalRunner again. The updated logic inside the FoodGenerator always adds a customization to every fountain drink to specify which type it is, e.g., Coca-Cola, Sprite, etc. It also randomly adds some customizations to the other food items.

```
----- got message -----
n◆.2020-03-14T15:10:16.773◆@Fountain Drink
SmallCoca-Cola◆?123 Main StChicagoIL      #A
66011&1234 5678 9012 3456
66011000◆@_
----- got message -----
p◆.2020-03-14T15:10:17.273◆@Fountain Drink
Large
    Sprite@◆◆@BurritBeefSour Cream◆◆@◆_A      #B
                                         FajitaChickenExtra Cheese◆◆@ 844 W Cermak
    RdChicagoIL
66014&1234 5678 9012 3456
66011000◆◆◆A
```

#A Fountain drink customization of Coca-Cola
#B Food item customization of Sour Cream

If you observe the schema-less consumer console window again and you will see an occasional record with some customizations as shown above. This indicates that we are producing messages based on the updated schema version 0.0.2, which is what we expected. Lastly, I will now run the `OrderValidationService` LocalRunner inside the `order-validation-service` module, which is still configured to use the 0.0.1 version of the domain-schema jar that contains the older version of the schema.

Since the 0.0.2 version of the schema is backwards-compatible with the 0.0.1 version used by the `OrderValidationService`, it will be able to consume the messages based on the newer schema version. As you can see from the deserialized Avro messages shown below since these newer messages are being deserialized with the older schema, the newly added `customizations` field is ignored. This is as expected and does not impact the functionality of the consumer whatsoever since they were never aware of these fields to begin with.

```
{"order_id": 55, "customer_id": 73, "restaurant_id": 0, "time_placed": "2020-03-
14T15:10:16.773", "order_status": "NEW", "details": [{"quantity": 7, "total": 7.0,
"food_item": {"item_id": 10, "item_name": "Fountain Drink", "item_description": "
```

```

    "Small", "price": 1.0}]], "delivery_location": {"street": "123 Main St", "city": "Chicago", "state": "IL", "zip": "66011"}, "payment_method": {"card_type": "AMEX", "account_number": "1234 5678 9012 3456", "billing_zip": "66011", "ccv": "000"}, "total": 7.0}

{"order_id": 56, "customer_id": 168, "resturant_id": 0, "time_placed": "2020-03-14T15:10:17.273", "order_status": "NEW", "details": [{"quantity": 2, "total": 4.0, "food_item": {"item_id": 11, "item_name": "Fountain Drink", "item_description": "Large", "price": 2.0}}, {"quantity": 1, "total": 7.99, "food_item": {"item_id": 1, "item_name": "Burrito", "item_description": "Beef", "price": 7.99}}, {"quantity": 2, "total": 13.98, "food_item": {"item_id": 3, "item_name": "Fajita", "item_description": "Chicken", "price": 6.99}}], "delivery_location": {"street": "844 W Cermak Rd", "city": "Chicago", "state": "IL", "zip": "66014"}, "payment_method": {"card_type": "AMEX", "account_number": "1234 5678 9012 3456", "billing_zip": "66011", "ccv": "000"}, "total": 25.97}

```

It is also worth pointing out that no code changes were required to the OrderValidationService. Therefore, had this been a production environment the currently running instance of the service could remain running without disruption even though the mobile application had made changes to its code base. Making them completely decoupled from one another even when an API change (message format) is made.

7.5 Summary

- The different microservice communication styles and why Pulsar is a perfect fit for asynchronous publish/subscribe-based inter-service communication.
- The Pulsar Schema Registry enables message producers and consumers to coordinate on the structure of the data at the topic level and enforces schema compatibility for message producers.
- The Pulsar Schema Registry supports eight different compatibility strategies including forward, backward, and full, and that each of the compatibility checks are from the consumer's perspective.
- The Avro's interface definition language (IDL) is a great way to model events consumed within Pulsar because it allows you to modularize your types and share them across services easily.
- The Pulsar Schema Registry can be configured to enforce forward and/or backward schema compatibility for a Pulsar Topic, by ensuring that the connecting producer or consumer are using a schema that is compatible with all existing clients.

8

Pulsar Function Patterns

This chapter covers

- Designing an application based on Pulsar Functions.
- Implementing well-established messaging patterns using Pulsar Functions.

In the previous chapter I introduced a hypothetical food delivery service named GottaEat and outlined the basic order entry use case in which customers place orders with the company's mobile application. As you may recall, the first microservice in that process was the `OrderValidationService` which is responsible for ensuring that the order is valid before forwarding the order to the drivers for delivery if it was valid or notifying the customer of any errors with the order.

However, the term validate is a bit more complicated than merely ensuring all of the fields are of the proper type and format. In this particular scenario an order is only considered valid if the method of payment provided by the customer is approved and the funds from the bank are authorized, and there is at least one restaurant open and willing to provide all of the requested food items, and most importantly if the delivery address provided by the customer can be resolved to both a latitude-longitude pair and a street address. If we are unable to confirm all of these, then the order is considered invalid and the customer must be notified accordingly. Consequently, the `OrderValidationService` is not a simple microservice that can make all of these decisions on its own, but instead must coordinate with other systems and is therefore a good example of how a Pulsar Application can be composed of several smaller functions and services.

The `OrderValidationService` must integrate with several other microservices and external systems in order to perform the payment processing, geo-encoding, and food order placement required to fully validate an order. Therefore, it is best to look for existing solutions to these types of challenges rather than reinvent the wheel, and the catalog of patterns contained within Gregor Hohpe's [Enterprise Integration Patterns](#) book serves as a great reference in this regard. It contains several technology-agnostic, time-tested patterns

to solve common integration challenges. These patterns are categorized according to the type of problem they address and are applicable to most message-based integration platforms. In the next sections, I will demonstrate how these patterns would be implemented using Pulsar Functions.

8.1 Data Pipelines

In order to effectively design your Pulsar Functions based applications, you will want to familiarize yourself with the concepts of DataFlow programming and Data Pipelines. I will describe these programming models at a high level and point out how Pulsar Functions are a natural fit for this programming style.

8.1.1 Procedural Programming

Traditionally, computer programs were modeled as a series of sequential operations where each operation depended upon the output of the previous operation. These programs cannot be executed in parallel because they operated on the same data and therefore had to wait for the previous operation to complete before executing the next. Consider the logic for a basic order entry application in this programming model. You would write a simple function called “`processOrder`” that would perform the following sequence of steps (either directly or indirectly via a call to another function) to complete the process and return an order number to indicate success;

1. Check the inventory for the given item to make sure it is in stock.
2. Retrieve customer information (shipping address, payment method, coupons, loyalty).
3. Calculate the price including sales tax, shipping, coupons, loyalty discounts, etc.
4. Collect the payment from the customer.
5. Decrease the item count in the inventory.
6. Notify the fulfillment center of the order so it can be processed and shipped.
7. Return the order number to the customer.

Each of these steps acts upon the same order and depends upon the output from the previous step; i.e., you cannot collect payment from the customer before you have calculated the price, etc. Thus, each step has to wait for the previous step to complete before proceeding, making it impossible to perform any of these steps in parallel.

8.1.2 DataFlow Programming

In contrast, dataflow programming focuses on the movement of data through a series of independent data processing functions that are connected via explicitly defined inputs and outputs. These pre-defined sequences of operations are commonly referred to as data pipelines and are what your Pulsar Function applications should be modelled as. Here the focus is one moving the data through a series of stages, each of which processes the data and produces a new output. Each processing stage in a data pipeline should be able to

its processing based solely on the content of the incoming message. This eliminates any processing dependencies and allows each function to execute as soon as data is available.

A common analogy for a data pipeline is an assembly line in an automobile factory. Rather than assembling a car in one location piece by piece, each car passes through a series of stages to construct the vehicle. A different piece of the car is added at each stage but can be done in parallel rather than sequentially. Consequently, multiple cars can be assembled in parallel, effectively increasing the throughput of the factory.

Pulsar Function based applications should be designed as topologies consisting of several individual Pulsar Functions that perform the data processing operations and are connected together via Pulsar input and output topics. These topologies can be thought of as directed-acyclic-graphs (DAGs) with the functions/microservices acting as the processing units and the edges representing the input/out topic pairings used to direct data from one function to another as shown in Figure 8.1.

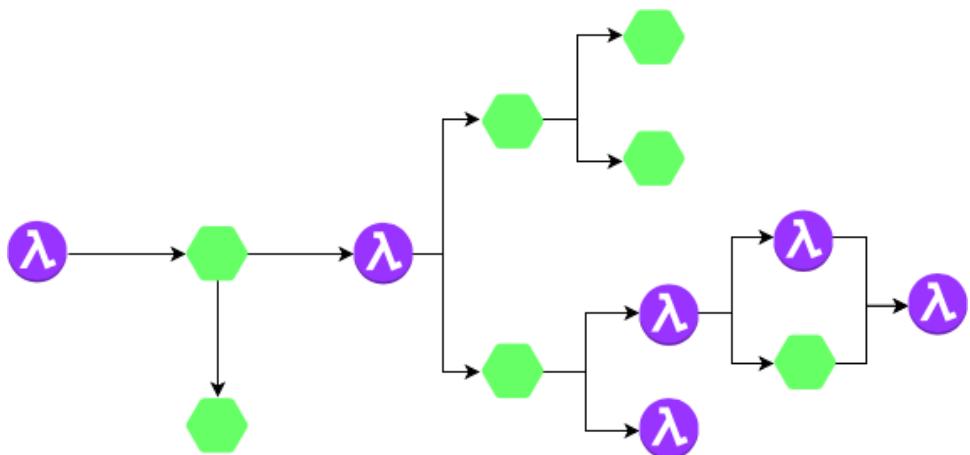


Figure 8.1: A Pulsar Application is best represented as a data pipeline through which data flows from left to right through the Functions and Microservices to implement the business logic in a series of steps.

The data pipeline acts as a distributed processing framework where each Function inside the data pipeline is an independent processing unit that can execute as soon as the next input arrives. Additionally, these loosely-coupled components communicate asynchronously with one another which allows them to operate at their own rate and not be blocked waiting for a response from another component. This in turn allows you to have multiple instances of any component running in parallel to provide the necessary throughput you require. Therefore, when you design your application, keep this in mind, as you will want to keep your functions and services as independent as possible in order to exploit this parallelism if needed. Let's revisit the order entry use case to demonstrate how you would implement it as a data pipeline similar to the one shown in Figure 8.2. As the term "dataflow" implies, it is best to focus on the flow of data shown along the bottom of the figure.

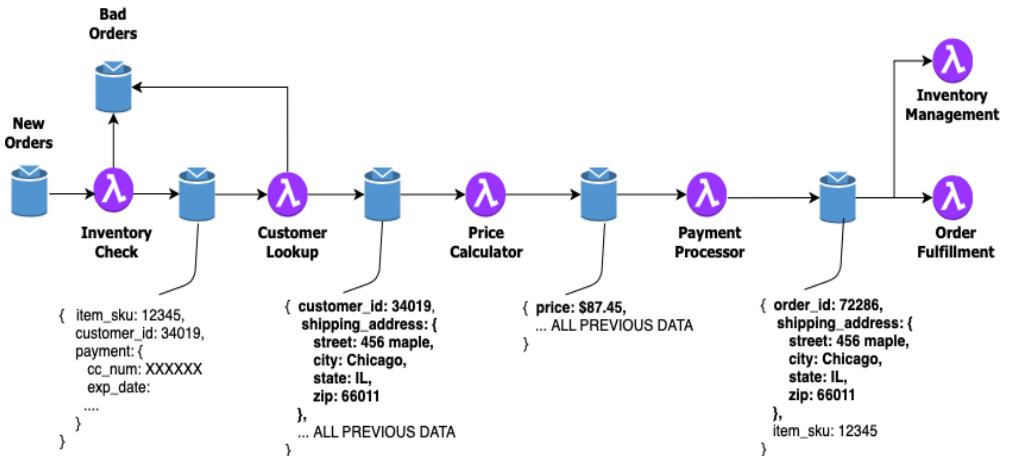


Figure 8.2: The data flow for the order entry use case. As the data flows through the various steps in the process, the original order data is augmented with additional information that will be used in the next step of the process.

As you can see, each step in the process passes the original order through along with an additional piece of information that is required in the subsequent step, e.g., the customer id and shipping address have been added to the message by the customer lookup service, etc. Since all of the data required for the processing at that stage is in the message, each function can execute as soon as the next piece of data arrives.

The payment processor removes the payment information from the message and publishes messages containing the newly generated order_id, the shipping address, and the item SKU. Multiple Functions are consuming these messages; the inventory management function uses the SKU to decrease the item count from the available inventory while the order fulfillment Function needs the SKU and the shipping address to send the item to the customer. Hopefully, this gives you a better idea of how Pulsar Function-based applications should be designed and modelled before we jump into some of the more advanced design patterns in the next section.

8.2 Message Routing Patterns

A message router is an architectural pattern that is used to control the flow of data through the topology of a Pulsar application by directing messages to different topics based on specific conditions. Each of the patterns covered in this section provide proven guidelines for dynamically routing messages and I will cover how you can implement them using Pulsar Functions.

8.2.1 Splitter

The `OrderValidationService` receives a message that contains three related pieces of information that must be validated in different ways; the delivery address, the payment

information, and the food order itself. Validating each of these pieces of information requires interacting with an external service that may have a slow response time. A naïve approach to this problem would be to perform these steps in serial fashion one after another, and if any of them failed to exit the process. However, this approach will result in very high latency time for each incoming order. This is due to the fact that when performing these three subtasks sequentially, the overall latency will be equal to the sum of the individual latencies.

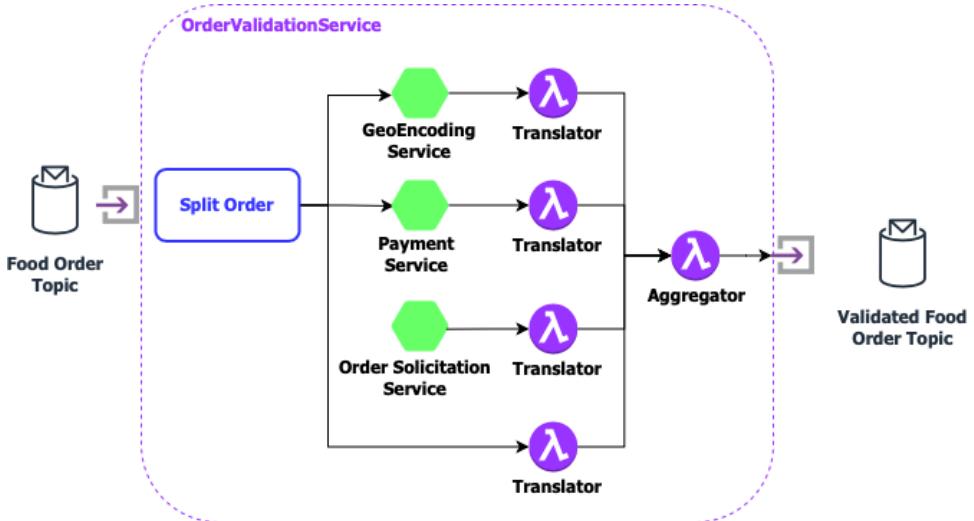


Figure 8.3: The Topology of the OrderValidationService is comprised of several other Microservices and Functions and utilizes the Splitter Pattern.

Since there are no dependencies between the results of these intermediate validation services e.g., the Payment validation isn't dependent on the result of geo-encoding, a better approach would be to have each of these tasks performed in parallel. With parallel execution, the overall latency would be reduced to the latency of the longest running subtask. In order to achieve this parallel subtask execution, the `OrderValidationService` will implement the *Splitter* pattern to break out the individual elements of the message so that they may be processed with different services. As you can see from Figure 8.3 the `OrderValidationService` is composed of several smaller functions that implement the entire validation process.

Our solution should also be efficient in terms of its use of network resources and avoid sending the entire food order item to each microservice since they only need a portion of the message in order to perform their processing. As you can see from the code in Listing 8.1, we only send a portion of the message along with the order-ID to each of these intermediate services. The order-ID will be used to correlate the results into a final result by the Aggregator function.

Listing 8.1 The OrderValidationService's Implementation of the Splitter Pattern

```
public class OrderValidationService implements Function<FoodOrder, Void> {

    private boolean initialized;
    private String geoEncoderTopic, paymentTopic,
    private String restaurantTopic, orderTopic;

    @Override
    public Void process(FoodOrder order, Context ctx) throws Exception {
        if (!initialized) {
            init(ctx);      #A
        }

        ctx.newOutputMessage(geoEncoderTopic, AvroSchema.of(Address.class))
            .property("order-id", order.getMeta().getOrderId() + "")      #B
            .value(order.getDeliveryLocation()).sendAsync();      #C

        ctx.newOutputMessage(paymentTopic, AvroSchema.of(Payment.class))
            .property("order-id", order.getMeta().getOrderId() + "")      #D
            .value(order.getPayment()).sendAsync();      #D

        ctx.newOutputMessage(orderTopic, AvroSchema.of(FoodOrderMeta.class))
            .property("order-id", order.getMeta().getOrderId() + "")      #E
            .value(order.getMeta()).sendAsync();      #E

        ctx.newOutputMessage(restaurantTopic, AvroSchema.of(FoodOrder.class))
            .property("order-id", order.getMeta().getOrderId() + "")      #F
            .value(order).sendAsync();      #F

        return null;
    }
    private void init(Context ctx) {      #G
        geoEncoderTopic = ctx.getUserConfigValue("geo-topic").toString();
        paymentTopic = ctx.getUserConfigValue("payment-topic").toString();
        restaurantTopic = ctx.getUserConfigValue("restaurant-topic").toString();
        orderTopic = ctx.getUserConfigValue("aggregator-topic").toString();
        initialized = true;
    }
}
```

#A Initialize all of the topic names so we know where to publish the messages

#B Add the order-id to the message properties so we can use it to correlate the results.

#C Send just the Address element of the message to the GeoEncoder Service.

#D Send just the Payment element of the message to the Payment Service

#E Send the food order metadata to the aggregator topic directly since we don't need to process it.

#F Send just the FoodOrder element of the message to the OrderSolicitation Service.

The asynchronous nature of the processing of these individual message elements makes collecting the results challenging. Each of these elements are processed by different services with different response times, e.g., the geo-encoder will invoke a web service, the payment service needs to communicate with a bank to secure the funds, and each restaurant will need to respond manually to either accept or reject the order. These types of issues make the process of combining multiple but related messages complicated, which is where the *Aggregator* pattern comes into play.

An aggregator is a stateful component that receives all of the response messages from the invoked services, i.e., GeoEncoder, Payment, etc. and correlates the responses back

together using the order-ID. Once a complete set of responses have been collected, a single aggregated message is published to the output topic. When you choose to implement the *Aggregator* pattern, you must consider the following three key factors:

- **Correlation:** How are messages correlated together?
- **Completeness:** When have we ready to publish the resulting message?
- **Aggregation:** How are the incoming messages combined into a single result?

For our particular use case we have decided that the order-ID will serve as the correlation id, which will help us identify which response messages belong together. The result will be considered complete only after we have received all three messages for the order, this is also referred to as the “Wait for All” strategy. Lastly, the resulting responses will be combined into a single object of type `ValidatedFoodOrder`.

Let’s take a look at the Aggregator code shown in Listing 8.2 for the implementation details. Given the strongly typed nature of Pulsar Functions, I cannot define the interface to accept multiple response object types, e.g., an `AuthorizedPayment` object from the Payment service, an `Address` type from the Geo-encoding service, etc. Therefore, I use a translator function between these services and the `OrderValidationAggregator`. Each of these translator functions converts the intermediate services natural return type into a `ValidatedFoodOrder` object which allows me to accept messages from each of these services within a single Pulsar Function.

Listing 8.2 The OrderValidationService’s Aggregator Function

```
public class OrderValidationAggregator implements Function<ValidatedFoodOrder,  
    Void> {  
  
    @Override  
    public Void process(ValidatedFoodOrder in, Context ctx) throws Exception {  
  
        Map<String, String> props = ctx.getCurrentRecord().getProperties();  
        String correlationId = props.get("order-id");  
  
        ValidatedFoodOrder order;  
        if (ctx.getState(correlationId.toString()) == null) {      #A  
            order = new ValidatedFoodOrder();  
        } else {  
            order = deserialize(ctx.getState(correlationId.toString()));      #B  
        }  
  
        updateOrder(order, in);      #C  
  
        if (isComplete(order)) {      #D  
            ctx.newOutputMessage(ctx.getOutputTopic(),  
                AvroSchema.of(ValidatedFoodOrder.class))  
                .properties(props)  
                .value(order).sendAsync();  
  
            ctx.putState(correlationId.toString(), null);      #E  
        } else {  
            ctx.putState(correlationId.toString(), serialize(order));      #F  
        }  
    }  
}
```

```

        return null;
    }

private boolean isComplete(ValidatedFoodOrder order) {      #G
    return (order != null && order.getDeliveryLocation() != null
        && order.getFood() != null && order.getPayment() != null
        && order.getMeta() != null);
}

private void updateOrder(ValidatedFoodOrder val,
                       ValidatedFoodOrder res) {      #H
    if (res.getDeliveryLocation() != null
        && val.getDeliveryLocation() == null) {
        val.setDeliveryLocation(response.getDeliveryLocation());
    }

    if (resp.getFood() != null && val.getFood() == null) {
        val.setFood(response.getFood());
    }

    if (resp.getMeta() != null && val.getMeta() == null) {
        val.setMeta(response.getMeta());
    }

    if (resp.getPayment() != null && val.getPayment() == null) {
        val.setPayment(response.getPayment());
    }
}

private ByteBuffer serialize(ValidatedFoodOrder order) throws IOException {
    ...      #I
}

private ValidatedFoodOrder deserialize(ByteBuffer buffer) throws IOException,
                                         ClassNotFoundException {
    ...      #J
}
}

```

#A Check to see if we already have received some responses for this order
#B If we have, then deserialize the bytes into a ValidatedFoodOrder object.
#C Every message will be of type ValidatedFoodOrder, but will only contain one of the four fields
#D Check to see if we have received all four messages which indicates that we are done.
#E Once the order is aggregated, we can purge it.
#F If not, serialize the object and store it in the context until the next message arrives.
#G An object is only considered complete if we have received all four messages
#H Copies over whatever fields are in the received object.
#I Helper method to convert a ValidatedFoodOrder object into a ByteBuffer
#J Helper method to read a ValidatedFoodOrder object from a ByteBuffer

It is important to point out that due to the parallel nature of streaming architectures in general, the Aggregator may receive message from multiple orders at any time and in no particular order. Therefore, the Aggregator maintains an internal list of active orders that it has already received messages for. If no list exists for a given order-ID then it is assumed to be the first message in the collection and an entry is added to the internal list. This list needs

to be purged periodically to ensure that it doesn't grow indefinitely which is why the Aggregator makes sure to purge the list once an aggregation is complete.

8.2.2 Dynamic Router

The *Splitter* pattern is useful when you want to process different pieces of the message in parallel and you already know in advance exactly how many elements you will have and that the number will remain static. However, there are situations where you cannot determine where the message will be routed ahead of time, and you must make that determination based upon the content of the message itself and other external conditions. One such example is the `OrderSolicitationService`, which is one of the three microservices invoked by the `OrderValidationService`.

This service notifies a subset of participating restaurants of incoming food orders that they can fulfill and awaits a response from each of the restaurateurs as to whether they will accept the order or not, and if so when it will be ready for pick-up. Obviously, the list of restaurants is dependent upon several factors. We want to route the orders based on the restaurants ability to provide the food, i.e., orders for Big Macs go to McDonalds, etc. At the same time, we don't want to indiscriminately broadcast the order to every single McDonald's restaurant, so we narrow the list down based upon their proximity to the delivery location. Since this list is constructed in response to each message, the *Recipient List* pattern is the best choice.

The overall flow of the `OrderSolicitationService` is depicted in Figure 8.4, which consists of three distinct phases. The first phase computes the intended list of recipients based on the factors we already discussed. During the second phase the recipient list is iterated over and the `FoodOrder` is forwarded to each recipient. The third and final phase is when the service awaits the responses from each of the recipients and selects a "winner" to fulfill the order. Once a winner is selected, all the other recipients are notified that they "lost" and that the food order is no longer available.

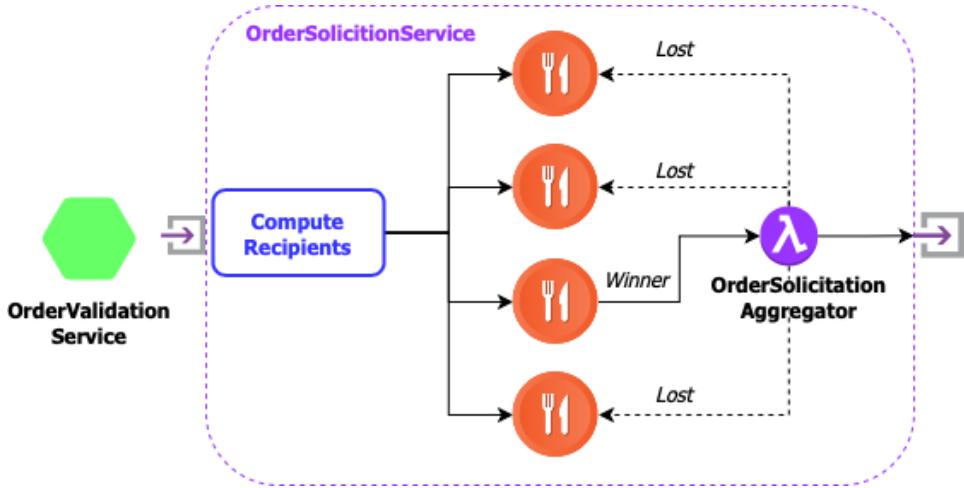


Figure 8.4: The Topology of the OrderSolicitationService which implements the Dynamic Router Pattern.

The actual implementation of this logic is shown in Listing 8.3 and relies on the message properties to convey metadata that is critical for the Aggregator. First of all, the order-ID is included with in the message to identify which `FoodOrder` the response is associated with. The `all-restaurants` property is used to encode all of the candidates that have been solicited for this order. Having this information in the message enables the Aggregator to know all of the restaurants it needs to send the “you didn’t win” message to. The last piece of metadata contained within the message properties is the `return-addr` property, which contains the name of the topic that the Aggregator is subscribed to. This allows us to avoid having to hard code this information into each message recipient’s logic, and instead we can provide this information dynamically. This is an implementation of the *Return Address* pattern defined in the *Enterprise Integration Patterns* book.

Listing 8.3 The OverSolicitationService's implementation of the Recipient List Pattern

```
public class OrderSolicitationService implements Function<FoodOrder, Void> {
    private String rendezvous = "persistent://resturants/inbound/accepted";

    @Override
    public Void process(FoodOrder order, Context ctx) throws Exception {
        List<String> cand = getCandidates(order,
                order.getDeliveryLocation());      #A

        if (CollectionUtils.isNotEmpty(cand)) {
            String all = StringUtils.join(cand, ",");
            int delay = 0;
            for (String topic: cand) {      #B
                try {
                    ctx.newOutputMessage(topic, AvroSchema.of(FoodOrder.class))
                }
            }
        }
    }
}
```

```

        .property("order-id", order.getMeta().getOrderId() + "")      #C
        .property("all-restaurants", all)      #D
        .property("return-addr", rendevois)    #E
        .value(order).deliverAfter( (delay++ * 10), TimeUnit.SECONDS); #F
    } catch (PulsarClientException e) {
        e.printStackTrace();
    }
}
}

return null;
}

private List<String> getCandidates(FoodOrder order, Address deliveryAddr) {
...      #G
}
}

```

#A Build the recipient list based on the order and delivery address

#B Send the FoodOrder to every recipient in the list

#C Use the order ID for correlation

#D Include all the restaurants so we can notify the losers.

#E Tell each recipient where to send their response message.

#F Stagger the delivery of the messages to minimize the number of rejected responses.

#G The logic to build the recipient list.

The recipient list is returned in order of preference, e.g., the restaurant closest to the delivery location, or the one that has received the least amount of business from us tonight, etc. and we use Pulsar's delayed message delivery capabilities to space out the solicitation requests. The goal of this is to minimize the number of times we need to reject a `FoodOrder` that was accepted by a restaurant. We don't want to aggravate our participating restauranteurs by bombarding them with orders that they accept but ultimately have rejected. Therefore, we scale up the number of restaurants we notify slowly to prevent having too many outstanding solicitations at the same time.

Since the `OrderSolicitationService` can send the `FoodOrder` to multiple recipients, it will need to reconcile the responses and award the order to only one of the respondents. While there are many strategies available, for now it will just accept the first response. This reconciliation logic will be implemented using an *Aggregator* similar to what we used for the `OverValidationService`. As you can see in Listing 8.3, I am using the message properties to pass along the name of the topic that each of the recipient should respond to. The corresponding Aggregator should be configured to listen to this topic so that it receives the response messages and can notify the non-winning restaurants that the order has been awarded to a different restaurant. This restauranteur's mobile application can then react to the non-winning notification by removing the order from view, etc.

Listing 8.4 The OverSolicitationService's implementation of the Aggregator Pattern

```

public class OrderSolicitationAggregator implements Function<SolicitationResponse, Void> {

    @Override
    public Void process(SolicitationResponse response, Context context)
        throws Exception {

```

```

Map<String, String> props = context.getCurrentRecord().getProperties();
String correlationId = props.get("order-id");
List<String> bids = Arrays.asList(
    StringUtils.split(props.get("all-restaurants")));      #A

if (context.getState(correlationId) == null) {      #B
    // First response wins
    String winner = props.get("restaurant-id");      #C
    bids.remove(winner);      #D
    notifyWinner(winner, context);      #E
    notifyLosers(bids, context);      #F

    // Record the time we received the winning bid.
    ByteBuffer bb = ByteBuffer.allocate(32);
    bb.asLongBuffer().put(System.currentTimeMillis());
    context.putState(correlationId, bb);
}

return null;
}

private void notifyLosers(List<String> bids, Context context) {
    ...
}

private void notifyWinner(String s, Context context) {
    ...
}

```

#A Decode all the IDs of all the solicited restaurants
#B First response back wins
#C Get the restaurant Id of the winner from the response message
#D Remove the winner from the list of all restaurants
#E Send a message to the winning restaurant letting them know
#F Send a message to all the non-winning restaurants

As you can see from Listing 8.4, the correlation will still be done by the order-ID, but the completeness criteria will be “First one wins” instead of waiting for a response from all the message recipients as we did for the *Splitter* pattern.

Even though we do our best to prevent having multiple outstanding solicitation messages, the aggregator still needs to accommodate for his scenario. It does so by retaining the time that the winning bid was received for each order. This allows the Aggregator to ignore all subsequent responses for the same order since we know that another restaurant has already been awarded the order.

In order to prevent this data structure from growing too large and causing an out-of-memory condition, I have incorporated a background process that periodically wakes up and purges all records in the list that are older than a certain period of time, which can be determined by the timestamp of the winning bid.

8.2.3 Content Based Router

A content-based router uses the message contents to determine which topic to route it to. The basic concept is that the content of each message is inspected and then is routed to a specific destination based upon value(s) found or not found in the content. For the order validation use case, the `PaymentService` receives a message that will vary depending on the payment type being used by the customer.

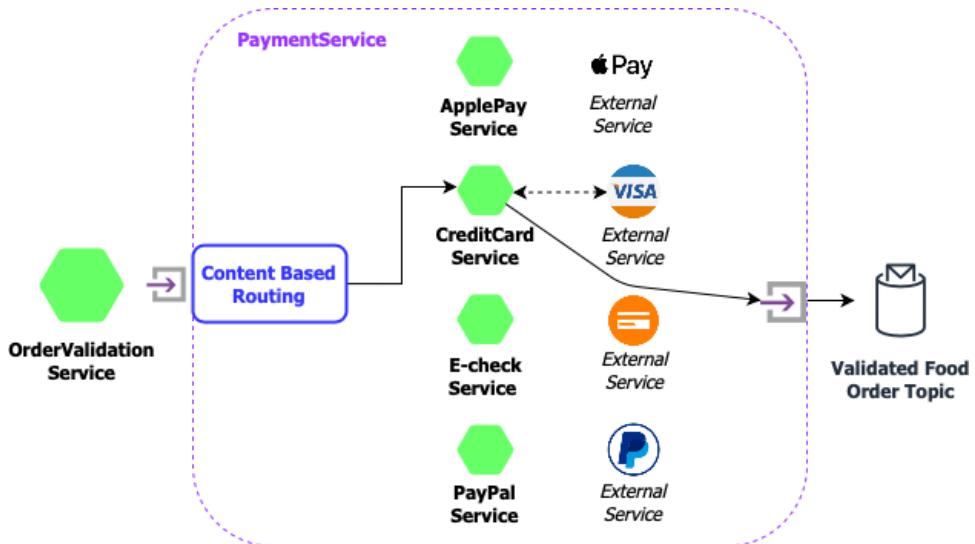


Figure 8.5: The `PaymentService` Topology implements the Content-Based-Router pattern and routes the payment information to the appropriate service based on the method of payment provided with the order.

Currently, the system supports credit card payments, PayPal, Apple Pay, and electronic check. Each of these payment methods must be validated by different external systems. Therefore, the goal of the `PaymentService` is to direct the message to the proper system based on the content of the message. Figure 8.5 depicts the scenario where the method of payment on the order was a credit card and the `Payment` details are forwarded to the `CreditCardService`.

Each of the supported payment types has an associated intermediate microservice, i.e. `ApplePayService`, `CreditCardService`, etc., that is configured with the proper credentials, endpoint, etc. These intermediate microservices then make a call to the proper external service to get payment authorization, and upon receipt of the authorization forwards the response on to the `OrderValidationService`'s Aggregator where it is combined with the other responses associated with the order. Listing 8.5 shows the implementation of the `Content-Based Routing` pattern inside the `PaymentService`.

Listing 8.5 The PaymentService's Implementation of the Content-Based-Router Pattern

```
public class PaymentService implements Function<Payment, Void> {
    private boolean initialized = false;
    private String applePayTopic, creditCardTopic, echeckTopic, paypalTopic;

    public Void process(Payment pay, Context ctx) throws Exception {
        if (!initialized) {
            init(ctx);
        }

        Class paymentType = pay.getMethodOfPayment().getType().getClass();
        Object payment = pay.getMethodOfPayment().getType();

        if (paymentType == ApplePay.class) {
            ctx.newOutputMessage(applePayTopic, AvroSchema.of(ApplePay.class))
                .properties(ctx.getCurrentRecord().getProperties())
                .value((ApplePay) payment).sendAsync();      #A
        } else if (paymentType == CreditCard.class) {
            ctx.newOutputMessage(creditCardTopic, AvroSchema.of(CreditCard.class))
                .properties(ctx.getCurrentRecord().getProperties())
                .value((CreditCard) payment).sendAsync();    #B
        } else if (paymentType == ElectronicCheck.class) {
            ctx.newOutputMessage(echeckTopic, AvroSchema.of(ElectronicCheck.class))
                .properties(ctx.getCurrentRecord().getProperties())
                .value((ElectronicCheck) payment).sendAsync(); #C
        } else if (paymentType == PayPal.class) {
            ctx.newOutputMessage(paypalTopic, AvroSchema.of(PayPal.class))
                .properties(ctx.getCurrentRecord().getProperties())
                .value((PayPal) payment).sendAsync();          #D
        } else {
            ctx.getCurrentRecord().fail();      #E
        }
    }

    return null;
}

private void init(Context ctx) {      #F
    applePayTopic = (String)ctx.getUserConfigValue("apple-pay-topic").get();
    creditCardTopic = (String)ctx.getUserConfigValue("credit-topic").get();
    echeckTopic = (String)ctx.getUserConfigValue("e-check-topic").get();
    paypalTopic = (String)ctx.getUserConfigValue("paypal-topic").get();
    initialized = true;
}
```

#A Send ApplePay objects to the ApplePayService
#B Send CreditCard objects to the CreditCardService
#C Send ElectronicCheck objects to the ElectronicCheckService
#D Send PayPal objects to the PayPalService
#E Reject any other payment method
#F The output topics for the intermediate services are configurable.

After the `CreditCardService` receives an authorization number for the transaction, it is then sent to `validated food order` topic since we will need that authorization number in order to collect the funds, etc.

8.3 Message Transformation Patterns

Common examples of streaming data include IoT sensors, server and security logs, real-time advertising, and click-stream data from mobile apps and websites. In each of these scenarios we have data sources that are continuously generating thousands or millions unstructured or semi-structured data elements, most commonly plain text, JSON, or XML. Each of these data elements must be transformed into a format that is suitable for processing and analysis.

This category of processing is common among all streaming platforms, and these data transformation tasks are similar to traditional ETL processing since the primary concern is to ensure that the ingested data is normalized, enriched, and transformed into a format more suitable for processing. Message transformation patterns are used to manipulate the content of the messages as they flow through the DAGs topology to address these types of issues within your streaming architecture. Each of the patterns covered in this section provide proven guidelines for dynamically transforming messages.

8.3.1 Message Translator

As we saw earlier, the `OrderValidationService` makes several asynchronous calls to different services, each of which produces messages with different schema types. All of these messages must be combined by the `OrderValidationAggregator` into a single response. However, a Pulsar Function can only be defined to accept incoming messages of a single type so we cannot publish these messages directly to the service's input topic as shown in Figure 8.6 as the schemas would not be compatible.

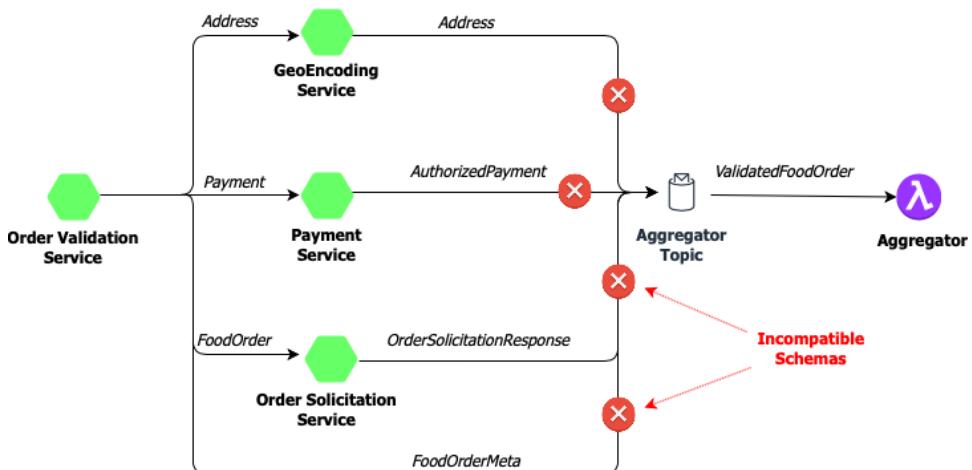


Figure 8.6: Each of the intermediate microservices produce messages with different schema types. Therefore, they cannot publish them directly to the `OrderValidationAggregator`'s input topic.

In order to accommodate the consumption of messages with different schemas, the results from each of these intermediate microservices must be routed through a message

translator function which converts these response messages into the same type as the `OrderValidationAggregator`'s input topic, which in this case is the schema shown in Listing 8.6. I have chosen to use an object type that is a composite of each of these message types. This approach allows me to simply copy the response from each of the intermediate services directly into the corresponding placeholder inside the `ValidatedFoodOrder` object.

Listing 8.6 The ValidatedFoodOrder Definition

```
record ValidatedFoodOrder {  
    FoodOrderMeta meta;      #A  
    com.gottaeat.domain.restaurant.SolicitationResponse food;    #B  
    com.gottaeat.domain.common.Address delivery_location;    #C  
    com.gottaeat.domain.payment.AuthorizedPayment payment;    #D  
}
```

#A The FoodOrderMeta data forwarded from the OrderValidationService

#B The response type from the OrderSolicitationService

#C The response type from the GeoEncodingService

#D The response type from the PaymentService

While the logic for consuming each response message type is slightly different, the concept is essentially the same. As you can see from Listing 8.7 the logic for handling the `AuthorizedPayment` messages produced by the `PaymentService` is straight-forward. Simply create an object of the appropriate type and copy over the `AuthorizedPayment` object published by the `PaymentService` into the corresponding field in the wrapper object before sending it to the Aggregator for consumption.

Listing 8.7 The PaymentAdaptor Implementation

```
public class PaymentAdapter implements Function<AuthorizedPayment, Void> {  
  
    @Override  
    public Void process(AuthorizedPayment payment, Context ctx)  
        throws Exception {  
        ValidatedFoodOrder result = new ValidatedFoodOrder();    #A  
        result.setPayment(payment);    #B  
  
        ctx.newOutputMessage(ctx.getOutputTopic(),  
            AvroSchema.of(ValidatedFoodOrder.class))    #C  
            .properties(ctx.getCurrentRecord().getProperties())    #C  
            .value(result).send();  
  
        return null;  
    }  
}
```

#A Create the new wrapper object

#B Update the payment field with the `AuthorizedPayment`

#C Publish a message of type `ValidatedFoodOrder` to the Aggregator's input topic.

#D Copy over the order-ID so we can correlate it with the other response messages.

There are similar adaptors for the each of the other microservices as well. Once all of the object values have been populated, the order is considered “validated” and can be published to the validated food order topic for further processing.

It is worth noting that each of these adaptor functions must consume from a topic that stores the microservice’s respective response messages before publishing the wrapper objects to the `OrderValidationAggregator`’s input topic. Therefore, you will need to create these response topics and configure the microservices to publish to them as shown in Figure 8.7.

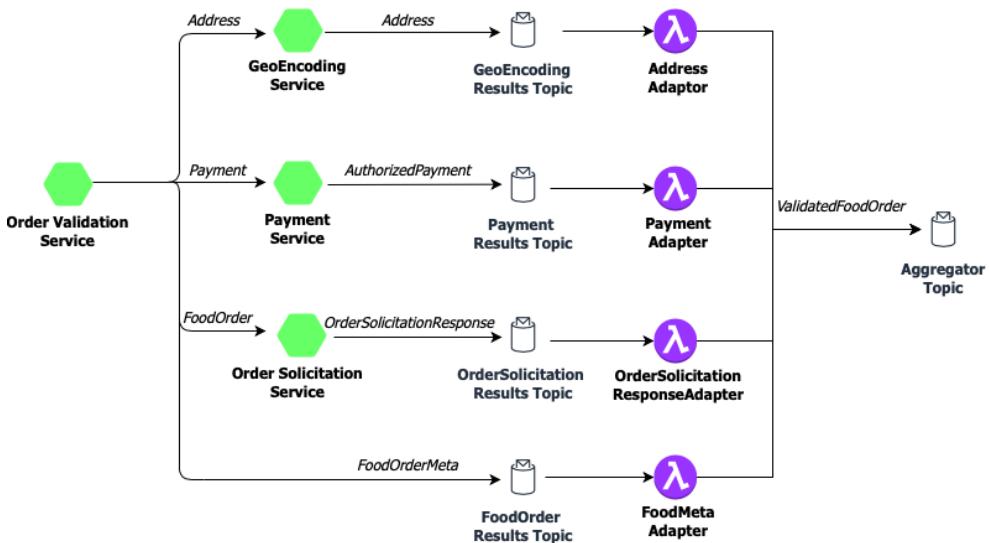


Figure 8.7: Each of the intermediate microservices publish their response messages to topics with the appropriate schema type. The associated adapter functions then consume those messages and convert them to the `ValidatedFoodOrder` schema type expected in the Aggregator topic.

In addition to being useful in situations where you need to combine the results of several different, this pattern can also be used to accommodate the ingestion of data from external systems such as a database, etc. and translate them into the required schema format for processing.

8.3.2 Content Enricher

When processing streaming events, it is often useful to augment the incoming message with additional information that the downstream system requires. In our example, the incoming customer order event to the `OrderValidationService` will contain a raw, unvalidated street address, but the consuming microservices also require a geo-encoded location with a latitude and longitude pair in order to provide navigation to the delivery drivers, etc.

This type of problem is typically addressed with the *Content Enricher Pattern*, which is the term for a process that uses information inside an incoming message to augment the original

message contents with the new information. In our particular use case, we will retrieve data from an external source and add it to the outgoing message as shown in Figure 8.8. Our GeoEncodingService will take the delivery address details provided by the customer and pass them to the Google Maps API webservice. The resulting latitude and longitude that corresponds to the address will then be included in the outbound message.

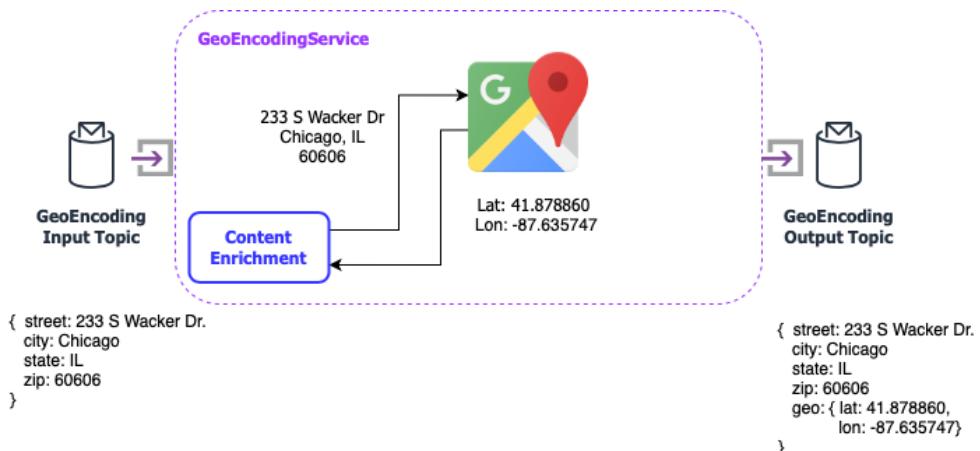


Figure 8.8: The GeoEncoderService implements the Content Enrichment pattern by using the provided street address to lookup the corresponding Latitude and Longitude and adding it to the Address object.

Listing 8.8 shows the implementation of the GeoEncoderService that invokes the Google Maps service and augments the incoming Address object with the associated latitude and longitude value.

Listing 8.8 The GeoEncoderService's implementation of the Content Enricher Pattern

```

public class GeoEncoderService implements Function<Address, Address> {
    boolean initialized = false;
    String apiKey;
    GeoApiClient geoContext;

    public void process(Address addr, Context context) throws Exception {
        if (!initialized) {
            init(context);
        }

        Address result = new Address();
        result.setCity(addr.getCity());
        result.setState(addr.getState());
        result.setStreet(addr.getStreet());
        result.setZip(addr.getZip());

        try {
            GeocodingResult[] results =
                GeocodingApi.geocode(geoContext, formatAddress(addr)).await();
        }
    }
}

```

```

        if (results != null && results[0].geometry != null) {
            Geometry geo = results[0].geometry;
            LatLon ll = new LatLon();
            ll.setLatitude(geo.location.lat);
            ll.setLongitude(geo.location.lng);
            result.setGeo(ll);
        }

        } catch (InterruptedException | IOException | ApiException ex) {
            context.getCurrentRecord().fail();
            context.getLogger().error(ex.getMessage());
        } finally {
            return result;
        }
    }

    private void init(Context context) {
        apiKey = context.getUserConfigValue("apiKey").toString();
        geoContext = new GeoApiClientBuilder()
            .apiKey(apiKey).maxRetries(3)
            .retryTimeout(3000, TimeUnit.MILLISECONDS).build();
        initialized = true;
    }
}

```

The service uses an API key provided by the configuration properties to invoke the Google Maps web service and uses the first response it gets as the source of the latitude and longitude values. If the call to the web service isn't successful, we fail the message so that it can be retried at a later time.

While using an external service is one of the more common usages for the *Content Enrichment Pattern*, there are implementations that merely perform internal calculations based on the message contents such as computing the message size or MD5 hash of the message contents and adding that to the message properties. This allows the message consumer to validate that the message contents have not been altered.

Another common use case is to retrieve the current time from the operating system and appending that timestamp to the message to indicate when it was received or processed. This information is useful for maintaining the message order if you wish to process the messages in the order they were received, or for identifying bottlenecks in the DAG if you appended a received timestamp at each step of the process.

8.3.3 Content Filter

Often times it can be useful to remove or mask one or more data elements from an incoming message due to security concerns, etc. The content filter pattern is essentially the inverse of the content enricher pattern because it is designed to perform this type of transformation.

The `OrderValidationService` that we discussed earlier is an example of a content filter that breaks up the incoming `FoodOrder` message into smaller individual pieces before routing them to the appropriate microservice for processing. Not only does this minimize the amount of data sent to each service, but it also hides the sensitive payment information from all the other services that do not require access to that information.

Consider another scenario where an event contains a sensitive data element such as a credit card number. The content filter can detect such patterns in the messages and remove the data element entirely, mask it with a one-way hashing function such as SHA-256 or encrypt the data field.

8.4 Summary

- The Pulsar Functions framework is a distributed processing framework that is well suited for DataFlow programming where the data is processed in stages that can be executed in parallel like an assembly line.
- Applications based on Pulsar Functions can be modelled as data pipelines where the Functions perform the computations, and direct data between one another using the input/out topics.
- When designing your message-passing microservice application, it is often to use existing design patterns such as those found in the *Enterprise Integration Patterns* book and other sources.
- Well established messaging patterns can be implemented using Pulsar Functions, which allows you to use time-tested solutions inside your applications.

9

Resiliency Patterns

This chapter covers:

- Making your Pulsar Function based applications resilient to adverse events
- Implementing well-established resiliency patterns using Pulsar Functions

As the architect of the GottaEat order entry microservice, your primary goal is to develop a system that can accept incoming food orders from customers 24 hours a day, 7 days a week and within an acceptable response time to the customer. Your system must be available at all times; otherwise, your company will lose not only revenue and customers, but its reputation will suffer as well. Therefore, you must design your system to be both highly-available and resilient in order to provide continuity of service. Everyone wants their systems to be resilient, but what does that actually mean? Resilience is the ability of a system to withstand disruptions caused by adverse events and conditions while maintaining an acceptable level of performance relative to any number of quantitative metrics such as availability, capacity, performance, reliability, robustness, and usability.

Being resilient is important because no matter how well your Pulsar application is designed, an unanticipated incident such as the loss of electrical power or network communications will eventually emerge that will disrupt the topology. Implicit in the previous statement is the idea that adverse events and conditions will occur, it really isn't a matter of if, but when. Resiliency is about what your software does when these disruptive events occur. Does the Pulsar Function detect these events and conditions? Does it properly respond to them once they are detected? Does the Function properly recover afterward?

A highly resilient system will utilize several reactive resiliency techniques to actively detect these adversities and respond to them in order to return the system back to its normal operating state automatically as shown in Figure 9.1. This is particularly useful in a streaming environment, where any disruption of service can result in the data not being captured from the source and lost forever.

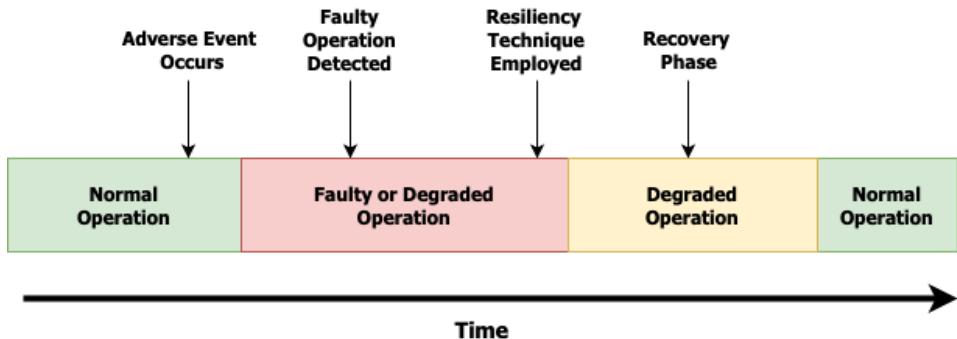


Figure 9.1: A resilient system will automatically detect adverse events or conditions and take proactive measures to return itself to a normal operating state.

Obviously, the key to employing any reactive technique is the ability to detect the adverse conditions. In this chapter we will cover how to detect faulty conditions with a Pulsar Function application, and some of the resiliency techniques that you can use within your Pulsar Functions to make them more resilient.

9.1 Pulsar Function Resiliency

As we saw in Chapter 8, all Pulsar Function based applications are essentially topologies consisting of several individual Pulsar Functions interconnected via input and output topics. These topologies can be thought of as *directed-acyclic-graphs* (DAGs) with data flowing through different paths based on the values of the messages. From a resiliency perspective it makes sense to consider this entire DAG as the “system” that must be insulated from the impact of adverse conditions.

9.1.1 Adverse Events

In order to implement an overall resiliency strategy, we must first identify all of the adverse events that could potentially impact a running Pulsar application topology. Rather than attempt to list out every single possible condition that can occur with a Pulsar Function, we will classify the adverse events into the following categories: Function death, lagging function, and non-progressing function.

FUNCTION DEATH

Let’s start with the most drastic event first, *Function Death*, which would be when a Pulsar Function within the application terminates abruptly. This condition can be caused by any number of physical factors such as a server crash or power outage or non-physical factors such as an out of memory condition within the Function itself. Regardless of the underlying reason, the impact on the system will be severe as data flowing through the DAG will come to an abrupt halt.

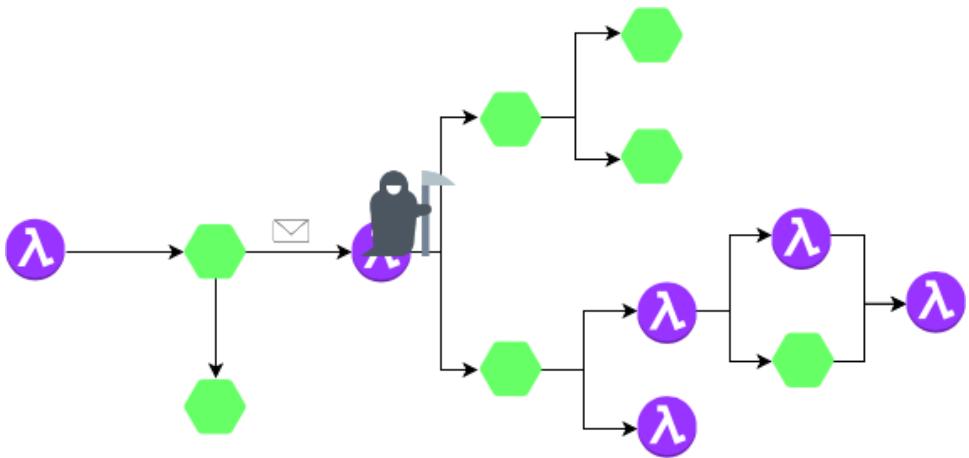


Figure 9.2: When a function dies and stops consuming messages, the entire application is essentially blocked at that point in the DAG and all downstream processing will stop.

If the given function shown in Figure 9.2 then all of the downstream consumers will stop receiving messages and the DAG will be essentially blocked at this point in the flow. From an end user perspective, the application will have stopped producing output, which in our situation means that the entire food order entry process would come to a screeching halt.

Fortunately, the functions input topic will act as a buffer and retain the incoming messages up to the limit imposed by the message retention policy for the topic. I only mention this caveat to impress upon you the importance of resolving this issue sooner rather than later, because eventually messages could get dropped if the function is not restarted.

If you are using the recommend Kubernetes runtime for hosting your Pulsar Functions, then the Pulsar Functions framework will automatically detect the failure of the associated K8s pod and restart it for you as long as you have sufficient computing resources in your Kubernetes environment. You should also have proper monitoring and alerting in place to detect the loss of a physical host and respond accordingly as an additional layer of resiliency.

LAGGING FUNCTION

Another condition that will have a negative impact on the performance of a Pulsar application is the *Lagging Function*, which occurs when there is a **sustained** arrival rate of messages into a Function's input topic that is greater than the Function is capable of processing. Under these circumstances, the Function will fall further and further behind in the processing of the incoming messages which will lead to a growing lag between the time a message is ready to be processed and when it eventually is processed.

Let's consider a scenario where the Function A is publishing 150 events per second to another Function's input topic, as shown in Figure 9.3. Unfortunately, the second Function can only process 75 events per second which leaves a 75 event per second deficit that is getting buffered inside the topic. If this processing imbalance remains in place the volume of messages within the input topic will continue to grow.



Figure 9.3: Function B is only able to process 75 events per second, while Function A is producing events at a rate of 150 per second. This leads to a backlog of 75 events per second, which introduces 1 second of latency per second.

Initially this lag will start to impact the business SLA, and in our use case the order entry process will be slow for our customers as they have to wait for their order to get processed and confirmed. Eventually, the lag will be too great for customers to bear and they will abandon their orders due to the lack of responsiveness of the mobile application. This could lead to situations where the customer's order is placed into the queue and processed AFTER the customer has decided due to the lack of a response that their order was never placed, which would be a nightmare from a business process as we would have to refund the charged amount and notify the restaurant that order has been cancelled.

To put some numbers behind this statement, let's imagine the scenario where such a condition was to start inside our order validation DAG during a peak business time such as a Friday night around 7:00 pm. Orders that were placed 10 minutes later would be placed behind the 45,000 ($75 \text{ eps} * 60 \text{ sec} * 10 \text{ minutes}$) other orders that have built up inside Function B's input topic. At a processing rate of 75 per second, it would take 10 minutes to process those existing messages before finally processing the order that was placed at 7:10.

Therefore, in order to meet your business SLAs and avoid abandoned orders due to a lagging function we will need to conduct performance testing to determine the average processing rate of each Function in the order validation service and continuously monitor it for any such processing imbalance in the future.

Fortunately, if you are using the recommended Kubernetes runtime for hosting your Pulsar Functions, then the Pulsar Functions framework allows you to scale up the parallelism of any Function using the command shown in Listing 9.1, which should help alleviate the imbalance. Therefore, the remedy for this adverse event is to update the Function's parallelism count to an acceptable level. Since the message input vs consumed ratio is 2:1 in our current hypothetical example, you would want the ratio of Function A vs Function B instances to be at least the same, if not more.

Listing 9.1 Increasing the Function Parallelism

```
$ bin/pulsar-admin functions update \
--name FunctionA \
--parallelism 5 \
...
```

While adjusting the ratio of instances to be exactly 2:1 would theoretically alleviate the problem, I would recommend having one or two additional instances beyond that ratio on the consumer side. Not only would this provide excess processing capacity that would allow your application to handle any additional surge in messages, but it would also make your Function resilient to the loss of one or more instances before any lag is experienced.

NON-PROGRESSING FUNCTION

A *Non-Progressing Function* is different from a lagging function in two aspects, the first is in their ability to successfully process messages. With a lagging function all of the messages are getting processed successfully albeit at too slow of a pace to keep up, whereas with a non-progressing function some or all of the messages cannot be processed successfully.

The second aspect is the way in which the problem can be resolved. With a lagging function the most common remedy is to increase the parallelism of the function instances to accommodate the incoming message volume. Unfortunately, there is no easy fix for a non-progressing function and the only resolution is to add processing logic inside your Pulsar Function to detect and react to these processing exceptions. So, let's take a step back and review the limited number of options we have when dealing with processing exceptions within a Pulsar Function.

You can effectively ignore the error entirely and explicitly acknowledge the message anyways which tells the Broker that you are done processing it. Obviously, this is not a viable option for some use cases, such as our Payment Service where we need an authorized payment to continue processing the order. Another option is to negatively acknowledge (aka negative ack) the message within a catch block of your Pulsar Function code, which tells the Broker to redeliver the message after a one-minute delay. Lastly, there is the possibility that no acknowledgment is sent from your function at all, due to an uncaught exception or a network timeout when calling a remote service, etc. As you can see from Figure 9.4, in either case these messages will be marked for redelivery.

As more and more of these negatively acknowledged messages build up in the topic, the system will gradually slow as they are repeatedly tried, fail, and tried again. This wastes processing cycles on non-productive work and what's worse is that these messages will never get cleared from topic which will only compound their impact over time. Hence, the term non-progressing function, as it is failing to make progress on the new messages.

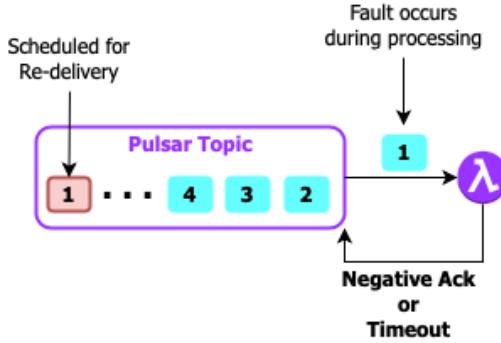


Figure 9.4: When a Pulsar Function sends a negative acknowledgement or fails to acknowledge a message within the configured time, the message is scheduled for redelivered after a one-minute delay.

Let's revisit the scenario where a Function can successfully process 75 events per second, and that Function is making a call to a remote service such as a credit card authorization service. Furthermore, the endpoint that the function calling is actually a load balancer which distributes the calls across 3 different instance of the service and one of them goes down. Every third message will immediately throw an exception and the message will be negatively acked, resulting in approximately 25 events per second not succeeding. This would drop the effective processing rate of the Function from 75 eps to 50 eps. This decrease in the processing rate is the first telltale sign of a non-progressing function.

Scaling up the instances will not solve the problem either, because that will also scale up the number the faults at a commensurate rate. If we were to double the parallelism of the functions to achieve a 150 eps consumption rate, we would end up with 50 eps that are failing and getting reprocessed. In fact, no matter how many instances we add, a third of all messages will still need reprocessing. It is also worth noting that each of these reprocessed messages will experience a one-minute latency penalty as well which will have a negative impact on your applications perceived responsiveness.

Now let's consider the impact of a network outage on this same Function. Every time a message is processed a call is made to the load balancer, but since the network is down, we cannot reach it. Not only do 100% of the messages fail, but each message takes 30 seconds to process since they have to wait for a network timeout exception to be thrown. The impact to the DAG will be the same as if the function had died, but unfortunately it hasn't. Instead, this Function is effectively in a zombie state and even restarting the Function won't help either since the underlying issue is external to the Function itself.

While the underlying issues preventing the messages from being processed can be vastly different, they can be classified into two broad categories based on their likelihood of self-correcting: *transient faults* and *non-transient faults*. Transient faults include the temporary loss of network connectivity to components and services, the momentary unavailability of a service, or timeouts that can occur when your Pulsar Function calls a remote service that is busy. These types of faults are often self-correcting, and if the message is reprocessed after a suitable delay, it is likely to succeed. One such scenario would be if the Payment Service is

making a call to an external credit card authorization service during a period where the service was overloaded with requests. There is a high probability of a subsequent call succeeding once the service has had a chance to recover.

Non-transient faults, in contrast, require external intervention in order to be resolved and include things like catastrophic hardware failures, expired security credentials, or bad message data. Consider the scenario where the order validation service publishes a message to the Payment Service's input topic that contains invalid payment information such as a bad credit card number. No matter how many times we attempt to get authorization to use the card as a method of payment it will always be declined. Another potential scenario would be where our credentials for the Payment Service have expired and consequently, all of our attempts to authorize the ANY customer's credit card will fail.

Often times it will be difficult to distinguish one scenario from the other and we need to be able to detect non-transient faults such as the invalid credit card number from transient faults, such as an overloaded remote service, so that we can respond to them differently. Fortunately, we can use the corresponding exception types and other data to make intelligent guesses as to the transient nature of a given fault and determine the proper course of action accordingly.

9.1.2 Fault Detection

When it comes to the detection of faulty conditions within a Pulsar Function topology, one only need to examine the degree to which data is flowing through the entire topology. Simply put, is the application keeping up with the data volume being fed to it from the input topic or is it falling behind? Data should flow through the DAG just like blood flowing through your body, uninterrupted and at a steady pace. There shouldn't be any blockages that are cutting off the flow to certain areas.

All of the previous adverse events we discussed thus far all have a similar impact on the flow of data, a steady increase in un-processed messages. Within Pulsar the key metric that would indicate such a blockage would be message backlog. The presence of an ever-increasing message backlog at the Pulsar application's input topic is an indication of degraded or faulty operation. To be clear, I am not talking about the absolute number of messages in the backlog, but rather the *trend* of that number itself over a period of time such as your peak business hours.

When a Pulsar application or Function cannot keep up with the growing data volume in its input topic as shown in figure 9.5, this condition is known as *back-pressure* and is indicative of lack of processing capacity and degraded performance within the application which must be remedied in order to meet the business SLAs. The term back-pressure is borrowed from fluid dynamics and used to indicate some resistance or opposing force restricting the desired flow of data through the topology, just like a clog in your kitchen sink creates an opposing force to the water draining. Similarly, this condition is not isolated to the Pulsar Function that is consuming the messages, but also has an impact throughout the entire topology.

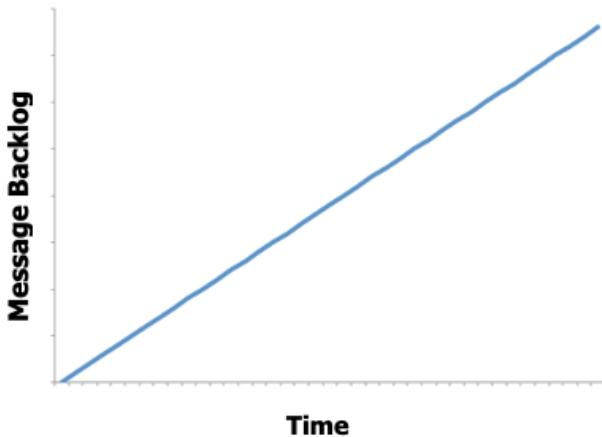


Figure 9.5: The condition where the message backlog for a particular subscription increases steadily over time is referred to as back-pressure and is indicative of degraded performance within a Pulsar Function or application.

All of the adverse events we have discussed thus far; function death, function Lag, and non-progressing functions can be detected by the presence of back-pressure on the Function's input topic(s). You should monitor the following topic-level metrics to detect back-pressure within a Pulsar Function:

- **pulsar_rate_in** that measures the rate at which messages are coming into the topic in messages per second.
- **pulsar_rate_out** that measures the rate at which messages are being consumed from the topic in messages per second.
- **pulsar_storage_backlog_size** that measures the total backlog size of the topic in bytes.

All of these metrics are periodically published to Prometheus and can be monitored by any observability framework that integrates with that platform. Any increase in message backlog within any of the Function's input topics is indicative of one or more of these events and should trigger an alert.

9.2 Resiliency Design Patterns

In the previous section we discussed some of the options for providing resiliency to your Pulsar applications using features provided by the Pulsar Function framework itself such as automatic restarts for Functions that die. However, it is not uncommon for your Pulsar Functions to have to interact with an external system to perform its processing. Doing so indirectly introduces the resiliency issues of these external systems into your Pulsar application. If these remote systems are unresponsive, the result will be lagging or non-progressing functions inside your application.

As we saw in Chapter 8, the GottaEat order validation process depends upon several third-party services in order to accept any incoming food orders and if we are unable to communicate with these external systems for any reason, our entire business will come to a complete halt. Given that all of our interaction with these services is over a network connection there is the distinct possibility of intermittent network failures, periods of high latency, and unreachable services, etc. Therefore, it is critical that our application be resilient to these types of failures and able to recover from them automatically. While you could attempt to implement these patterns yourself, this is one of those scenarios where it is best to use a third-party library that was developed by experts to solve these types of problems.

Issues arising from interacting with remote services are so common that Netflix developed its own fault tolerance library named **Hystrix** and open-sourced it in 2013 which contained several resiliency patterns that deal with exactly these types of issues. While this library is no longer being actively developed, several of its concepts have been implemented in a new open-source project called resilience4j. As we shall see, this makes it easy to utilize these patterns inside your Java based Pulsar Functions because the majority of the logic has already been implemented inside the resilience4j library itself, allowing you to focus on which patterns to use. Therefore, we will need to add the configuration shown in Listing 9.1 to the dependencies section of the Maven pom.xml file in order to add the library to our project.

Listing 9.1 Adding the resilience4j Library to the Project

```
<dependencies>
    <dependency>
        <groupId>io.github.resilience4j</groupId>
        <artifactId>resilience4j-all</artifactId>
        <version>1.7.1</version>
    </dependency>

    ...
</dependencies>
```

In the following sections I will introduce these patterns and how they can be used to make your Pulsar Function-based microservices resilient to these failure scenarios, specifically those that interact with external systems over the network. In addition to providing the context and problem that the pattern solves, I will also cover the issues and considerations to take when implementing the pattern, and examples of when the pattern would be applicable to your use case. Lastly, it is worth noting that these patterns were designed such that they can be used in combination with one another. This is particularly useful when you need to utilize more than one of these patterns within your Pulsar Function, e.g., you may want to use both the retry and circuit breaker patterns when interacting with an external web service.

9.2.1 Retry Pattern

When communicating with a remote service any number of transient faults may occur including loss of network connectivity, the temporary unavailability of a service, and service timeouts that can occur when the remote service is extremely busy. These types of faults are

generally self-correcting and subsequent calls to the service are likely to succeed. If you encounter such a failure inside your Pulsar Function, then the *Retry* pattern allows you to handle the failure in one of three ways depending on the error; if the error indicates that the failure isn't transient in nature, such as an authentication failure due to bad credentials, then you should not retry the call since the same failure is likely to occur.

If the exception indicates that the connection either timed out or otherwise indicates that request was rejected due to the system being busy, then it is best to wait for a period of time before retrying the call, in order to allow the remote service time to recover. Otherwise, you may want to retry the call immediately since we have no reason to believe that the subsequent call will fail. In order to implement this type of logic inside your Function, you would need to wrap the remote service call inside a Decorator as shown in Listing 9.2

Listing 9.2 Utilizing the Retry Pattern inside a Pulsar Function

```
import io.github.resilience4j.retry.Retry;
import io.github.resilience4j.retry.RetryConfig;
import io.github.resilience4j.retry.RetryRegistry;      #A
import io.vavr.CheckedFunction0;
import io.vavr.control.Try;

public class RetryFunction implements Function<String, String> {

    public String apply(String s) {      #B

        RetryConfig config =
            RetryConfig.custom()      #C
                .maxAttempts(2)      #D
                .waitDuration(Duration.ofMillis(1000))    #E
                .retryOnResult(response -> (response == null))    #F
                .retryExceptions(TimeoutException.class)      #G
                .ignoreExceptions(RuntimeException.class)    #H
                .build();

        CheckedFunction0<String> retryableFunction =
            Retry.decorateCheckedSupplier(
                RetryRegistry.of(config).retry("name"),    #I
                () -> {      #J
                    HttpGet request =
                        new HttpGet("http://www.google.com/search?q=" + s);

                    try (CloseableHttpResponse response =
                            HttpClient.createDefault().execute(request)) {
                        HttpEntity entity = response.getEntity();
                        return (entity == null) ? null : EntityUtils.toString(entity);
                    }
                });

        Try<String> result = Try.of(retryableFunction);    #K
        return result.getOrDefault();    #L
    }
}
```

#A We rely on several classes within the resilience4j library.

#B This is the method defined in the Function interface that will be invoked for each message.

#C Create our own custom retry configuration.

#D Specifies a maximum of two retry attempts.
 #E Specifies a pause of 1 second between retries.
 #F Perform a retry if the returned result is null.
 #G Specifies the list of exceptions that are treated as transient faults and retried
 #H Specifies the list of exceptions that are treated as non-transient and not retried
 #I Decorate the function with our custom retry configuration
 #J Provide the lambda expression that will be executed and retried if necessary
 #K Executes the decorated function until a result is received or the retry limit is reached.
 #L Return the result or null if no response was received.

The code shown in Listing 9.2 simply takes as input a String and calls the Google search API with the given input String and returns the result. Most importantly, it will make multiple attempts to call the HTTP endpoint without having to negatively acknowledge the message and have it redelivered to the function. Instead, the message is delivered from the Pulsar topic just once, and the retries are all made within the same call to the Pulsar Function's apply method as shown in Figure 9.6. This allows us to avoid the wasteful cycle of having the same message delivered multiple times before deciding to give up on the external system.

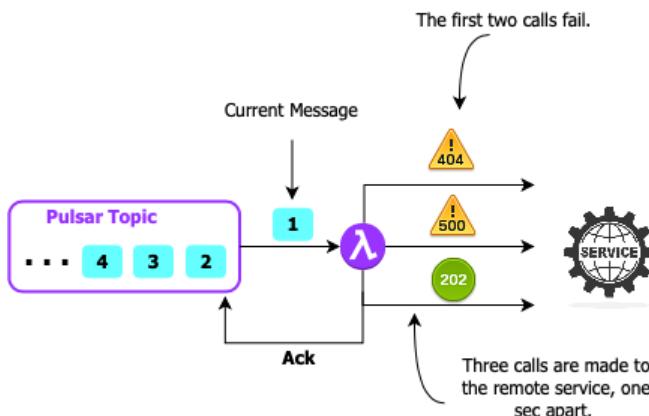


Figure 9.6: When using the Retry pattern, the remote service is called repeatedly with the same message. In this particular case the first two calls fail, but the third one succeeds so the message is acknowledged. This allows us to avoid having to negatively ack the message and delay the processing by 1 minute each time.

The first parameter passed into the `Retry.decorateCheckedSupplier` method is the `Retry` object we configured earlier in the code which defines the number of retry attempts to make, how long to pause in between them, and which exceptions indicate that we should retry the function call and which ones indicate we should not.

For those of you not familiar with the use of Lambda expressions inside Java, the actual logic that will be called is encapsulated inside the second parameter, which takes in a function definition as indicated by the `() ->` syntax and includes all the code inside the following brackets. The resulting object is a decorated function that is then passed into the `Try.of` method which handles all of the retry logic automatically for you. The term

decorated comes from the decorator pattern which is a well-known design pattern that allows behavior to be dynamically added to an object runtime.

While you could have implemented similar logic using a combination of try/catch statements and a counter for the number of attempts, etc. one can easily argue that the decorated function approach provided by the resilience4j library is much more elegant solution which also allows you to dynamically configure the retry properties via user configuration properties provided when the Pulsar Function is deployed.

ISSUES AND CONSIDERATIONS

While this is a very useful pattern to use when interacting with an external system such as a web service or a database, be sure to consider the following factors when implementing it inside your Pulsar Function:

- Adjust the retry interval to match the business requirements of your application. An overly aggressive retry policy with very short intervals between retries could generate additional load on service that is already overloaded, making matters even worse. One might want to consider using an exponential backoff policy that increases the time between the retries in an exponential manner, e.g., 1 sec, 2 sec, 4 sec, 8 sec, etc.
- Consider the criticality of the operation when choosing the number of retries to attempt. In some scenarios it may be best to fail fast rather than impact the processing latency with multiple retry attempts. For a customer facing application for example, it is better to fail after a small number of retries with a short delay between them than to introduce a lengthy delay to the end user.
- Be careful when using this pattern on operations that are idempotent, otherwise you might experience unintended side effects. For instance, a call made to an external credit card authorization service that is received and processed successfully but fails to send a response. Under these circumstances, if the retry logic sends the request again the customer's credit card would be charged twice.
- It is important to log all retry attempts so that the underlying connectivity issues can be identified and corrected as soon as possible. In addition to regular log files, Pulsar metrics can be used to communicate the number of retry attempts to the Pulsar Administrator.

9.2.2 Circuit Breaker

While the Retry pattern was designed to handle transient faults because it enables an application to retry an operation in the expectation that it'll succeed. The *Circuit Breaker pattern* on the other hand prevents an application from performing an operation that is likely to fail due to a non-transient fault.

The circuit breaker pattern, which was popularized by Michael Nygard in his book, [*Release It!*](#) is designed to prevent overwhelming a remote service with additional calls when we already know that it is has been unresponsive. Therefore, after a certain number of failed attempts, we will consider that the service is either unavailable or overloaded and reject all subsequent calls to the service. The intent is to prevent the remote service from becoming

further overloaded by bombarding it with requests that we already know are unlikely to succeed.

The pattern gets its name due to the fact that its operation is modelled after the physical electric circuit breakers found inside houses. When the number of faults within a given period of time exceeds a configurable threshold, the circuit breaker “trips”, and all invocations of the function that is decorated with the circuit breaker will fail immediately. The circuit breaker acts as a state machine that starts in the *closed* state, which indicates that the data can flow through the circuit breaker.

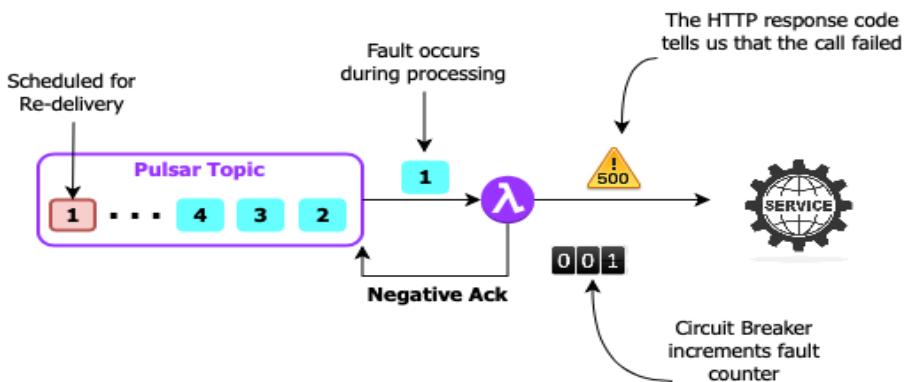


Figure 9.7: A circuit breaker starts in the *closed* state, which means that the service will be called for every message that is processed. The service call made when processing message #1 throws an exception so the fault counter is incremented, and the message is negatively acked. If the counter exceeds the configured threshold, the circuit breaker will “trip” and transition to an open state.

As we can see in Figure 9.7, all incoming messages result in a call to the remote service. However, the circuit breaker keeps track of how many calls to the service produced an exception. When the number of exceptions exceeds a configured threshold, it is an indication that the remote service is either unresponsive or too busy to process additional requests. Therefore, in order to prevent additional load on an already overloaded service, the circuit breaker transitions to the open state as shown in Figure 9.8. This is analogous to what an electrical circuit breaker does when it experiences an electrical surge and “trips” (opens) the circuit to prevent the flow of electricity into an already overloaded power outlet.

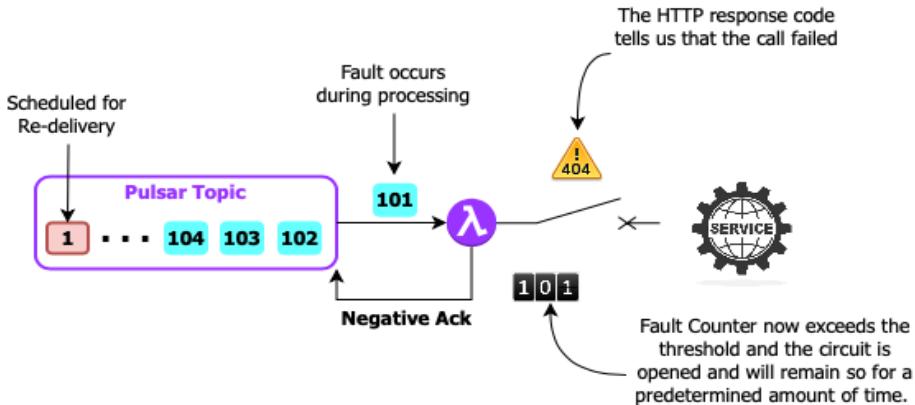


Figure 9.8: Once the circuit breaker's fault counter exceeds the configured threshold, the circuit transitions to the *open* state, and all subsequent messages are immediately negatively acked. This prevents making additional calls to a service that are likely to fail and gives the service some time to recover.

Once the circuit breaker has been tripped, it will remain so for pre-configured amount of time. No calls will be made to the service until that time period has expired. This gives the service time to fix the underlying issue before allowing the application to resume making calls to it. After the time period has elapsed the circuit transitions to the *half-open* state in which a limited number of requests are permitted to invoke the remote service.

The half-open state is intended to prevent a recovering service from being flooded with requests from all the backlogged messages. As the service recovers, its capacity to handle requests might be limited at first. Only sending a limited number of requests prevents the recovering system from being overwhelmed after it has recovered. The circuit breaker maintains a success counter that is incremented for every message that is allowed to invoke the service and completes successfully as shown in Figure 9.9.

If all of these requests are successful, as indicated by the success counter, it is assumed that the fault that was causing the previous issue has been resolved and that the service is ready to accept requests again. Thus, the circuit breaker will then switch back to the closed state and resumes normal operation. However, if any of the messages sent during the half-open state fail, the circuit breaker will immediately switch back to the open state (no failure count threshold applies) and restarts the timer to give the service additional time to recover from the fault.

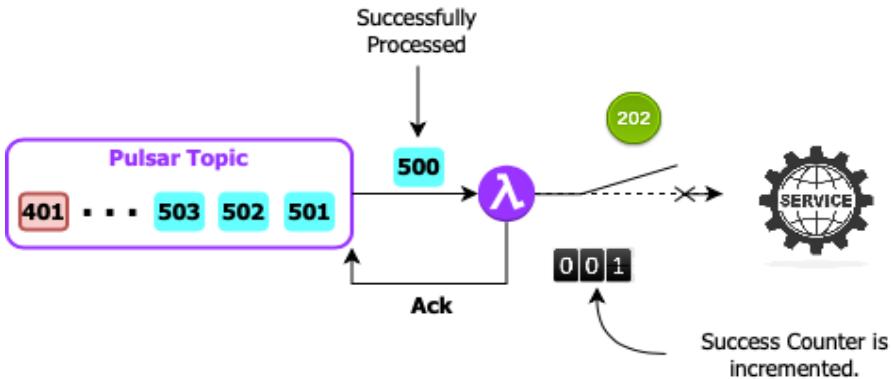


Figure 9.9: When the circuit breaker is in the *half-open* state, a limited number of messages are permitted to invoke the service. Once a sufficient number of these calls complete successfully, the circuit transitions back to the *closed* state. However, if ANY of the messages fail the circuit breaker immediately transitions back to the *open* state.

This pattern provides stability to the application while the remote service recovers from a non-transient fault by quickly rejecting a request that is likely to fail and might lead to cascading failures throughout the entire application. Often times, this pattern is combined with the Retry pattern to handle both transient and non-transient faults within the remote service.

In order to utilize the circuit breaker pattern inside your Pulsar Function, you would need to wrap the remote service call inside a *Decorator* as shown in Listing 9.3 which shows the implementation of the `CreditCardAuthorizationService` that is called from the GottaEat Payment Service when the customer uses a credit card for payment.

Listing 9.3 Utilizing the Circuit Breaker Pattern inside a Pulsar Function

```
...
import io.github.resilience4j.circuitbreaker.*;
import io.vavr.CheckedFunction0;
import io.vavr.control.Try;

public class CreditCardAuthorizationService
    implements Function<CreditCard, AuthorizedPayment> {

    public AuthorizedPayment process(CreditCard card, Context ctx)
        throws Exception {

        CircuitBreakerConfig config = CircuitBreakerConfig.custom()
            .failureRateThreshold(20)      #B
            .slowCallRateThreshold(50)     #C
            .waitDurationInOpenState(Duration.ofMillis(30000))   #D
            .slowCallDurationThreshold(Duration.ofSeconds(10))    #E
            .permittedNumberOfCallsInHalfOpenState(5)           #F
            .minimumNumberOfCalls(10)       #G
            .slidingWindowType(SlidingWindowType.TIME_BASED)    #H
            .slidingWindowSize(5)         #I
    }
}
```

```

        .ignoreException(e -> e instanceof
            UnsuccessfulCallException &&
            ((UnsuccessfulCallException)e).getCode() == 499 )      #J
        .recordExceptions(IOException.class,
            UnsuccessfulCallException.class)      #K
        .build();

CheckedFunction0<String> cbFunction =
    CircuitBreaker.decorateCheckedSupplier(
        CircuitBreakerRegistry.of(config).circuitBreaker("name"),      #L
        () -> {      #M
            OkHttpClient client = new OkHttpClient();
            StringBuilder sb = new StringBuilder()
                .append("number=").append(card.getAccountNumber())
                .append("&cvc=").append(card.getCcv())
                .append("&exp_month=").append(card.getExpMonth())
                .append("&exp_year=").append(card.getExpYear());
        });

MediaType mediaType =
MediaType.parse("application/x-www-form-urlencoded");
RequestBody body =
    RequestBody.create(sb.toString(), mediaType);

Request request = new Request.Builder()
    .url("https://noodlio-pay.p.rapidapi.com/tokens/create")
    .post(body)
    .addHeader("x-rapidapi-key", "SIGN-UP-FOR-KEY")
    .addHeader("x-rapidapi-host", "noodlio-pay.p.rapidapi.com")
    .addHeader("content-type",
        "application/x-www-form-urlencoded")
    .build();
try (Response response = client.newCall(request).execute()) {
    if (!response.isSuccessful()) {
        throw new UnsuccessfulCallException(response.code());
    }
    return getToken(response.body().string());
}
}

Try<String> result = Try.of(cbFunction);      #N
return authorize(card, result.getOrNull());      #O
}

private String getToken(String json) {
    ...
}

private AuthorizedPayment authorize(CreditCard card, String token) {
    ...
}
}

```

#A We are using classes from the circuitbreaker package.

#B The number of failures before transitioning to the open state.

#C The number of slow calls before transitioning to the open state.

#D How long to remain in the open state before transitioning to the half-open state

#E Any call that takes more than 10 sec is considered slow and added to the count.

```
#F The number of calls that can be made in the half-open state.  
#G The minimum number of calls before the failure count is applicable  
#H Using a time-based window for the failure count.  
#I Use a time window of 5 minutes before restarting failure count.  
#J List of exceptions that are not added to the failure count.  
#K List of exceptions that are added to the failure count.  
#L Use our own custom circuit breaker configuration  
#M Provide the lambda expression that will be executed and if permitted by the circuit breaker  
#N Executes the decorated function until a result is received or the retry limit is reached.  
#O Return the authorized payment or null if no response was received.
```

The first parameter passed into the `CircuitBreaker.decorateCheckedSupplier` method is a `CircuitBreaker` object based on the configuration defined earlier in the code which specifies the failure count threshold, how long to remain in the open state, and which exceptions indicate a failed method call and which ones do not, etc. Additional details on these parameters and others can be found in the [resilience4j documentation](#).

The actual logic that will be called if the circuit breaker is closed is defined inside a Lambda expression which passed in as the second parameter. As you can see, the logic inside the function definition is sends an HTTP request to a third-party credit card authorization service named [Noodlio Pay](#), which returns an authorization token that can later be used to collect payment from the credit card provided. The resulting object is a decorated function that is then passed into the `Try.of` method which handles all of the circuit breaker logic automatically for you. If the call is successful, the authorization token is extracted from the Noodlio Pay response object and returned inside the `AuthorizedPayment` object.

ISSUES AND CONSIDERATIONS

While this is a very useful pattern to use when interacting with an external system such as a web service or a database, be sure to consider the following factors when implementing it inside your Pulsar Function:

- Consider adjusting the circuit breaker's strategy based on the severity of the exception itself, as a request might fail for multiple reasons, some of which might be transient and others that are non-transient. In the presence of a non-transient error, it might make sense to open the circuit breaker immediately rather than waiting for a specific number of occurrences.
- Avoid using a single circuit breaker within a Pulsar Function that utilizes multiple independent providers. For example, the GottaEat Payment Service uses four different remote services based on the method of payment provided by the customer. If the call to the Payment Service went through a single circuit breaker, then error responses from one faulty service could trip the circuit breaker and prevent calls to the other three services that are likely to succeed.
- A circuit breaker should log all failed requests so that the underlying connectivity issues can be identified and corrected as soon as possible. In addition to regular log files, Pulsar metrics can be used to communicate the number of retry attempts to the Pulsar Administrator.
- Can be used in combination with the Retry pattern.

9.2.3 Rate Limiter

While the Circuit Breaker pattern was designed to prohibit service calls only after a pre-configured number of faults have been detected over a period of time, the Rate Limiter pattern on the other hand, prohibits service calls after a pre-configured number of calls have been made within a given period regardless of whether or not they were successful.

As the name implies, the *Rate Limiter* pattern is used to limit the frequency that a remote service can be called and is useful for situations where you want to restrict the number of calls made over a given period of time. One such example would be when we are calling a web service such as the Google API, that restricts the number of calls you can make with a “free” API key to only 60 per minute.

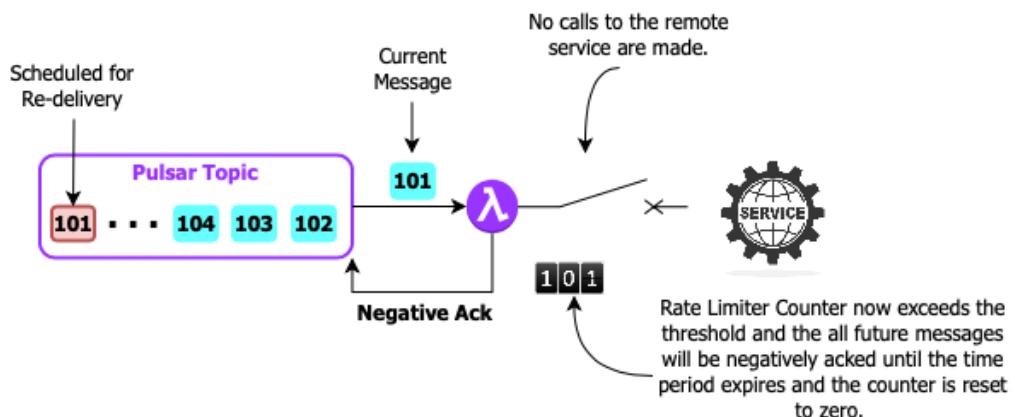


Figure 9.10: If the Rate Limiter is configured to permit 100 calls per minute, then the first 100 messages would be permitted to invoke the service. The 101st message and all subsequent messages will not be permitted to call the service and instead will be negatively acknowledged. After one-minute elapses, another 100 messages can be processed.

When using this pattern all incoming messages result in a call to the remote service up to a pre-configured number. The Rate Limiter keeps track of how many calls to the service have been made and once the limit is reached, prohibits any additional calls for the remainder of the pre-configured time window as shown in Figure 9.10. In order to utilize the rate limiter pattern inside your Pulsar Function, you would need to wrap the remote service call inside a Decorator as shown in Listing 9.4

Listing 9.4 Utilizing the Rate Limiter Pattern inside a Pulsar Function

```
import io.github.resilience4j.decorators.Decorators;
import io.github.resilience4j.ratelimiter.*;

...
public class GoogleGeoEncodingService implements Function<Address, Void> {

    public Void process(Address addr, Context ctx) throws Exception {
```

```

if (!initialized) {
    init(ctx);      #A
}

CheckedFunction0<String> decoratedFunction =
Decorators.ofCheckedSupplier(getFunction(addr))      #B
    .withRateLimiter(rateLimiter)      #C
    .decorate();

LatLon geo = getLocation(      #D
    Try.of(decoratedFunction)      #E
        .onFailure(
            (Throwable t) -> ctx.getLogger().error(t.getMessage())
        ).getOrNull());

if (geo != null) {
    addr.setGeo(geo);
    ctx.newOutputMessage(ctx.getOutputTopic(),
        AvroSchema.of(Address.class))
        .properties(ctx.getCurrentRecord().getProperties())
        .value(addr)
        .send();
} else {
    // We made a valid call, but didn't get a valid geo back
}
return null;
}

private void init(Context ctx) {
config = RateLimiterConfig.custom()
    .limitRefreshPeriod(Duration.ofMinutes(1))      #F
    .limitForPeriod(60)      #G
    .timeoutDuration(Duration.ofSeconds(1))      #H
    .build();

rateLimiterRegistry = RateLimiterRegistry.of(config);
rateLimiter = rateLimiterRegistry.rateLimiter("name");
initialized = true;
}

private CheckedFunction0<String> getFunction(Address addr) {      #I
    CheckedFunction0<String> fn = () -> {
        OkHttpClient client = new OkHttpClient();
        StringBuilder sb = new StringBuilder()
            .append("https://maps.googleapis.com/maps/api/geocode")
            .append("/json?address=")
            .append(URLEncoder.encode(addr.getStreet().toString(),
                StandardCharsets.UTF_8.toString())).append(",")
            .append(URLEncoder.encode(addr.getCity().toString(),
                StandardCharsets.UTF_8.toString())).append ","
            .append(URLEncoder.encode(addr.getState().toString(),
                StandardCharsets.UTF_8.toString()))
            .append("&key=").append("SIGN-UP-FOR-KEY");

        Request request = new Request.Builder()
            .url(sb.toString())
            .build();
    }
}

```

```

try (Response response = client.newCall(request).execute()) {
    if (response.isSuccessful()) {
        return response.body().string();      #J
    } else {
        String reason = getErrorStatus(response.body().string());
        if (NON_TRANSIENT_ERRORS.stream().anyMatch(  #K
s -> reason.contains(s))) {
            throw new NonTransientException();
        } else if (TRANSIENT_ERRORS.stream().anyMatch(
s -> reason.contains(s))) {
            throw new TransientException();
        }
        return null;
    }
}

private LatLon getLocation(String json) {      #L
...
}

```

#A Initializes the Rate Limiter

#B Decorates the REST call to the Google Maps API

#C Assigns the Rate Limiter

#D Parses the response from the REST call

#E Invokes the decorated function

#F The rate interval is one minute

#G Sets a limit of 60 calls per rate interval

#H

#I Returns a function containing the REST API call logic

#J If we got a valid response, return the JSON string with the Lat/Lon

#K Determine the error type based on the error message

#L Parses the JSON response from the Google Maps REST API call.

This function calls the Google Maps API and limits the number of attempts to 60 per minute in order to stay in compliance with Google's terms of use for a free account. If the number of calls exceeds this amount, then Google would block access to the API for our account as a preventative measure for a brief period of time. Therefore, we have taken proactive steps to prevent that condition from occurring.

ISSUES AND CONSIDERATIONS

While this is a very useful pattern to use when interacting with an external system such as a web service or a database, be sure to consider the following factors when implementing it inside your Pulsar Function:

- This pattern will almost certainly decrease the throughput for the function, so be sure to account for this throughout the entire data flow to make sure the upstream functions don't overwhelm the rate limited function.
- This pattern should be used to protect a remote service from getting overwhelmed with calls or to restrict the number of calls in order to avoid exceeding a quota which

would result you getting “locked” out by the third-party vendor.

9.2.4 Time Limiter

The *Time Limiter* pattern is used to limit the amount of time spent calling a remote service before terminating the connection from the client side. This is effectively short circuits the timeout mechanism on the remote service and allows us to determine when to give up on the remote call and terminate the connection from the client side.

This behavior is useful when you have a tight SLA on the entire data pipeline and have allocated a small amount of time for interacting with the remote service. This allows the Pulsar Function to continue processing without a response from the web service rather than waiting 30+ seconds for the connection to timeout. Doing so boosts the throughput of the Function since it no longer has to waste 30 seconds waiting on a response from a faulty service.

Consider the scenario where you first make a call to an internal caching service such as Apache Ignite to see if you have the data you need prior to make a call to an external service to retrieve the value. The purpose of doing so is to speed up the processing inside your Function by eliminating the need for a lengthy call to a remote service. However, you run the risk of your caching service itself being unresponsive which would result in a lengthy pause while making that call. This would defeat the entire purpose of the cache in the first place. Therefore, you decide to allocate a time budget to the cache call of 500 milliseconds to limit the impact an unresponsive cache can have on your Function.

Listing 9.5 Utilizing the Time Limiter Pattern inside a Pulsar Function

```
import io.github.resilience4j.timelimiter.TimeLimiter;
import io.github.resilience4j.timelimiter.TimeLimiterConfig;
import io.github.resilience4j.timelimiter.TimeLimiterRegistry;

public class LookupService implements Function<Address, Address> {
    private TimeLimiter timeLimiter;
    private IgniteCache<Address, Address> cache;
    private String bypassTopic;
    private boolean initialized = false;

    public Address process(Address addr, Context ctx) throws Exception {
        if (!initialized) {
            init(ctx);      #A
        }

        Address geoEncodedAddr = timeLimiter.executeFutureSupplier(      #B
            () -> CompletableFuture.supplyAsync(() ->
                { return cache.get(addr); }));
            #C

        if (geoEncodedAddr != null) {      #D
            ctx.newOutputMessage(bypassTopic, AvroSchema.of(Address.class))
                .properties(ctx.getCurrentRecord().getProperties())
                .value(geoEncodedAddr)
                .send();
        }
    }

    return null;
}
```

```

    }

    private void init(Context ctx) {
        bypassTopic = ctx.getUserConfigValue("bypassTopicName")
        .get().toString();      #E
        TimeLimiterConfig config = TimeLimiterConfig.custom()
            .cancelRunningFuture(true)    #F
            .timeoutDuration(Duration.ofMillis(500))    #G
            .build();

        TimeLimiterRegistry registry = TimeLimiterRegistry.of(config);
        timeLimiter = registry.timeLimiter("my-time-limiter");    #H

        IgniteConfiguration cfg = new IgniteConfiguration();      #I
        cfg.setClientMode(true);
        cfg.setPeerClassLoadingEnabled(true);

        TcpDiscoveryMulticastIpFinder ipFinder =
        new TcpDiscoveryMulticastIpFinder();

        ipFinder.setAddresses(Collections.singletonList(
        "127.0.0.1:47500..47509"));

        cfg.setDiscoverySpi(new TcpDiscoverySpi().setIpFinder(ipFinder));
        Ignite ignite = Ignition.start(cfg);    #J
        cache = ignite.getOrCreateCache("geo-encoded-addresses");    #K
    }
}

```

#A Initializes the Time Limiter and the Ignite Cache
#B Invokes the cache lookup and limits the duration of the call via the Time Limiter
#C The cache lookup that is executed asynchronously
#D If we have a cache hit then publish the value to the different output topic
#E The bypass topic is configurable
#F Cancel running calls that exceed the time limit
#G Sets a time limit of 500 milliseconds
#H Creates the Time Limiter based on the configuration
#I Configures the Apache Ignite client
#J Connects to the Apache Ignite service
#K Retrieves the local cache that stores geo-encoded addresses.

In order to utilize the time limiter pattern inside your Pulsar Function, you would need to wrap the remote service call inside a `CompletableFuture` and execute it via the `TimeLimiter's executeFutureSupplier` method as shown in Listing 9.5. This lookup function assumes that the cache is populated inside the Function that calls the Google Maps API. This allows us to restructure the Geoencoding process slightly to have the lookup occur before the call to the `GeoEncodingService` as shown in Figure 9.11

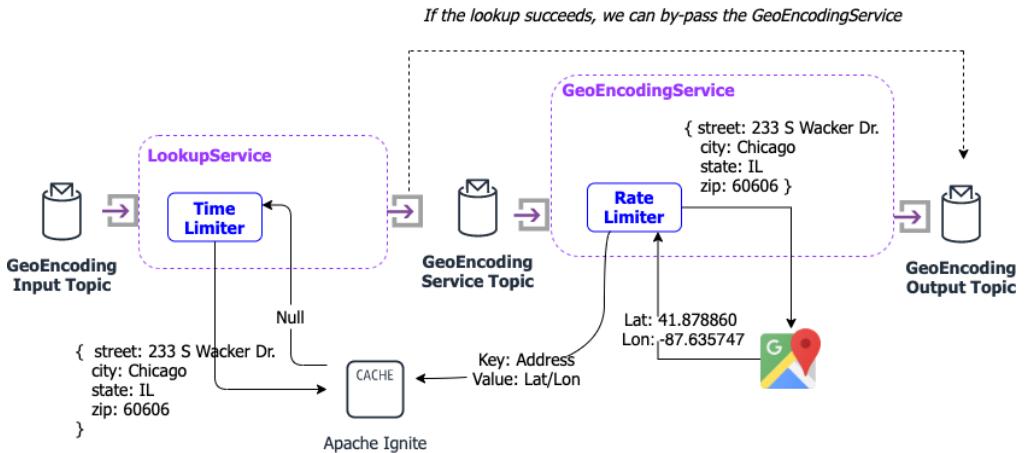


Figure 9.11: The Lookup Service can be used to prevent unnecessary calls to the GeoEncodingService if we already have the Lat/Lon pair for a given address. The cache is populated with the results from the Google Maps API calls.

Again, this type of design is intended to speed up the geo-encoding process and having and thus we need to limit the amount of time we are willing to wait to get a response back from Apache Ignite before abandoning the call. If the cache lookup succeeds, then the **LookupService** will publish a message directly to the same **GeoEncoding output topic** as the **GeoEncodingService** does. The downstream consumer doesn't care who published the message as long as it has the correct contents.

ISSUES AND CONSIDERATIONS

While this is a very useful pattern to use when interacting with an external system such as a web service or a database, be sure to consider the following factors when implementing it inside your Pulsar Function:

- Adjust the time limit to match the business SLA requirements of your application. An overly aggressive time limit will cause successful calls to be abandoned too soon. Resulting in unnecessary work on the remote service and missing data inside your Pulsar Function.
- Do not use this pattern on operations that are idempotent, otherwise you might experience unintended side effects. It is best to only use this pattern on lookup calls and other non-state changing functions or services.

9.2.5 Cache

If you would prefer NOT to write a separate function just to perform lookups, the **resiliency4j** library also provides a way to decorate a function call with a cache as well. Listing 9.6 shows how to decorate a Java lambda expression with a Cache abstraction. The cache abstraction

stores the result of every invocation of the function in a cache instance and tries to retrieve a previous cached result from the cache before it invokes the lambda expression.

Listing 9.6 Utilizing the Resiliency Cache inside a Pulsar Function

```
import javax.cache.CacheManager;
import javax.cache.Caching;
import javax.cache.configuration.MutableConfiguration;
import io.github.resilience4j.cache.Cache;
import io.github.resilience4j.decorators.Decorators;

...
public class GeoEncodingServiceWithCache implements Function<Address, Void> {

    public Void process(Address addr, Context ctx) throws Exception {

        if (!initialized) {
            init(ctx);      #A
        }

        CheckedFunction1<String, String> cachedFunction = Decorators
            .ofCheckedSupplier(getFunction(addr))    #B
            .withCache(cacheContext)    #C
            .decorate();

        LatLon geo = getLocation(    #D
        Try.of(() -> cachedFunction.apply(addr.toString())).get());    #E

        if (geo != null) {
            addr.setGeo(geo);      #F
            ctx.newOutputMessage(ctx.getOutputTopic(), AvroSchema.of(Address.class))
                .properties(ctx.getCurrentRecord().getProperties())
                .value(addr)
                .send();
        }
        return null;
    }

    private void init(Context ctx) {
        CacheManager cacheManager =
        Caching.getCachingProvider().getCacheManager();    #G

        cacheContext = Cache.of(cacheManager.createCache(    #H
        "addressCache", new MutableConfiguration<>()));
        initialized = true;
    }

    private CheckedFunction0<String> getFunction(Address addr) {    #I
        // Same as before
    }

    private LatLon getLocation(String json) {    #L
        // Same as before
    }
}

#A Initializes the Cache
#B Decorates the REST call to the Google Maps API
```

```
#C Assigns the Cache
#D Parses the response from the cache or REST call
#E Checks the cache before invoking the function.
#F If we have the Lat/Lon then send it
#G Uses the configured Cache implementation
#H Creates the address cache
#I Returns a function containing the REST API call logic
#J If we got a valid response, return the JSON string with the Lat/Lon
```

You should configure the function to use a distributed cache implementation such as Ehcache, Caffeine, or Apache Ignite. If the cache retrieval from the distributed cache fails, the exception will be ignored, and the lambda expression is called to retrieve the value instead. Please refer to your preferred vendor's documentation for more details on how to configure and use a vendor specific implementation of the JCache functional specification.

The trade-off when using this approach vs. the separate lookup service approach is that you lose the ability to limit the amount of time you are willing to wait for the cache call to complete. Therefore, in the unlikely event that the distributed cache service is down, each call could take up to 30 seconds waiting for a network timeout.

ISSUES AND CONSIDERATIONS

While this is a very useful pattern to use when interacting with an external system such as a web service or a database, be sure to consider the following factors when implementing it inside your Pulsar Function:

- This is only useful on datasets that are relatively static in nature, such as geo-encoded addresses. Don't try caching data that is likely to change frequently, otherwise you run the risk of using incorrect data inside your application.
- Limit the size of the cache to avoid causing out of memory conditions within your Pulsar Function. If you need a larger cache, you should consider using a distributed cache such as Apache Ignite.
- Prevent data in your cache from becoming stale by implementing an aging policy on the data so that it is automatically purged once it reaches a certain age.

9.2.6 Fallback Pattern

The *Fallback* pattern is used to provide your Function with an alternative resource in the event that the primary resource is unavailable. Consider the case where you are calling an internal database through a load balancer endpoint and the load balancer fails. Rather than allowing the failure of that single piece of hardware to cause your entire application to fail, you could bypass the load balancer entirely and connect directly to the database instead. Listing 9.7 shows how to implement such a function.

Listing 9.7 Utilizing the Fallback Pattern inside a Pulsar Function

```
import io.github.resilience4j.decorators.Decorators;
import io.vavr.CheckedFunction0;
import io.vavr.control.Try;

public class DatabaseLookup implements Function<String, FoodOrder> {
    private String primary = "jdbc:mysql://load-balancer:3306/food";    #A
```

```

private String backup = "jdbc:mysql://backup:3306/food";      #B
private String sql = "select * from food_orders where id=?";
private String user = "";        #C
private String pass = "";
private boolean initialized = false;

public FoodOrder process(String id, Context ctx) throws Exception {
    if (!initialized) {
        init(ctx);
    }

    CheckedFunction0<ResultSet> decoratedFunction =
        Decorators.ofCheckedSupplier( () -> {
            try (Connection con =
DriverManager.getConnection(primary, user, pass)) {      #D
                PreparedStatement stmt = con.prepareStatement(sql);
                stmt.setLong(1, Long.parseLong(id));
                return stmt.executeQuery();
            }
        })
        .withFallback(SQLException.class, ex -> {      #E
            try (Connection con =
                DriverManager.getConnection(backup, user, pass)) {
                PreparedStatement stmt = con.prepareStatement(sql);
                stmt.setLong(1, Long.parseLong(id));
                return stmt.executeQuery();
            }
        })
        .decorate();

    ResultSet rs = Try.of(decoratedFunction).get();      #F
    return ORMapping(rs);      #G
}

private void init(Context ctx) {
    Driver myDriver;
    try {
myDriver = (Driver) Class.forName("com.mysql.jdbc.Driver")
.newInstance();
        DriverManager.registerDriver(myDriver);      #H
        // Set local variables from user properties
        ...
        initialized = true;
    } catch (Throwable t) {
        t.printStackTrace();
    }
}

private FoodOrder ORMapping(ResultSet rs) {
    // Performs the Relational-to-Object mapping
}
}

```

#A The primary connection goes through the load balancer.

#B The backup connection goes directly to the database server.

#C The DB credentials, set inside the init method from user properties.

#D First call is via the load balancer.

#E If an SQLException was thrown, retry the query via the backup URL

```
#F Get the successful ResultSet  
#G Perform the ORM to return the FoodOrder object.  
#H Loads the database driver class and registers it.
```

Another scenario where this pattern could come into play would be if we had multiple third-party credit card authorization services to choose from and we want to try the alternative service in the event that we cannot connect to the primary service for some reason. Note, that the first parameter to the `withFallback` method takes the exception types that trigger the fallback code so it would be important to only contact the second credit card authorization service if the error indicated that the primary service was unavailable, and NOT if the card was declined.

ISSUES AND CONSIDERATIONS

While this is a very useful pattern to use when interacting with an external system such as a web service or a database, be sure to consider the following factors when implementing it inside your Pulsar Function:

- This is only useful in situations where you have either an alternative route to the service that is reachable from the Pulsar Function or an alternative (backup) copy of the service that provides the same function.

9.2.7 Credential Refresh Pattern

The *Credentials Refresh* pattern is used to automatically detect when your session credentials have expired and need to be refreshed. Consider the case where you are interacting with an AWS service from inside your Pulsar Function that requires session tokens for authentication. Typically, these tokens are intended to be used for short periods of time and thus have an expiration time associated with them, e.g. 60 minutes. Therefore, if you are interacting with a service that requires such a token, you need a strategy for automatically generating a new “fresh” token when the current one expires in order to keep your Pulsar Function processing messages. Listing 9.8 shows how to automatically detect an expired session token and refresh it using the Vavr functional library for Java.

Listing 9.8 Automatic Credential Refresh

```
public class PaypalAuthorizationService implements Function<PayPal, String> {  
    private String clientId;  
    private String secret;      #A  
    private String accessToken;   #B  
    private String PAYPAL_URL = "https://api.sandbox.paypal.com";  
  
    public String process(PayPal pay, Context ctx) throws Exception {  
        return Try.of(getAuthorizationFunction(pay))           #C  
            .onFailure(UnauthorizedException.class, refreshToken())    #D  
            .recover(UnauthorizedException.class,  
                (exc) -> Try.of(getAuthorizationFunction(pay)).get()       #E  
                .getOrNull();      #F  
    }  
  
    private CheckedFunction0<String> getAuthorizationFunction(PayPal pay) {  
        CheckedFunction0<String> fn = () -> {  
            OkHttpClient client = new OkHttpClient();
```

```

        MediaType mediaType =
        MediaType.parse("application/json; charset=utf-8");
        RequestBody body =
    RequestBody.create(buildRequestBody(pay), mediaType);      #G

        Request request =
new Request.Builder()
    .url("https://api.sandbox.paypal.com/v1/payments/payment")
    .addHeader("Authorization", accessToken)      #H
    .post(body)
    .build();

try (Response response = client.newCall(request).execute()) {      #I
    if (response.isSuccessful()) {
        return response.body().string();
    } else if (response.code() == 500) {
        throw new UnauthorizedException();      #J
    }
    return null;
};

        return fn;
}

private Consumer<UnauthorizedException> refreshToken() {
Consumer<UnauthorizedException> refresher = (ex) -> {
    OkHttpClient client = new OkHttpClient();
    MediaType mediaType =
    MediaType.parse("application/json; charset=utf-8");
    RequestBody body = RequestBody.create("", mediaType);

    Request request = new Request.Builder().url(PAYPAL_URL +
        "/v1/oauth2/token"?grant_type=client_credentials")      #K
        .addHeader("Accept-Language", "en_US")
        .addHeader("Authorization",
    Credentials.basic(clientId, secret))      #L
        .post(body)
        .build();

    try (Response response = client.newCall(request).execute()) {
        if (response.isSuccessful()) {
            parseToken(response.body().string());      #M
        }
    } catch (IOException e) {
        e.printStackTrace();
    };
    return refresher;
}

private void parseToken(String json) {
// Parses the new access token from the response object
}

private String buildRequestBody(PayPal pay) {
// Build the payment authorization request JSON
}
}

```

#A Static credentials used to get an access token

```

#B Local copy of the access token
#C First attempt to authorize the payment using the current access token
#D If an Unauthorized exception is raised, invoke the refreshToken function.
#E Attempt to recover from the Unauthorized exception by calling the authorize method again
#F Return the final result
#G Build the JSON request body
#H Provide the current access token value for authorization
#I Authorize the payment
#J Raise the Unauthorized exception based on the response code
#K Requesting new access token
#L Provide the static credentials when requesting a new access token
#M Parse the new access token from the JSON response

```

The key to this pattern is the wrapping the first call to the PayPal payment authorization REST API inside of a Try container type which represents a computation that may either result in an exception or return a successfully computed value. It also allows us to chain computations together which allows us to handle exceptions in a more readable manner. In the example shown in Listing 9.8, the entire try/fail/refresh token/retry logic is handled in just 5 lines of code. While you could easily implement the same logic using the more traditional try/catch logic constructs, it would be much harder to follow.

The first call wrapped in the Try construct invokes the PayPal payment authorization REST API and if it was successful, returns the JSON response from that successful call. The more interesting scenario is when that first call fails, because then the function inside the onFailure method is called if and only if the exception type is `UnauthorizedException`. That only occurs when the first call returned a status code of 500.

The function inside the onFailure method attempts to remedy the situation by refreshing the access token. Finally, the recover method is used to make another call to the PayPal payment authorization REST API now that the access token has been refreshed. Thus, if the initial failure was due to an expired session token, this code block will attempt to resolve it automatically without manual intervention or any downtime. Even the message won't have to be failed and retried at a later point, which is critical since our Function is interacting directly with a client application where response time is important.

ISSUES AND CONSIDERATIONS

While this is a very useful pattern to use when interacting with an external system such as a web service that uses expiring authorization tokens, be sure to consider the following factors when implementing it inside your Pulsar Function:

- This pattern is only applicable to token-based authentication mechanisms that provide a token refresh API that is exposed to the Pulsar Function.
- The tokens returned from the token refresh API should be stored in a secure location in order to prevent unauthorized access to the service.

9.3 Multiple Layers of Resiliency

As you may have noticed, in the previous section I only used one of these patterns at a time. But what if you want to want to more the one of the previous patterns inside your Pulsar

Function? There are situations where it would be quite useful to have your Function utilize both the Retry, Cache and Circuit Breaker patterns.

Fortunately, as I mentioned at the beginning of the chapter, you can easily decorate your remote service call with any number of these patterns from the resilience4j library to provide you with multiple layers of resiliency inside your Functions. Listing 9.9 demonstrates just how easy this is to accomplish for the GeoEncoding function.

Listing 9.9 Multiple Resiliency Patterns inside a Pulsar Function

```
public class GeoEncodingService implements Function<Address, Void> {

    private boolean initialized = false;
    private Cache<String, String> cacheContext;
    private CircuitBreakerConfig config;
    private CircuitBreakerRegistry registry;
    private CircuitBreaker circuitBreaker;
    private RetryRegistry retryRegistry;
    private Retry retry;

    public Void process(Address addr, Context ctx) throws Exception {

        if (!initialized) {
            init(ctx);
        }

        CheckedFunction1<String, String> resilientFunction = Decorators
            .ofCheckedSupplier(getFunction(addr))      #A
            .withCache(cacheContext)      #B
            .withCircuitBreaker(circuitBreaker)      #C
            .withRetry(retry)      #D
            .decorate();

        LatLon geo = getLocation(Try.of(() ->
            resilientFunction.apply(addr.toString()))).get();      #E

        if (geo != null) {
            addr.setGeo(geo);
            ctx.newOutputMessage(ctx.getOutputTopic(), AvroSchema.of(Address.class))
                .properties(ctx.getCurrentRecord().getProperties())
                .value(addr)
                .send();
        } else {
            // We made a valid call, but didn't get a valid geo back
        }

        return null;
    }

    private void init(Context ctx) {      #F
        // Configure a cache (once)
        CacheManager cacheManager = Caching.getCachingProvider()
            .getCacheManager();

        cacheContext = Cache.of(cacheManager
            .createCache("addressCache", new MutableConfiguration<>()));

        config = CircuitBreakerConfig.custom()
    }
}
```

```

...
cbRegistry = CircuitBreakerRegistry.of(config);
circuitBreaker = cbRegistry.circuitBreaker(ctx.getFunctionName());

RetryConfig retryConfig = RetryConfig.custom()
...
retryRegistry = RetryRegistry.of(retryConfig);
retry = retryRegistry.retry(ctx.getFunctionName());
initialized = true;
}
}

```

#A The Google Maps REST API call.
#B Decorated with a cache
#C Decorated with a circuit breaker
#D Decorated with a retry policy
#E Calls the decorated function to get the actual Lat/Lon.
#F Initialize the resiliency configurations as before

In this particular configuration, the cache will be checked first before the function is invoked. Any calls that fail will be retried based on the defined configuration, and if a sufficient number of calls fail then the circuit breaker will be tripped to prevent subsequent calls from being made to the remote service until sufficient time has passed to allow the service to recover.

Remember that these code-based resiliency patterns are also used in conjunction with the resiliency capabilities provided by the Pulsar Function framework such as auto-restarting K8s pods, etc. Together, these can help minimize your downtime and keep your Pulsar Function applications running smoothly.

9.4 Summary

- There are several different adverse events that can occur that can impact a Pulsar Function, and message backpressure is a good metric to use for fault detection.
- You can use the resiliency4j library to implement various common resiliency patterns within your Pulsar Functions, particularly those that interact with external systems.
- You can utilize multiple patterns inside the same Pulsar Function to increase its resiliency to failures.

10

Data Access

This chapter covers:

- Storing and retrieving data with Pulsar Functions.
- Using Pulsar's internal state mechanism for data storage and retrieval.
- Accessing data from external systems with Pulsar Functions.

Thus far all of the information used by our Pulsar Functions has been provided inside the incoming messages. While this is an effective way to exchange information, it is not the most efficient or desirable way for Pulsar Functions to exchange information with one another. The biggest drawback to this approach is that it creates a dependency on the message source to provide your Pulsar Function with the information it needs to do its job. This violates the encapsulation principle of object-oriented design which dictates that the internal logic of a Function should not be exposed to the outside world. Currently, any changes to the logic inside one Function might require changes to the upstream Function that provides the incoming messages.

Consider a use case where you are writing a Function that requires a customer's contact information including their cell phone number. Rather than passing a message containing all of that information, wouldn't it be easier to just pass the customer ID which can then be used by our Function to query the database and retrieve the information we need instead? In fact, this is a common access pattern if the information required by the Function exists in an external data source such as a database. This approach enforces encapsulation and prevents changes in one Function from directly impacting other Functions by relying on each Function to gather the information it needs instead of providing it inside the incoming message.

In this chapter I will walk through several uses cases that need to store and/or retrieve data from an external system and demonstrate how to do so using Pulsar Functions. In doing so, I will cover a variety of different data stores and describe the various criteria used to select one technology over another.

10.1 Data Sources

The GottaEat application that we have been developing thus far is hosted within the context of a much larger enterprise architecture that consists of multiple technologies and data storage platforms as shown in Figure 10.1. These data storage platforms are used by multiple applications and computing engines within the GottaEat organization and act as a single source of truth for the entire enterprise.

As you can see, Pulsar Functions can access data from a variety of data sources; from low-latency in-memory data grids and disk-backed caches to high-latency data lakes and blob storage. These data stores allow us to access information supplied by other applications and computing engines. For instance, the GottaEat application we have been developing has a dedicated MySQL database that it uses to store customer and driver information such as their login credentials, contact information, and home address. This information is collected by some of the non-pulsar microservices shown in Figure 10.1 that provide non-streaming services such as user login, account update, etc.

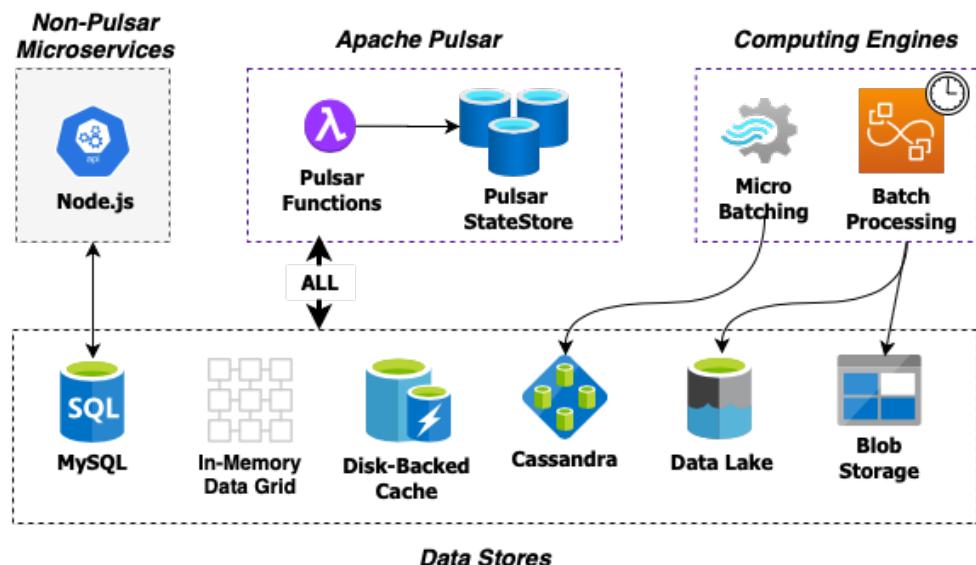


Figure 10.1: The enterprise architecture of the GottaEat organization will be comprised of various computing engines and data storage technologies. Therefore, it is critically important that we can access these various data repositories from our Pulsar Functions.

Within the GottaEat enterprise architecture there are distributed computing engines, such as Apache Spark and Apache Hive, that perform batch processing of extremely large datasets stored inside the data lake. One such dataset is the driver location data which contains the latitude and longitude coordinates of every driver that is logged into our application. The GottaEat mobile application automatically records the driver's location every 30 seconds and forwards it to Pulsar for storage in the data lake. This dataset is then used in the

development and training of the company's machine learning models. Given sufficient information, these models can accurately predict various key business metrics such as estimated delivery times of placed orders, or the areas within a city most likely to have the most food orders in the next hour so that we can position the drivers accordingly.

These computing engines are scheduled to run periodically in order to calculate various data points that are required by the machine learning models and stores them inside the Cassandra database so that they can be accessed by the Pulsar Functions that are hosting the ML models in order to provide real-time predictions.

10.2 Data Access Use Cases

In this section I will demonstrate how to access data from various data stores using Pulsar Functions. Before I jump right into the use cases, I wanted to discuss how you go about deciding which data store is best for a given use case. From a Pulsar Function perspective which data store to use will depend on a number of factors, including the availability of the data, and the maximum processing time allocated to a particular Pulsar Function. Some of our Pulsar Functions such as delivery time estimation will be directly interacting with the customer/driver and need to provide a response in near real-time. For these types of Functions, all the data access should be limited to low-latency data sources in order to ensure predictable response times.

There are four data stores within the GottaEat architecture that can be classified as low-latency due to their ability to provide sub-second data access time. These include the Pulsar State Store backed by Apache BookKeeper, an In-Memory DataGrid (IMDG) provided by Apache Ignite, a Disk-Backed Cache, and the distributed column-store provided by Apache Cassandra. The in-memory data retains all of the data in memory and thus provides the fastest access times. However, its storage capacity is limited by the amount of RAM that is available on the machines hosting the Pulsar Functions. It is used to pass information between Pulsar Functions that are not directly connected via topic, such as two-factor authentication within the device validation use case.

10.2.1 Device Validation

When a customer first downloads the GottaEat mobile application from the App Store and installs it on their phone, the customer needs to create an account. The customer chooses an email/password to use for their login credentials and submits them along with the mobile number and device id used to uniquely identify the device. As you can see from Figure 10.2, this information is then stored in the MySQL database to be used to authenticate the user whenever they login in the future.

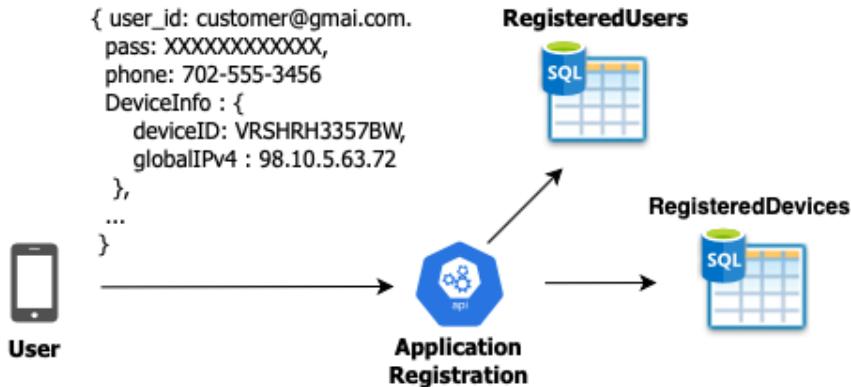


Figure 10.2: A Non-Pulsar microservice handles the application registration use case and stores the provided user credentials and device information in the MySQL database.

When a user logs into the GottaEat application we will use the information stored in the database to validate the provided credentials and authentication the device they are using to connect with. Since we recorded the device id when the user first registered, we can compare the device id provided when the user logged in with the one on record. This acts like a “cookie” does for a web browser and allows us to confirm that the user is using a trusted device.

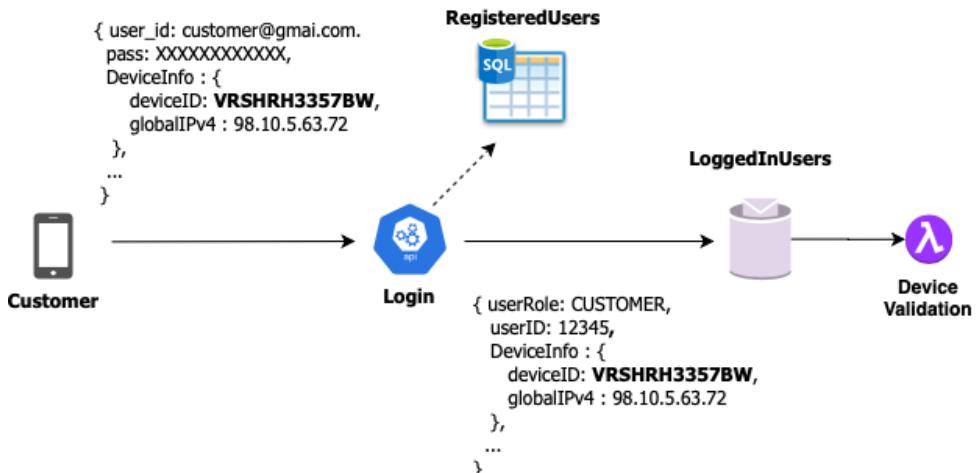


Figure 10.3: The device Id is provided when the customer logs into the mobile application, which we then cross-reference known devices that we have previously associated with the user.

As you can see from Figure 10.3, after we validate the user’s credentials against the values stored in the `RegisteredUsers` table, we place the user record in the `LoggedInUsers`

topic for further processing by the `DeviceValidation` Function to ensure that the user is using a trusted device.

This provides an additional level of security and prevents identity thieves from placing fraudulent orders using stolen customer credentials because it also requires the would-be fraudster to provide the correct device id in order to place an order. You might be asking yourself what happens if the customer is simply using a new or different device? This is a perfectly valid scenario that we need to handle, such as when they buy a new cell phone or log into their account from a different device. In such a scenario we will use SMS based two-factor authentication, in which we send a randomly-generated 5-digit PIN code to the mobile number that the customer provided when they registered and wait for them to respond back with that code. This is a very common technique used to authenticate users of mobile applications.

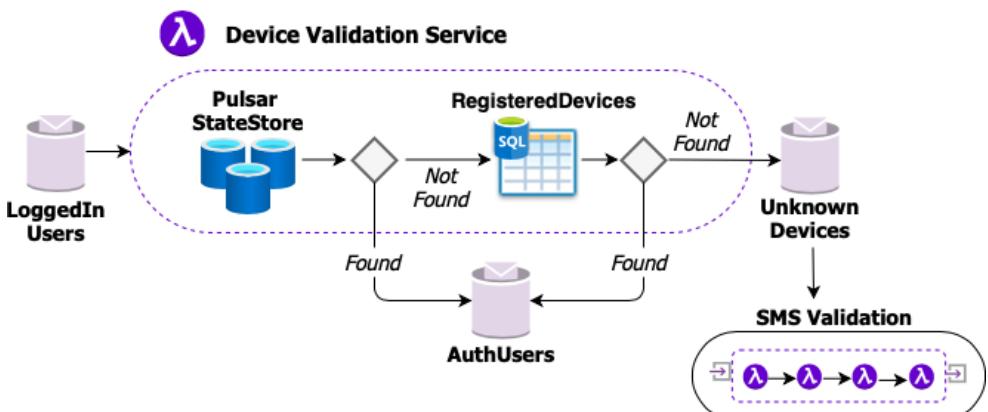


Figure 10.4: Inside the Device Validation Service, we compare the given device id with the most recently used device by the customer and if it the same we authorize the user. Otherwise, we check to see if it is the list of known devices for the user and initiate two-factor authentication if necessary.

As you can see from Figure 10.4, the device validation process first attempts to retrieve the customer's most recently used device id from Pulsar's internal State Store, which you may recall from Chapter 4 is a key-value store that can be used by Stateful Pulsar Functions to retain stateful information. The Pulsar State Store uses Apache BookKeeper for storage in order to ensure that any information written to it will be persisted to disk and replicated. This ensures that the data will outlive any associated Pulsar Function instance that is reading the data.

However, this reliance on BookKeeper for storage comes at a performance cost, since the data has to be written to disk. Therefore, even though the State Store provides the same key-value semantics as other data stores such as the In-Memory DataGrid, it does so at a much higher latency rate. Consequently, it is NOT recommended to use the State Store for data that will be read and written frequently. It is better suited for infrequent read/write use cases such as the device id, which is only called once per user session.

Listing 10.1, shows the logical flow of the `DeviceValidationService` as it attempts to locate the device id from the two different data sources. The base class contains all of the source code related to connecting to the MySQL database and is also used by the `DeviceRegistrationService` to update the database if the SMS authentication is successful.

Listing 10.1 The DeviceValidationService

```
public class DeviceValidationService extends DeviceServiceBase implements
    Function<ActiveUser, ActiveUser> {

    ...

    @Override
    public ActiveUser process(ActiveUser u, Context ctx) throws Exception {
        boolean mustRegister = false;

        if (StringUtils.isBlank(u.getDevice().getDeviceId())) {      #A
            mustRegister = true;
        }

        String prev = getPreviousDeviceId(
            u.getUser().getRegisteredUserId(), ctx);      #B

        if (StringUtils.equals(prev, EMPTY_STRING)) {
            mustRegister = true;      #C
        } else if (StringUtils.equals(prev, u.getDevice().getDeviceId())) {
            return u;      #D
        } else if (isRegisteredDevice(u.getUser().getRegisteredUserId(),
            u.getDevice().getDeviceId().toString())) {      #E
            ByteBuffer value = ByteBuffer.allocate(
                u.getDevice().getDeviceId().length());
            value.put(u.getDevice().getDeviceId().toString().getBytes());
            ctx.putState("UserDevice-" + u.getDevice().getDeviceId(), value);      #F
        }
    }

    if (mustRegister) {
        ctx.newOutputMessage(registrationTopic, Schema.AVRO(ActiveUser.class))
            .value(u)
            .sendAsync();      #G
        return null;
    } else {
        return u;
    }
}

private String getPreviousDeviceId(long registeredUserId, Context ctx) {
    ByteBuffer buf = ctx.getState("UserDevice-" + registeredUserId);      #H
    return (buf == null) ? EMPTY_STRING :
        new String(buf.asReadOnlyBuffer().array());
}

private boolean isRegisteredDevice(long userId, String deviceId) {
    try (Connection con =getDbConnection();
        PreparedStatement ps = con.prepareStatement("select count(*) "
        + "from RegisteredDevice where user_id = ? "
        + "AND device_id = ?")) {
        ps.setLong(1, userId);
    }
}
```

```

ps.setString(2, deviceId);
ResultSet rs = ps.executeQuery();
if (rs != null && rs.next()) {
    return (rs.getInt(1) > 0);      #I
}
} catch (ClassNotFoundException | SQLException e) {
// Ignore these
}
return false;
}
...
}

```

#A If the user didn't provide a device id at all.

#B Get the previous device id for this user from the state store.

#C There is no previous device id for this user

#D The provided device id matches the value in the state store

#E See if the device id is in the list of known devices for the user

#F This is a known device, so update the state store with the provided device id

#G This is an unknown device, so publish a message in order to perform SMS validation

#H Lookup the device id in the state store using the user id

#I Return true if the provided device id is associated with the user

#J Query the RegisteredDevice table by user id and device id.

If the device cannot be associated with the current user, then a message will be published to the `UnknownDevices` topic that is used to feed the SMS validation process. As you may have noticed in Figure 10.4, the SMS validation process is comprised of a sequence of Pulsar Functions that must coordinate with one another in order to perform the two-factor authentication.

The first Pulsar Function within the SMS validation process reads messages off of the `UnknownDevices` topic and uses the registered user ID to retrieve the mobile number that we will be sending the SMS validation code to. This is a very straight-forward data access pattern in which we use the primary key in order to retrieve information from a relational database and forward it to another Function. While this approach is typically considered an anti-pattern because it violates the encapsulation principle I mentioned earlier, I decided to break it out into its own Function for the two reasons. The first is reusability of the code itself, as there are multiple use cases in which we will need to retrieve all of the information about a particular user and I want to have that logic contained within a single class rather than spread across multiple Functions so that it is easy to maintain. The second reason is that we will need some of this information later on within the `DeviceRegistration` service, so I decided to pass all of the information along rather than perform the same database query twice in order to reduce processing latency.

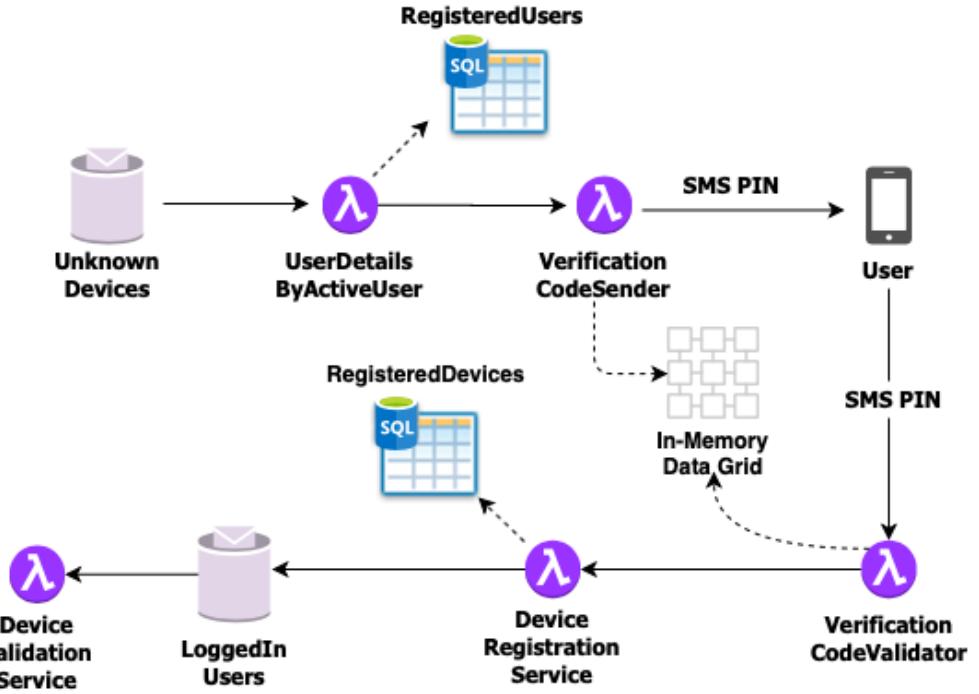


Figure 10.5: The SMS validation process is performed by a collection of Pulsar Functions that send a 5-digit code to the user's registered mobile number and validate the code sent back from the user.

As you can see from Listing 10.2, the `UserDetailsByActiveUser` Function provides the registered user ID to the `UserDetailsLookup` class which performs the database query to retrieve the data and return an `UserDetails` object.

Listing 10.2 The UserDetailsByActiveUser Function

```

public class UserDetailsByActiveUser implements
    Function<ActiveUser, UserDetails> {

    private UserDetailsLookup lookupService = new UserDetailsLookup();      #A

    @Override
    public UserDetails process(ActiveUser input, Context ctx) throws Exception{
        return lookupService.process(
            input.getUser().getRegisteredUserId(),      #B
            ctx);          #C
    }
}

```

#A Create an instance of the class that performs the database lookup

#B Pass in the primary key field used in the lookup

#C Pass in the context object so that lookup class can access the database credentials, etc.

This pattern of extracting just the primary key from the incoming message and passing it to different class that performs the lookup can be re-used for any use case, and in the case of the GottaEat application is used by a different Pulsar Function flow to retrieve the user information for a given food order as shown in Listing 10.3.

Listing 10.3 The UserDetailsByFoodOrder Function

```
public class UserDetailsByFoodOrder implements
    Function<FoodOrder, UserDetails> {

    private UserDetailsLookup lookupService = new UserDetailsLookup();      #A

    @Override
    public UserDetails process(FoodOrder order, Context ctx) throws Exception {
        return lookupService.process(
            order.getMeta().getCustomerId(),      #B
            ctx);      #C
    }
}
```

#A Create an instance of the class that performs the database lookup

#B Pass in the primary key field used in the lookup

#C Pass in the context object so that lookup class can access the database credentials, etc.

The underlying lookup class used to provide the information is shown in Listing 10.4, and encapsulates the logic required to query the database tables in order to retrieve all of the information. This data access pattern can be used to retrieve data from any relational database table that you need to access from your Pulsar Functions.

Listing 10.4 The UserDetailsLookup Function

```
public class UserDetailsLookup implements Function<Long, UserDetails> {

    ...
    private Connection con;
    private PreparedStatement stmt;
    private String dbUrl, dbUser, dbPass, dbDriverClass;

    @Override
    public UserDetails process(Long customerId, Context ctx) throws Exception {

        if (!isInitialized()) {      #A
            dbUrl = (String) ctx.getUserConfigValue(DB_URL_KEY);
            dbDriverClass = (String) ctx.getUserConfigValue(DB_DRIVER_KEY);
            dbUser = (String) ctx.getSecret(DB_USER_KEY);
            dbPass = (String) ctx.getSecret(DB_PASS_KEY);
        }

        return getUserDetails(customerId);
    }

    private UserDetails getUserDetails(Long customerId) {
        UserDetails details = null;

        try (Connection con = getDbConnection();
             PreparedStatement ps = con.prepareStatement("select ru.user_id,

```

```

+ " ru.first_name, ru.last_name, ru.email, "
+ "a.address, a.postal_code, a.phone, a.district,"
+ "c.city, c2.country from GottaEat.RegisteredUser ru "
+ "join GottaEat.Address a on a.address_id = c.address_id "
+ "join GottaEat.City c on a.city_id = c.city_id "
+ "join GottaEat.Country c2 on c.country_id = c2.country_id "
+ "where ru.user_id = ?");) {      #B

    ps.setLong(1, customerId);      #C

    try (ResultSet rs = ps.executeQuery()) {
        if (rs != null && rs.next()) {      #D
            details = UserDetails.newBuilder()
                .setEmail(rs.getString("ru.email"))
                .setFirstName(rs.getString("ru.first_name"))
                .setLastName(rs.getString("ru.last_name"))
                .setPhoneNumber(rs.getString("a.phone"))
                .setUserId(rs.getInt("ru.user_id"))
                .build();      #E
        }
    } catch (Exception ex) {
        // Ignore
    }
} catch (Exception ex) {
    // Ignore
}

return details;
}

. . .

}

```

#A Ensure we have all the required properties from the user context object.

#B Perform the lookup for the given user ID

#C Use the user ID in the first parameter in the prepared statement

#D If we found a matching record in the database

#E Map the relational data to the UserDetails object

The `UserDetailsByActiveUser` Function passes the `UserDetails` to the `VerificationCodeSender` Function which in turn sends the validation code to the customer and records the associated transaction id in the In-Memory DataGrid. This transaction id is required by the `VerificationCodeValidator` Function to validate the PIN sent back by the user, however as you can see from Figure 10.5 there isn't an intermediate Pulsar topic between the `VerificationCodeSender` Function and the `VerificationCodeValidator`, so we cannot pass this information inside a message and instead have to rely on this external data store to pass along this information.

The IMDG is perfect for this task since the information is short lived and does not need to be retained for longer the time that the code is valid. If the PIN sent back by the user is valid, the next Function will add the newly authenticated device to the `RegisteredDevices` table for future reference. Finally, the user will be sent back to the `LoggedInUser` topic so that the device id can be updated in the State Store.

Since the `VerificationCodeSender` and `VerificationCodeValidator` Functions use the IMDG to communicate with one another, I decided to have them share a common base class, shown in Listing 10.5 that provides connectivity to the shared cache.

Listing 10.5 The `VerificationCodeBase`

```
public class VerificationCodeBase {  
  
    protected static final String API_HOST = "sms-verify-api.com";      #A  
  
    protected IgniteClient client;      #B  
    protected ClientCache<Long, String> cache;      #C  
    protected String apiKey;  
    protected String datagridUrl;  
    protected String cacheName;      #D  
  
    protected boolean isInitialized() {  
        return StringUtils.isNotBlank(apiKey);  
    }  
  
    protected ClientCache<Long, String> getCache() {  
        if (cache == null) {  
            cache = getClient().getOrCreateCache(cacheName);  
        }  
        return cache;  
    }  
  
    protected IgniteClient getClient() {  
        if (client == null) {  
            ClientConfiguration cfg =  
                new ClientConfiguration().setAddresses(datagridUrl);  
            client = Ignition.startClient(cfg);      #E  
        }  
        return client;  
    }  
}
```

#A The hostname of the third-party service used to perform the SMS verification

#B Apache Ignite thin client

#C Apache Ignite Cache used to store the data

#D The name of the Cache both services will be accessing

#E Connect the thin client to the in-memory datagrid.

The `VerificationCodeSender` uses the mobile phone number from the `UserDetails` object provided by the `UserDetailsByActiveUser` Function to call the third-party SMS validation service as shown in Listing 10.6.

Listing 10.6 The `VerificationCodeSender` Function

```
public class VerificationCodeSender extends VerificationCodeBase  
    implements Function<ActiveUser, Void> {  
  
    private static final String BASE_URL = "https://" + RAPID_API_HOST + "/send-verification-  
        code";      #A  
    private static final String REQUEST_ID_HEADER = "x-amzn-requestid";
```

```

@Override
public Void process(ActiveUser input, Context ctx) throws Exception {
    if (!isInitialized()) {      #B
        apiKey = (String) ctx.getUserConfigValue(API_KEY);
        datagridUrl = ctx.getUserConfigValue(DATAGRID_KEY).toString();
        cacheName = ctx.getUserConfigValue(CACHENAME_KEY).toString();
    }

    OkHttpClient client = new OkHttpClient();
    Request request = new Request.Builder()
        .url(BASE_URL + "?phoneNumber=" + toE164FormattedNumber(
            input.getDetails().getPhoneNumber().toString())
        + "&brand=GottaEat")
        .post(EMPTY_BODY)
        .addHeader("x-rapidapi-key", apiKey)
        .addHeader("x-rapidapi-host", RAPID_API_HOST)
        .build();      #C

    Response response = client.newCall(request).execute();      #D
    if (response.isSuccessful()) {      #E
        String msg = response.message();
        String requestId = response.header(REQUEST_ID_HEADER);      #F
        if (StringUtils.isNotBlank(requestId)) {      #G
            getCache().put(input.getUser().getRegisteredUserId(), requestId);
        }
    }
    return null;
}

```

#A URL of the third-party service used to send the SMS verification code
#B Ensure we have all the required properties from the user context object.
#C Build the HTTP request object for the third-party service
#D Send the request object to the third-party service
#E If the request was successful
#F Retrieve the request ID from the response object
#G If we have a request ID, store it in the IMDG

If the HTTP call is successful, then the third-party service provides a unique request ID value in its response message that can be used to `VerificationCodeValidator` Function to validate the response sent back by the user. As you can see from Listing 10.6, we place store this request ID in the IMDG using the user ID as the key. Later, when the user responds with a PIN value, we can then use this value to authenticate the user as shown in Listing 10.7

Listing 10.7 The VerificationCodeValidator Function

```

public class VerificationCodeValidator extends VerificationCodeBase
    implements Function<SMSVerificationResponse, Void> {      #A

    public static final String VALIDATED_TOPIC_KEY = "";
    private static final String BASE_URL = "https://" + RAPID_API_HOST
        + "/check-verification-code";      #B
    private String valDeviceTopic;      #C

    @Override
    public Void process(SMSVerificationResponse input, Context ctx)
        throws Exception {

```

```

        if (!isInitialized()) {
            apiKey = (String) ctx.getUserConfigValue(API_KEY).orElse(null);
            valDeviceTopic = ctx.getUserConfigValue(VALIDATED_TOPIC_KEY).toString();
        }

        String requestID = getCache().get(input.getRegisteredUserId());      #D

        if (requestID == null) {
            return null;      #E
        }

        OkHttpClient client = new OkHttpClient();
        Request request = new Request.Builder()
            .url(BASE_URL + "?request_id=" + requestID + "&code="
                  + input.getResponseCode())
            .post(EMPTY_BODY)
            .addHeader("x-rapidapi-key", apiKey)
            .addHeader("x-rapidapi-host", RAPID_API_HOST)
            .build();      #F

        Response response = client.newCall(request).execute();
        if (response.isSuccessful()) {      #G
            ctx.newOutputMessage(valDeviceTopic,
                Schema.AVRO(SMSVerificationResponse.class))
                .value(input)
                .sendAsync();      #H
        }
    }
}

```

#A Takes in an SMS Verification Response object
#B URL of the third-party service used to perform the SMS verification
#C Pulsar topic to publish validated device info to
#D Get the request ID from the IMDG
#E If there was no request ID found for the user, then we are done
#F Construct the HTTP request to the third-party service
#G If the user responded with the correct PIN then the device was validated
#H Publish a new message to the validated device topic

This flow between the two Functions involved in the SMS validation process demonstrates how to exchange information between Pulsar Functions that are not connected by an intermediate topic. However, the biggest limitation to this approach is the amount of data that can be retained inside the memory of the datagrid itself. For larger datasets that need to be shared across Pulsar Functions a disk-backed cache is a better solution

10.2.2 Driver Location Data

Every 30 seconds a location message is sent from the driver's mobile application to a Pulsar topic. This driver location data is one of the most versatile and critical datasets at GottaEat,

not only does this information need to be stored in the data lake for training the machine learning models, but it is also useful for several real-time use cases such as in-route travel

time estimation to a pickup or drop-off location, providing driving directions, or allowing the customer to track the location of their order when it is in route to them just to name a few.

The GottaEat mobile application uses the location services provided by the phone's operating system to determine the latitude and longitude of driver's current location. This information is enriched with the current timestamp and the driver's id before it is published to the DriverLocation topic. The driver location information will be consumed in a variety of ways, which in turn dictates how the data should be stored, enriched, and sorted. One of the ways in which the location data will be accessed is directly by driver id when we want to find the most recent location of a given driver.

As you can see from Figure 10.6, the LocationTrackingService consumes these messages and uses them to update the driver location cache, which is a global cache that stores all of the location updates as key/value pairs where the key is the driver id, and the value is the location data. I have decided to store this information in a disk-backed cache so that it is accessible by any Pulsar Function that needs it. Unlike an IMDG, a disk-backed cache will spill any data that it cannot hold in memory to local disk for storage whereas an IMDG will silently drop the data. Given the anticipated volume of driver location we are expecting to have once the company is up and running, a disk-backed cache is the best choice for storing this time-critical information while ensuring that none of it is lost.

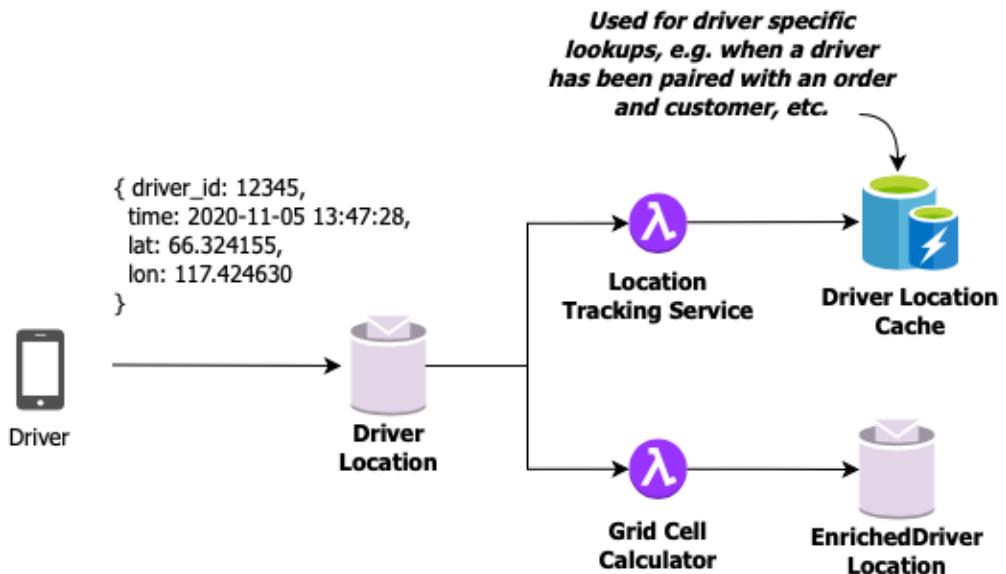


Figure 10.6: Each driver periodically sends its location information to the DriverLocation topic. This information is then processed by multiple Pulsar Functions before being persisted to a data store.

For simplicity's sake, we have decided to use the same underlying technology (Apache Ignite) to provide both the IMDG and the disk-backed cache at GottaEat. Not only does this decision reduce the number of APIs that our developers need to learn and use, but it also

simplifies life for the DevOps team as well because they only need to deploy and monitor two different clusters running the same software as shown in Figure 10.7. The only difference between the two clusters is the value of a single Boolean configuration setting called `persistenceEnabled` that enables the storage of cache data to disk, which allows it to store more data.

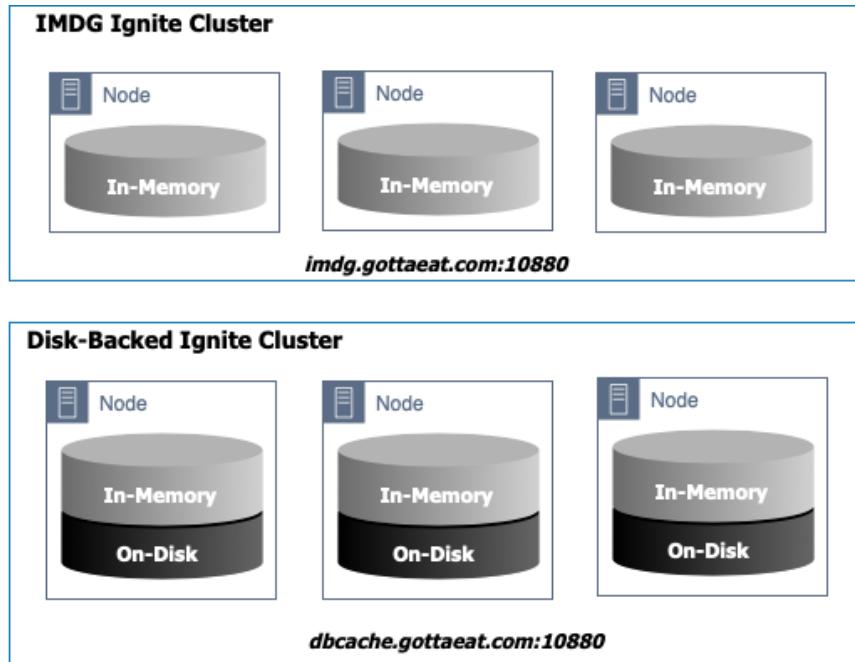


Figure 10.7: Inside the GottaEat enterprise architecture there are two different clusters running Apache Ignite, one configured to persist cache data to disk and one that isn't.

As you can see in the code in Listing 10.8, the data access for both the IMDG and disk-backed cache use the same API, i.e., you read and write values using a single key. The difference is in the expected data lookup time when retrieving the data. For an IMDG, all of the data is guaranteed to be in memory, and thus will provide a consistently fast lookup time. However, the trade-off is that the data is not guaranteed to be available when you attempt to read it.

In contrast, a disk-backed cache will retain only the most recent data in memory and store a majority of its data on disk as the data volume grows. Given that a significant amount of the data resides on disk, the overall lookup time for a disk-backed cache will be orders of magnitude slower. Therefore, it is best to this type of data store when data availability is more important than lookup times.

Listing 10.8 The LocationTrackingService

```

public class LocationTrackingService implements
    Function<ActiveUser, ActiveUser> {

    private IgniteClient client;
    private ClientCache<Long, LatLon> cache;

    private String datagridUrl;
    private String locationCacheName;

    @Override
    public ActiveUser process(ActiveUser input, Context ctx) throws Exception {

        if (!initialized()) {
            datagridUrl = ctx.getUserConfigValue(DATAGRID_KEY).toString();
            locationCacheName = ctx.getUserConfigValue(CACHENAME_KEY).toString();
        }

        getCache().put(
            input.getUser().getRegisteredUserId(),      #A
            input.getLocation());                      #B

        return input;
    }

    private ClientCache<Long, LatLon> getCache() {
        if (cache == null) {
            cache = getClient().getOrCreateCache(locationCacheName);      #C
        }
        return cache;
    }

    private IgniteClient getClient() {      #D
        if (client == null) {
            ClientConfiguration cfg =
                new ClientConfiguration().setAddresses(datagridUrl);
            client = Ignition.startClient(cfg);
        }
        return client;
    }
}

```

#A Use the userID as the key
#B Add the location to the disk-backed cache
#C Creates the location cache
#D Using an Apache Ignite client

The location information stored in the disk-backed cache can then be used to determine the current location of the specific driver assigned to the customer's order and fed to the mapping software on the customer's phone so that they can track the status of their order visually as shown in Figure 10.8.

Another use of the driver location data is determining which drivers are best suited to assign an order to. One of the factors in making this determination include the driver's proximity to the customer and/or restaurant that is fulfilling the order. While it is theoretically possible, it would be too time-consuming to calculate the distance between every driver and the delivery location each time an order is placed.

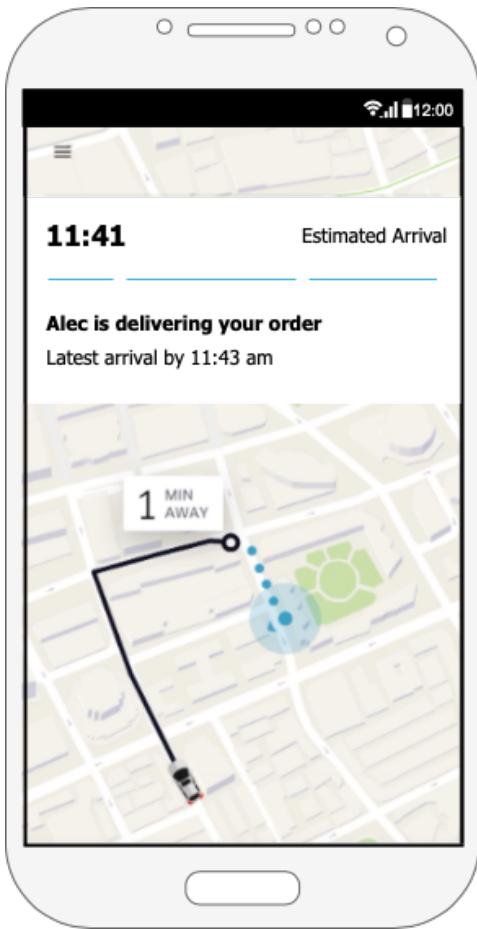


Figure 10.8: The driver location data is used to update the map display on the customer's mobile phone so they can see the driver's current location along with an estimated delivery time.

Therefore, we need a way to immediately identify drivers that are “close” to an order in near real time. One way to do this is to divide the map into smaller logical hexagonal areas known as **cells** as shown in Figure 10.9, and then group the drivers together based upon which of these **cells** that they are currently located in. This way, when an order comes in all we need to do is determine which **cell** the order is in and look for drivers that are currently in the same **cell** first. If no drivers are found, we can expand our search to adjacent **cells**. By pre-sorting the drivers in this manner, we are able to perform a much more efficient search.

The other Pulsar Function in Figure 10.6 is the `GridCellCalculator` which uses the [H3 open-source project](#) developed by Uber to determine which **cell** the driver is currently located in and appends the corresponding cell ID to the location data before publishing it to

the `EnrichedDriverLocation` topic for further processing. I left the code for that function out of this chapter, but you can find it in the GitHub repo associated with this book if you want more details.

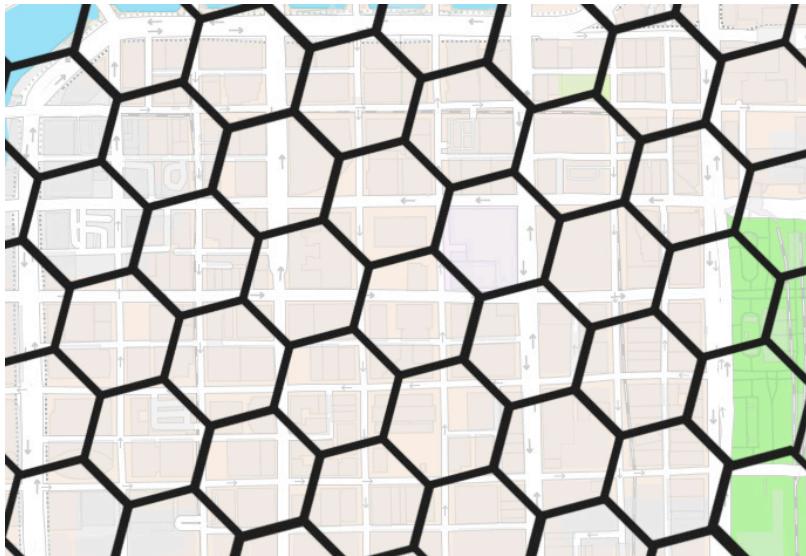


Figure 10.9: We use a global grid system of hexagonal areas that is overlaid onto a two-dimensional map. Each latitude, longitude pair maps to exactly one cell on the grid. Drivers are then bucketed together based on the hexagons they are currently located in.

As you can see from Figure 10.10 there is a global collection of disk-backed caches (one for each cell ID) used to retain the location data for drivers within a given cell ID. Pre-sorting the drivers in this manner allows us to narrow our search to only the few **cells** we are interested in and ignore all the rest. In addition to speeding up driver assignment decisions, bucketing the drivers together by cell ID enables us to analyze geographic information in order to set dynamic prices and make other decisions on a city-wide level such as surge pricing and incentivizing our drivers to move into cells without sufficient drivers to accommodate the order volume, etc.

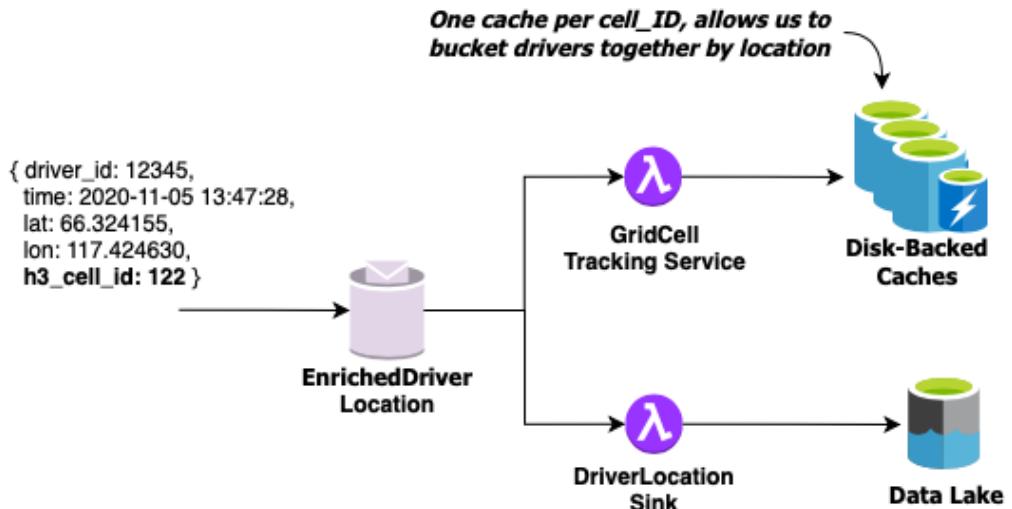


Figure 10.10: Messages inside the EnrichedDriverLocation topic have been enriched with the H3 cell ID, which is then used to group drivers together.

As you can see from the code in Listing 10.9, the `GridCellTrackingService` uses the cell ID from each incoming message to determine which cache to publish the location data into. The disk-backed caches use the following naming convention; “drivers-cell-XXX”, where the XXX is the H3 cell ID. Therefore, when we receive a message with a cell ID of 122, we place it inside the drivers-cell-122 cache.

Listing 10.9 The GridCellTrackingService

```

import com.gottaeat.domain.driver.DriverGridLocation;
import com.gottaeat.domain.geography.LatLon;

public class GridTrackingService
    implements Function<DriverGridLocation, Void> {

    static final String DATAGRID_KEY = "datagridUrl";

    private IgniteClient client;
    private String datagridUrl;

    @Override
    public Void process(DriverGridLocation in, Context ctx) throws Exception {
        if (!initialized()) {
            datagridUrl = ctx.getUserConfigValue(DATAGRID_KEY).toString();
        }

        getCache(input.getCellId())      #A
            .put(input.getDriverLocation().getDriverId(),      #B
                input.getDriverLocation().getLocation());      #C
        return null;
    }
}

```

```

private ClientCache<Long, LatLon> getCache(int cellID) {
    return getClient().getOrCreateCache("drivers-cell-" + cellID);      #D
}

private IgniteClient getClient() {
if (client == null) {
    ClientConfiguration cfg =
        new ClientConfiguration().setAddresses(datagridUrl);
    client = Ignition.startClient(cfg);
}
    return client;
}

private boolean initialized() {
    return StringUtils.isNotBlank(datagridUrl);
}

```

#A Get the cache for the given cell ID we calculated
#B Use the userID as the key
#C Add the location to the disk-backed cache
#D Return or create the cache for the specified cellID.

The other consumer of the enriched location messages is a Pulsar IO connector named `DriverLocationsSink` that batches up the messages and writes them into the data lake. In this particular case, the final destination of the sink is the HDFS filesystem which allows other teams within the organization to perform analysis on the data using various computing engines such as Hive, Storm, or Spark.

Since the incoming data is of higher value to our delivery time estimation data models, the sink can be configured to write the data to a special directory inside HDFS. This allows us to pre-filter the most recent data and group it together for faster processing by the process that uses this data to pre-compute data for the delivery time feature vector in the Cassandra database.

10.3 Summary

- Pulsar's internal state store provides a convenient location for storing infrequently accessed data without having to rely on an external system.
- You can access a variety of external data sources from inside Pulsar Functions, including in-memory data grids, disk-backed caches, relational databases, and many others.
- Consider the latency and data storage capabilities when determining the data storage system that you want to use. Lower-latency systems are typically better for stream processing systems.

11

Machine Learning in Pulsar

This chapter covers:

- Exploring how Pulsar Functions can be used to provide near real-time machine learning
- Developing and maintaining the collection of inputs required by the machine learning model to provide a prediction
- Executing any PMML supported model inside a Pulsar Function
- Executing non-PMML models inside a Pulsar Function

One of the primary goals of machine learning is the extraction of actionable insights from raw data. Having insights that are actionable means that you can use them to make strategic, data-driven decisions that result in a positive outcome for your business and customers. For instance, every time a customer places an order on the GottaEat application, we want to be able to provide the customer with an accurate estimated delivery time. To accomplish this, we would need to develop a machine learning model that predicts the delivery time of any given order based on a number of factors that allow us to make a more accurate decision.

Typically, these actionable insights are generated by machine learning (ML) models that have been developed and trained to take in a pre-defined set of inputs known as a **feature vector** and uses that information make predictions such as when an order will arrive at the customer's location. It is the responsibility of the data science team to develop and train these models, including the feature vector definitions. Since this is a book about Apache Pulsar and not data science, I will focus primarily on the deployment of these machine learning models inside the Pulsar Function framework rather than the development and training of the models themselves.

11.1 Deploying Machine Learning Models

Machine learning models are developed in a variety of languages and toolkits, each with their own respective strengths and weaknesses. Regardless of how these ML models are developed and trained, they must eventually be deployed to a production environment in order to provide real business value. At a high level, there are two execution modes for machine learning models that are deployed to production: batch-processing mode and near real-time processing.

11.1.1 Batch Processing

As the name suggests, executing a model in batch processing mode refers to the process where you feed a large batch data into your model to produce multiple predictions at the same time rather than on a case-by-case basis. These batches of predictions can then be cached and reused as needed.

One such example are marketing emails from an ecommerce site that provides a list of product recommendations based upon your purchase history with the retailer and what similar customers have purchased, etc. Since these recommendations are based upon a historical and slow-changing dataset, i.e., your purchase history, they can be generated at any time. Typically, these marketing emails are generated once per day and delivered to customers based on the customers' local time zones. Since these recommendations are generated for all customers who haven't opted-out, this is a great candidate for batch processing with the ML model. However, if your recommendations need to factor in the users' most recent activities, such as the current contents of their shopping cart, then you cannot execute the ML model in batch mode because that data won't be available. In such a scenario, you will want to deploy your ML model for near real-time processing mode.

11.1.2 Near Real-Time

Utilizing a machine learning model to generate predictions on a case-by-case basis using data that is only available in the incoming payload is commonly referred to as near real-time processing. This type of processing is considerably more complex than batch processing primarily due to the latency constraints placed on systems that need to serve the predictions.

A perfect candidate for near real-time deployment is the estimated time-to-delivery component of the GottaEat food delivery service that provides an estimate of when the food will be delivered for every newly placed food order. Not only does the feature vector for this machine learning model depend upon data provided as part of the order, i.e., the delivery address, but it also needs to generate the estimate within a few hundred milliseconds. In short, when deciding upon which of these execution modes to use for your ML models you should consider the following factors:

- The availability of all the data required in the model's feature vector, and where that is stored. Whether all the data you need to execute the model is readily available in the system or if some of it is provided in the request itself.
- How quickly the accuracy of the recommendations degrades over time. Can you pre-compute the recommendation and re-use it for a specific period or not?

- How quickly you can retrieve all the data used in the feature vector. Is it stored in memory or on non-real-time systems such as HDFS, traditional databases, etc.?

As you might have guessed, Pulsar Functions are great candidate for deploying your machine learning models in near real-time mode for a multitude of reasons. First and foremost, they get executed immediately when a request comes in and have direct access to the data provided as part of the request, which eliminates any processing and data lookup latency. Secondly, as we saw in the previous chapter, you can retrieve data from a variety of external data sources to populate the feature vector with any necessary data that exists outside of the request itself. Last, and most importantly is the flexibility Pulsar Functions provides when it comes to deployment options. The ability to write your Functions in Java, Python, or Go allows your data science team to develop the models in their language/framework of their choice and still deploy it inside a Pulsar Function using third-party libraries that are bundled with the Function to execute the model at runtime.

11.2 Near Real-Time Model Deployment

Because there are so many machine learning frameworks and everything is evolving so quickly, I have decided present a generic solution that provides all the essential elements required to deploy an ML model within a Pulsar Function. Pulsar Function's broad programming language support enables you to deploy ML models from a variety of frameworks provided there is a third-party library that supports the execution of those models. For instance, the existence of a Java library for TensorFlow, allows you to deploy and execute any ML models that were developed using the TensorFlow toolkit. Similarly, if your ML model relies on the Panda's library written in Python, then you can easily write a Python-based Pulsar Function that includes the Pandas library, etc. Regardless of the language of choice the deployment of a near real-time model within a Pulsar Function follows the same high-level pattern.

As you can see from Figure 11.1, there are a sequence of one or more Pulsar Function's that are responsible for the collection of data required by the feature vector of the model from external data sources based on the original message. This acts as a sort of data enrichment pipeline that augments the incoming data from the message with additional data elements that are needed by the ML model. In the delivery time estimation example, we might only be given the id of the restaurant that is preparing the order. This pipeline would then be responsible for retrieving the geographical location of that restaurant so that it can be given to the model.

Once this data is collected, it is fed into the Pulsar Function that will invoke the ML model using a third-party library to execute the model with the proper framework, e.g., TensorFlow, etc. It is worth noting that the ML model itself is retrieved from the Pulsar Function context object, which allows us to deploy the model independently of the Function itself. This decoupling of the model from the packaged and deployed Pulsar Function provides us the opportunity to dynamically adjust our models on the fly, and more importantly allows us to change models based on external factors and conditions.

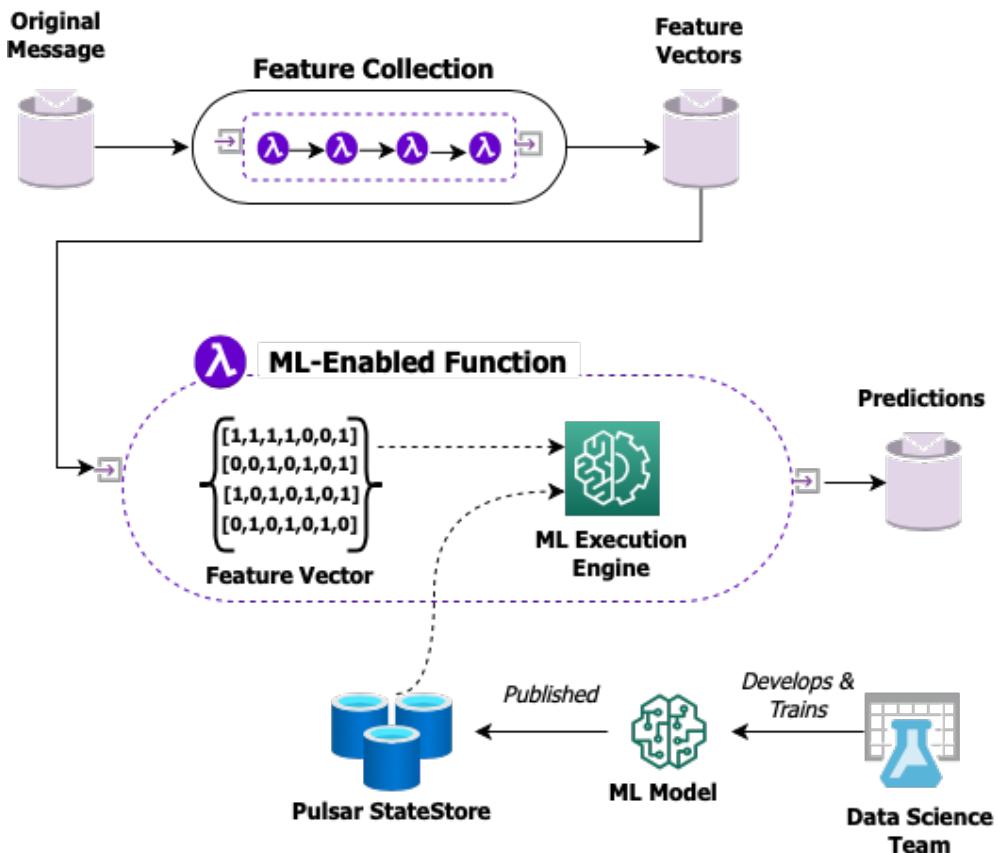


Figure 11.1: The original message that necessitates the need for a prediction is enriched with a sequence of ancillary Pulsar Functions to produce a feature vector suited for the ML model. The ML-Enabled Function can then further populate the feature vector with data from other low-latency data sources before sending it along with the ML model it retrieves from Pulsar's internal state store to the ML execution engine for processing.

For example, we might require an entirely different delivery time estimation model for high volume periods such as lunch and dinner vs. non-peak hours. An external process would then be used to rotate in the “proper” model to be used for the current time. In fact, the data science team could train the same model with time-specific datasets to produce different models based on the time of day or the day of the week, etc. For instance, training the time-estimation model with only data from 4:00-5:00pm could produce a model for that specific timeframe.

The ML-enabled Function then sends the completed Feature Vector along with the correct version of the ML model that it retrieved from Pulsar’s internal state store to the ML execution engine (third-party library, etc.) which produces both a prediction and an associated level of confidence in the prediction. For instance, a prediction of an ETA of 7

minutes with an 84% degree of confidence. This information can then be used to provide an estimated time of arrival to the customer that placed the order.

While your Pulsar Function might differ based upon the ML framework you are using, the general outline remains the same. First, get the ML model from Pulsar's state store, and keep a copy in memory. Next, take the incoming pre-computed features from the incoming message and map them to the expected input fields of the feature vector. Compute and/or retrieve any additional features that weren't already pre-computed before calling the ML library for your programming language to execute the model with the given dataset and publishing the result.

11.3 Feature Vectors

In machine learning, the term feature is the term refers to an individual numeric or symbolic property that represents a unique aspect of an object. These features are often combined into an n-dimensional vector of numerical features, aka **feature vector**, that are then used as an input to a predictive model.

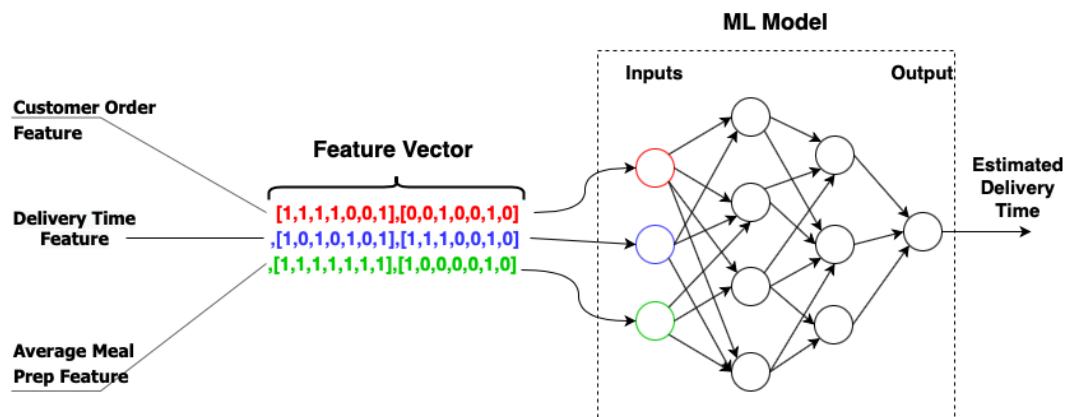


Figure 11.2: The delivery time estimation model requires a feature vector comprised of features from multiple sources. Each feature is an array of numerical values between 0 and 1.

Every feature in the feature vector represents an individual piece of data that is used to generate the prediction. Consider the estimated time-to-delivery feature of the GottaEat food delivery service. Every time a user orders food from a restaurant, a machine learning model estimates when the food will be delivered. Features for the model include information from the request (e.g., time of day, delivery location), as well as historical features (e.g., average meal prep time for the last seven days), and near-real time calculated features (e.g., average meal prep time for the last one hour). These features are then fed into the ML model to produce a prediction as shown in Figure 11.2.

Many algorithms in machine learning require a numerical representation of features, since such representations facilitate processing and statistical analysis such as linear regression,

etc. Therefore, it is common that each feature is represented as a numerical value as it makes them useful across various ML algorithms.

11.3.1 Feature Stores

Models intended to be deployed in near real-time mode have stringent latency requirements cannot reliably expect to compute features that require access to traditional data stores in a performant manner. For instance, it is not feasible to directly query a database to compute the average meal prep time for a given restaurant over a specific period with a sub-second response time consistently.

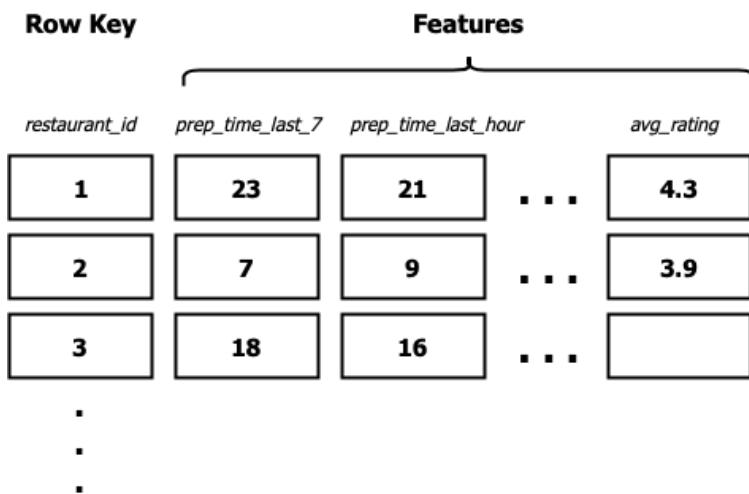


Figure 11.3: The restaurant feature store uses the unique `restaurant_id` field as the row key, and each row contains several features for each restaurant. Some of the features are specific to the delivery time estimation model while others are not.

Therefore, it is necessary to establish ancillary processes to precompute and index these necessary model features. Once computed, these features should be stored in a low latency datastore known as a **Feature Store** where they can be accessed quickly at prediction time.

As I mentioned earlier, our feature store is kept inside a low-latency column-centric database such as Apache Cassandra, which are designed to permit a large number of columns per row without incurring a performance penalty during the data retrieval process. This allows us to store features required by various ML models in the same table, e.g., delivery time estimation fields such as average prep time can be stored alongside features used by the restaurant recommendation model such as the average customer rating. Consequently, we don't need to maintain a separate table for each model we want to use, e.g., `restaurant_time_delivery_features`, etc. Not only does this simplify the management of the feature store database but it also promotes the usage of features across ML models.

In most organization the design of the tables within the feature store falls squarely on the data science team since they are the primary consumers of the feature store. Therefore, you will often only be provided the ML model along with a list of the features it requires as input when your company is ready to roll out the model to production.

11.3.2 Feature Calculation

Typically, features are associated with an entity type, e.g., restaurant, driver, customer, etc. because each feature represents a “has a(n)” relationship with each particular entity. For instance, a restaurant has an average meal preparation time of 23 minutes, or a driver has an average customer review of 4.3 stars, etc. The feature store is then populated via an ancillary process that precomputes the various features such as the average meal prep time for the last seven days for every restaurant, or the average delivery time in the city for the last one hour.

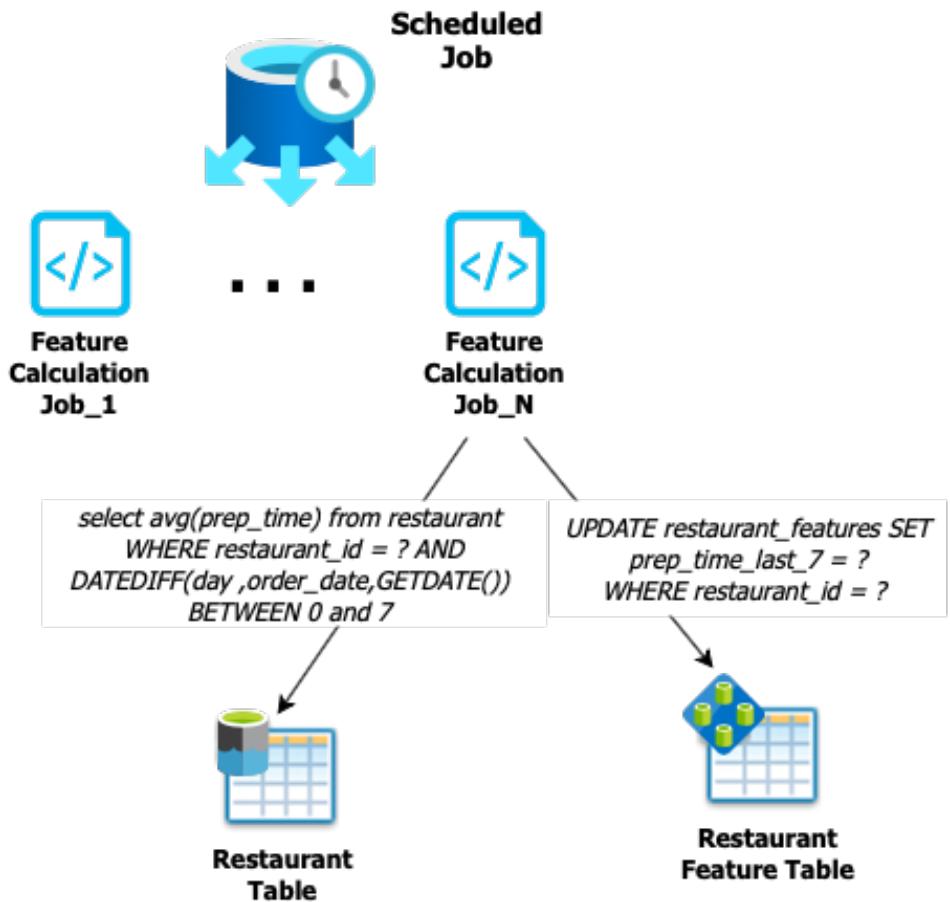


Figure 11.4: Feature calculation jobs will need to run periodically to update the features with new values based on the most recent data. A batch processing engine such as Apache Hive or Spark allows you to process many of these jobs in parallel.

These processes need to be scheduled to run periodically to cache the results in the feature store to ensure that these features can be retrieved with sub-second response time when the order arrives. A batch-oriented processing engine is best suited for performing the pre-calculation of the features as it can process a large volume of data efficiently. For instance, a distributed query engine such as Apache Hive or Spark can execute multiple concurrent queries against a large dataset stored on HDFS using standard SQL syntax as shown in Figure 11.4. These jobs can asynchronously compute the average meal prep time for all restaurants over a specific period and can be scheduled to run hourly to keep these features up to date.

As you can see from Figure 11.4, a schedule job can kick off multiple concurrent tasks to complete the work in a timely manner. For example, the scheduled job could first query the

restaurant table itself to determine the exact number of restaurant ids and then split the work by providing each instance of the restaurant feature calculation job with a sub-set of restaurant ids. This divide-and-conquer strategy will speed up the process considerably, allowing you to complete the work in a timely manner.

It is also important that you coordinate with the proper team(s) to develop, deploy, and monitor these feature calculation jobs, as they are now business critical applications that need to continuously work or else the predictions made by your ML models will degrade if the feature data is not updated in a timely manner. Using stale data in your ML models will result in inaccurate predictions, which can lead to poor business outcomes and unsatisfied customers.

11.4 Delivery Time Estimation

Now that we have sufficiently covered the background and theory, it is time to see how to deploy a ML model that has been developed and trained by the data science team at GottaEat. A perfect candidate would be the delivery time estimation ML model we have been discussing throughout the chapter. As with any ML model that needs to be deployed to production, the data science team is responsible for providing us with a copy of the trained model to be used.

11.4.1 Machine Learning Model Export

Just like most data science teams, the data science team at GottaEat uses a variety of languages and frameworks to develop their machine learning models. One of the biggest challenges is finding a single execution framework that supports the models developed in different languages.

Fortunately, there several projects seeking to standardize ML model formats to support separate languages for model training and model deployment. Projects such as the Predictive Model Markup Language (PMML) allow data scientists and engineers to export ML Models developed from a variety of languages into a language-neutral XML format.

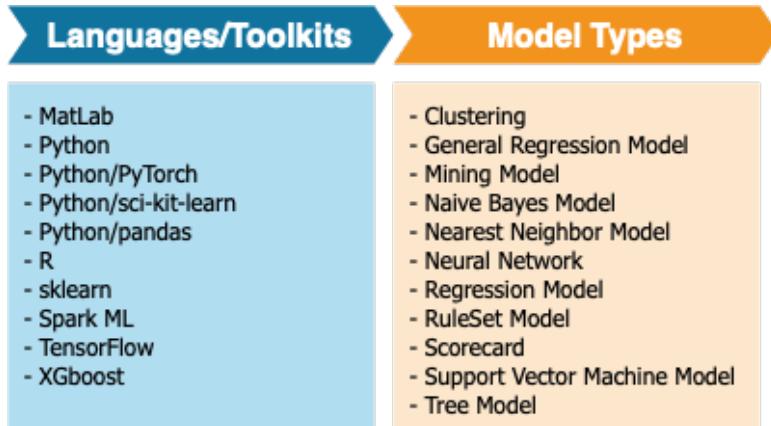


Figure 11.5: List of programming languages, toolkits, and machine learning model types supported by the Predictive Model Markup Language (PMML). Any supported models developed in these languages can be exported to PMML and executed inside a Pulsar Function.

As you can see from Figure 11.5, the PMML format can be used to represent models developed in a variety of machine learning languages. Consequently, this approach can be used on any of the supported model types that was developed in one of the supported languages and/or frameworks.

In the case of the delivery time estimation model, the data science team used the R programming language to develop and train this model. However, since there isn't direct support for running R code inside a Pulsar Functions, the data science team must first translate their R-based machine learning model into the PMML format which can easily be accomplished using the r2pmml toolkit as shown in Listing 11.1.

Listing 11.1 Exporting an R-based Model to PMML

```
// Model development code
dte <-(distance ~ ., data = df)      #A
library(r2pmml)      #B
r2pmml(dte, "delivery-time-estimation.pmml");    #C
```

#A Finalize the development of the machine learning model
#B Import the library that translates the R model into PMML
#C Perform the translation from R to PMML

The r2pmml library does a direct translation of the R-based model object into the PMML format and saves it to local file named `delivery-time-estimation.pmml`. The PMML file format specifies the data fields to use for the model, the type of calculation to perform (regression), and the structure of the model. In this case, the structure of the model is a set of coefficients, which is defined as shown in Listing 11.2. We now have a model specification

that we are ready to productize and apply to our production data set to generate delivery time predictions.

Listing 11.2 The Delivery Time Estimation Model in PMML Format.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<PMML version="4.2" xmlns="http://www.dmg.org/PMML-4_2">
<Header description="deliver time estimation">
    <Application name="R" version="4.0.3"/>
    <Timestamp>2021-01-18T15:37:26</Timestamp>
</Header>
<DataDictionary numberOfFields="4">
    <DataField name="distance" optype="continuous" dataType="double"/>
    <DataField name="prep_last_hour" optype="continuous" dataType="double"/>
    <DataField name="prep_last_7" optype="continuous" dataType="double"/>
    ...
</DataDictionary>
<RegressionModel functionName="regression">
    <MiningSchema>
        <MiningField name="distance"/>
        ...
    </MiningSchema>
    ...
    <NumericPredictor name="travelTime" coefficient="7.6683E-4"/>
    <NumericPredictor name="avgPreptime" coefficient="-2.0459"/>
    <NumericPredictor name="avgDeliveryTime" coefficient="9.4778E-5"/>
    ...
</RegressionModel>
</PMML>
```

Once the trained model has been exported to PMML, a copy of it should be published to the Pulsar state store so that the Pulsar Function can access it at runtime. This can be accomplished by using the pulsar-admin putstate command as shown in Listing 11.3, which uploads the given file to specified namespace inside Pulsar's state store. It is critically important that the namespace be the same as the one you will be using to deploy the Pulsar Function that will be using the model for it to have read access.

Listing 11.3 Upload Machine Learning Model to Pulsar's State Store

```
./bin/pulsar-admin functions putstate \      #A
--name MyMLEnabledFunction \
--namespace MyNamespace \
--tenant MyTenant \   #B
--state "{\"key\":\"ML-Model\", \
\"byteValue\": <contents of delivery-time-estimation.pmml >}"\     #C
```

#A Using the pulsar-admin CLI tool to upload the PMML file

#B You need to specify the correct tenant, namespace, and function name

#C Upload the contents of the generated PMML file

Having the ability to automatically push model changes to production fits nicely within the emerging field of Machine Learning Operations (ML Ops), which applies the agile principles of continuous integration, delivery, and deployment to the machine learning process in order to accelerate and simplify ML model deployment. In this case, we use a script or CI/CD pipeline tool to check out the latest version of a model from source control and upload it to Pulsar

using a simple shell script, etc. Additionally, ancillary jobs can be scheduled to rotate different versions of the same model automatically based on the time of day, etc.

11.4.2 Feature Vector Mapping

Another thing the data science team must provide us with is a complete definition of the feature vector we need to provide to the ML model, along with a map of where these features reside inside the Feature Store(s) so that we can retrieve these values at runtime and provide them to the model.

The development team at GottaEat decided that it would be easier to store the feature vector mapping information inside a Protobuf object because of its built-in support for associated maps in the protocol. This allows us to store the data in the correct format, and easily serialize/deserialize the data into a format that is compatible with the Pulsar state store, i.e., a byte array. The Protobuf protocol is also language neutral, so it can be used by any of the programming languages supported by Pulsar Functions, including Java, Python, and Go. The definition of the Protobuf object used to store the feature mapping information is shown in Listing 11.4 and contains three elements; the name of the table to query inside the Feature Store, a list of all the features stored in the specified table that are required by the ML model, and an associated map that defines the mapping between the features in the Feature Store and the fields in the Feature Vector sent to the model.

Listing 11.4 The Feature Vector Mapping Protocol

```
syntax = "proto3";  
  
message FeatureVectorMapping {  
    string featureStoreTable = 1;      #A  
    repeated string featureStoreFields = 2;    #B  
    map<string, string> fieldMapping = 3;    #C  
}
```

#A The name of the table inside the feature store that contains the fields specified

#B A list of all of the fields we need to retrieve from the Feature Store table

#C A mapping of each field name to its corresponding feature name inside the model's Feature Vector

The list of features is used to dynamically construct the SQL query used to retrieve the features to make the query as efficient as possible by returning only those values that we actually need instead of the entire row. The list also serves as a complete list of all the keys in the map data structure as well, which allows us to iterate over the list and retrieve all the feature-to-vector mappings contained in the map.

Once this information is captured inside a Protobuf object, it should be published to the Pulsar state store using the `putstate` command. However, the Pulsar Function name will be that of the Pulsar Function inside the feature collection pipeline that will perform the feature lookup for the deployed model. In the case of the delivery time estimation model, the feature extraction pipeline contains a Pulsar Function named `RestaurantFeaturesLookup` that queries the `Restaurant` table of the feature store using the id of the restaurant that will be preparing the food for the customer.

Listing 11.5 The Restaurant Feature Lookup Function

```
public class RestaurantFeaturesLookup implements
    Function<FoodOrder, RestaurantFeatures> {

    private CqlSession session;
    private InetSocketAddress node;
    private InetSocketAddress address;
    private SimpleStatement queryStatement;

    @Override
    public RestaurantFeatures process(FoodOrder input, Context ctx)
        throws Exception {
        if (!initialized()) {
            hostName = ctx.getUserConfigValue(HOSTNAME_KEY).toString();      #A
            port = (int) ctx.getUserConfigValueOrDefault(PORT_KEY, 9042);    #B
            dbUser = ctx.getSecret(DB_USER_KEY);
            dbPass = ctx.getSecret(DB_PASS_KEY);      #C
            table = ctx.getUserConfigValue(TABLE_NAME_KEY);      #D
            fields = new String(ctx.getState(FIELDS_KEY));      #E
            sql = "select " + fields + " from " + table +
                " where restaurant_id = ?"      #F
            queryStatement = SimpleStatement.newInstance(sql);      #G
        }
        return getRestaurantFeatures(input.getMeta().getRestaurantId());
    }

    private RestaurantFeatures getRestaurantFeatures (Long id) {
        ResultSet rs = executeStatement(id);      #H

        Row row = rs.one(); #I
        if (row != null) {
            return CustomerFeatures.newBuilder().setCustomerId(customerId)           ...
                .build();                                                     #J
        }
        return null;
    }

    private ResultSet executeStatement(Long customerId) {
        PreparedStatement pStmt = getSession().prepare(queryStatement);
        return getSession().execute(pStmt.bind(customerId));
    }

    private CqlSession getSession() {
        if (session == null || session.isClosed()) {
            CqlSessionBuilder builder = CqlSession.builder()
                .addContactPoint(getAddress())
                .withLocalDatacenter("datacenter1")
                .withKeyspace(CqlIdentifier.fromCql("featurestore"));

            session = builder.build();
        }
        return session;
    }

    #A Get the feature store hostname from the configs
    #B Get the feature store port from the configs
    #C Get the credentials for the feature store
    #D Get the name of the table in the feature store to query
    #E Get the all of features we need to retrieve from the feature store
```

```

#F Construct a SQL statement using the specified fields and table name
#G Construct a prepared statement using the specified fields
#H Execute the prepared statement for the given id
#I There is only one record for any given id
#J Map the fields to the outgoing schema

```

The RestaurantFeaturesLookup Function will connect to the feature store and retrieve only the specified features from the feature store that were specified by the data science team when they published them to the state store. Next, it maps those values to the outgoing schema type and publishes the message to the incoming topic of the Pulsar Function that will execute the delivery time estimation model.

11.4.3 Model Deployment

The final step in the process of deploying a machine learning model in near real-time mode is writing the Pulsar Function itself. Fortunately, once a machine learning model is exported to the PMML format it can be deployed to production using a wide variety of execution engines available in Java. In our case we will use an open-source library called **JPMMML**, which we can include in our maven dependencies as shown in listing 11.6.

Listing 11.6 The JPMMML Library Dependency

```

<dependency>
<groupId>org.jpmml</groupId>
<artifactId>pmmml-evaluator</artifactId>
<version>1.5.15</version>
</dependency>

```

This library allows us to import PMML models, and then use them to generate predictions. Once the PMML model has been loaded into the JPMMML evaluator class, we are ready to generate delivery time predictions on incoming food orders. Therefore, as you can see from Listing 11.7, the very first step in the process is to retrieve the PMML model that is stored inside Pulsar's state store and use it to initialize the appropriate model evaluator. In this case, we need a regression model evaluator since the delivery time estimation model uses linear regression.

Listing 11.7 The Delivery Time Estimation Function

```

import org.dmg.pmml.FieldName;
import org.dmg.pmml.regression.RegressionModel;
import org.jpmml.evaluator.ModelEvaluator;
import org.jpmml.evaluator.regression.RegressionModelEvaluator;      #A
import org.jpmml.model.PMMLUtil;

import com.gottaeat.domain.geography.LatLon;
import com.gottaeat.domain.order.FoodOrderML;

public class DeliveryTimeEstimator implements
    Function<FoodOrderML, FoodOrderML> {      #B

private IgniteClient client;
private ClientCache<Long, LatLon> cache;      #C
private String datagridUrl;

```

```

private String locationCacheName;

private byte[] mlModel = null;
private ModelEvaluator<RegressionModel> evaluator;      #D

    @Override
public FoodOrderML process(FoodOrderML order, Context ctx)
throws Exception {

    if (initialized()) {
        mlModel = ctx.getState(MODEL_KEY).array();      #E
        evaluator = new RegressionModelEvaluator(
            PMMLUtil.unmarshal(new ByteArrayInputStream(mlModel)));      #F
    }

    HashMap<FieldName, Double> featureVector = new HashMap<>();

    featureVector.put(FieldName.create("avg_prep_last_hour"),
        order.getRestaurantFeatures().getAvgMealPrepLastHour());

    featureVector.put(FieldName.create("avg_prep_last_7days"),
        order.getRestaurantFeatures().getAvgMealPrepLast7Days());      #G

    ...

    featureVector.put(FieldName.create("driver_lat"),
        getCache().get(order.getAssignedDriverId()).getLatitude());

    featureVector.put(FieldName.create("driver_long"),
        getCache().get(order.getAssignedDriverId()).getLongitude());      #H

    Long travel = (Long)evaluator.evaluate(featureVector)
        .get(FieldName.create("travel_time"));      #I

    order.setEstimatedArrival(System.currentTimeMillis() + travel);      #J
    return order;
}

...
}

```

#A Using the JPMML regression evaluator for the linear regression ML model
#B Incoming message contains order details and retrieved features from the feature store
#C The model requires information from the in-memory data grid
#D The JPMML model evaluator
#E Retrieve the model from the Pulsar state store
#F Load the model into the Regression Evaluator class
#G Populate the feature vector with values stored inside the incoming message
#H Populate the feature vector with the driver location data from the IMDG
#I Pass the feature vector to the model and retrieve the prediction
#J Compute the estimated arrival time by adding the predicted travel time to the current time

Once the evaluator has been initialized, the `DeliveryTimeEstimator` constructs a Feature Vector and populates it with values contained inside the incoming message along with some values from other low-latency data sources such as an in-memory datagrid. In this case the

model requires the driver's current location (latitude/longitude) which is only available from the IMDG.

Finally, it applies the evaluator to get a predicted value for the travel time (in milliseconds) of the assigned driver to get to the delivery location. This value is then added to the current time and written to the outgoing message. As you can see, this is straightforward process that can be used to host models that have been developed and trained in any of the numerous PMML-supported languages and frameworks.

11.5 Neural Nets

While the PMML format is very flexible and supports a wide variety of languages and machine learning models, there are instances when you will need to deploy an ML model that isn't supported by PMML. Neural net models which can be used for solving a variety of business problems such as sales forecasting, data validation, or natural language processing cannot be represented as PMML and therefore require a different approach to be deployed in near real-time mode.

Modeled loosely on the human brain, a neural net consists of thousands or even millions of artificial neurons (nodes) that are interconnected. Most of today's neural nets are organized into multiple layers of nodes as shown in Figure 11.6. Each node gets weighted input data, that is fed into the computing node function and outputs the result of the function to the subsequent layer in the network. The data is fed-forward through the network until a single value is produced.

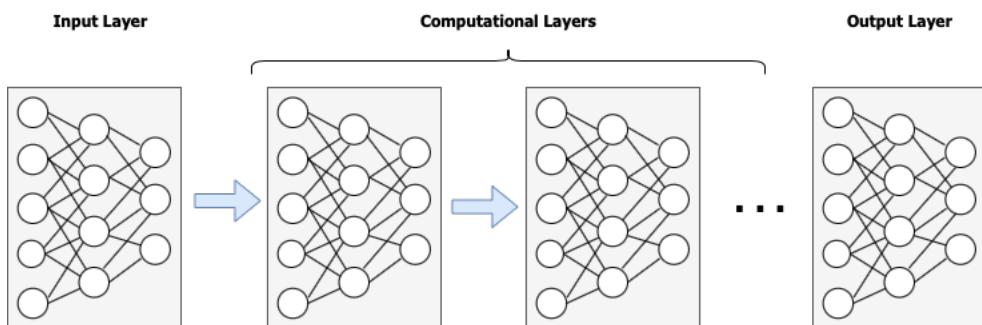


Figure 11.6: A neural net is comprised of an input layer, an output layer, and any number of hidden computational layers. The term **deep learning** refers to a neural network that is several computational layers deep.

The basic concept of layering is that each additional layer of the network increases the accuracy of the model itself. In fact, the term **deep learning** refers to the use of neural nets that are several layers "deep". In this section I will demonstrate the process of training and deploying a neural net using the popular high-level Neural Network API known as **Keras**, which is written in Python.

11.5.1 Neural Net Training

Neural nets are designed to recognize patterns in data, and they learn to do this by analyzing large training datasets. The process of training a neural net involves using a training dataset to determine the proper model weights for each node in the network. The training sets are often labelled, so that the expected outputs are known in advance, and then the weights are adjusted until the model produces the expected results. Therefore, it is important to remember that these weights are critical to the performance and accuracy of these models.

The first step is to train a model using the Keras library in Python using your training data as shown in Listing 11.8. The input to the model shown is ten binary features that describe the various features of a food order such as average order price of the customer, the distance between the customers home address and the delivery address of the order, etc. The output is a single variable that describes the probability that the order is fraudulent.

Listing 11.8 Training the Neural Net using Python

```
import keras
from keras import models, layers    #A
# Define the model structure
model = models.Sequential()      #B
model.add(layers.Dense(64, activation='relu', input_shape=(10,)))
...
model.add(layers.Dense(1, activation='sigmoid'))
model.compile(optimizer='rmsprop', loss='binary_crossentropy',
               metrics=[auc])

history = model.fit(x, y, epochs=100, batch_size=100,
                     validation_split = .2, verbose=0)    #C

model.save("games.h5")    #D
```

#A Use the Keras libraries

#B Define the model structure

#C Compile and fit the model

#D Save the model in h5 format

As you can see from Figure 11.7, each node inside a neural net takes in a feature vector as input along with a set of weights associated with each feature. These weights allow us to assign more importance to some features than others when generating our prediction, such as the distance between the delivery address of the order and the customer's home address. If that feature is a good indicator of a fraudulent transaction, then it should have a higher weight. The process of the training and fitting the model yields a set of optimal weights for each input node, in the neural net.

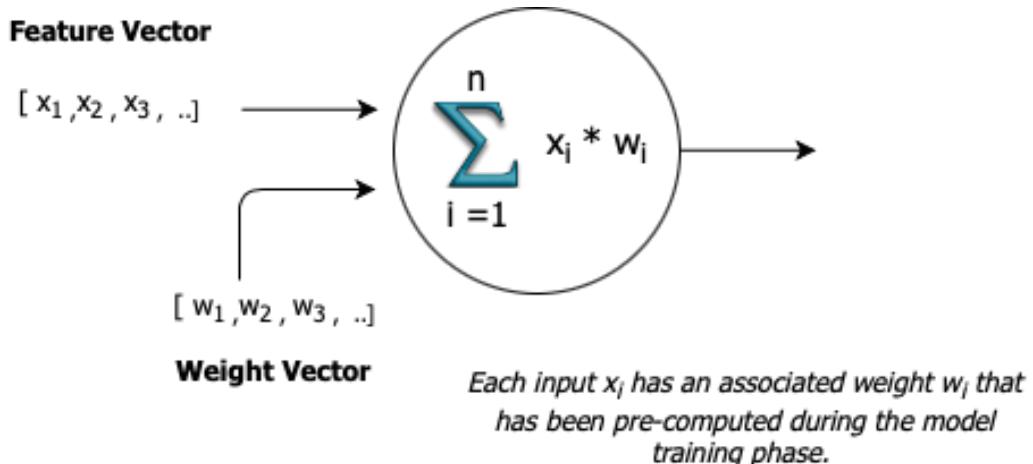


Figure 11.7: Each node in the neural net takes a feature vector as input along with a pre-computed set of weights that are associated with each feature. These weights are calculated during the model training phase and are critical to accuracy of the neural net.

Once the model has been properly trained, i.e., you have calculated the optimal weights for each input node, and is ready to deploy, you can save it in a specialized Keras-specific format known as **H5**. This format retains the complete models including the architecture, weights, and training configuration whereas Keras models exported to JSON formal only contain the model architecture but NOT the calculated weights. Therefore, be sure to always use the H5 format when exporting your Keras-based neural nets.

11.5.2 Neural Net Deployment in Java

To execute the Keras model within a Java runtime environment, we'll use the DeepLearning4j library (DL4J). It provides functionality for deep learning in Java and can load and utilize models trained with Keras. One of the key concepts to become familiar with when using DL4J is tensors. Java does not have a built-in library for efficient tensor options, which is why I included the ND4J library in the Maven dependencies shown in Listing 11.9.

Listing 11.9 DeepLearning4J Library Dependencies

```
<dependency>
    <groupId>org.deeplearning4j</groupId>
    <artifactId>deeplearning4j-core</artifactId>
    <version>1.0.0-M1</version>
</dependency>
<dependency>
    <groupId>org.deeplearning4j</groupId>
    <artifactId>deeplearning4j-modelimport</artifactId>
    <version>1.0.0-M1</version>
</dependency>
<dependency>
    <groupId>org.nd4j</groupId>
```

```
<artifactId>nd4j-native-platform</artifactId>
<version>1.0.0-M1</version>
</dependency>
```

Now that we have the DL4J libraries set up, we can start using the neural net in near real-time mode by embedding it inside a Pulsar Function as shown in Listing 11.10.

Listing 11.10 Deploying the Neural Net inside a Java-based Pulsar Function

```
import org.deeplearning4j.nn.modelimport.keras.KerasModelImport;
import org.deeplearning4j.nn.multilayer.MultilayerNetwork;      #A
import org.nd4j.linalg.api.ndarray.INDArray;
import org.nd4j.linalg.factory.Nd4j;

public class FraudDetectionFunction implements
    Function<FraudFeatures, Double> {      #B

    public static final String MODEL_KEY = "key";
    private MultiLayerNetwork model;        #C

    public Double process(FraudFeatures input, Context ctx)
        throws Exception {
        if (model == null) {
            InputStream modelHdf5Stream = new ByteArrayInputStream(
                ctx.getState(MODEL_KEY).array());      #D
            model = KerasModelImport.importKerasSequentialModelAndWeights(
                modelHdf5Stream);      #E
        }

        INDArray features = Nd4j.zeros(10);      #F
        features.putScalar(0, input.getAverageSpend());
        features.putScalar(1, input.getDistanceFromMainAddress());
        features.putScalar(2, input.getCreditCardScore());      #G
        . . .

        return model.output(features).getDouble(0);      #H
    }
}
```

#A Using the DL4J libraries

#B Input is a collection of features used for fraud detection

#C The machine learning model

#D Retrieve the trained model from the state store

#E Initialize the model with the HDF5 file

#F Create an empty feature vector for 10 features

#G Populate the feature vector with values

#H Execute the model using the given feature vector and return the predicted fraud probability

The model object provides predict and output methods. The predict method returns a class prediction (fraud or not fraud), while the output method returns a numeric value representing the exact probability. In this case we will return the numeric value so that we can evaluate it further, e.g., compare it to a configurable threshold used to define our threshold for fraud.

11.6 Summary

- Pulsar Functions can be used to provide near real-time machine learning on streaming data to produce actionable insights.
- Providing near real-time predictions requires a machine learning model that takes a pre-defined set of inputs known as a feature set.
- A feature is a numeric representation of an individual aspect of an object, such as the average meal preparation time of a restaurant.
- Most features within a feature vector cannot be calculated using the data from a single message, nor can they be computed in a timely manner. Therefore, it is common to have ancillary processing compute these values in the background and store them in a low-latency data store.
- The Predictive Model Markup Language (PMML) is a standard format for representing ML models developed in a variety of languages which helps make ML models portable.
- There is an open-source Java-based project that supports the evaluation of PMML models, which allows us to easily execute any PMML supported model inside a Pulsar Function.
- You can use other language specific libraries to execute non-PMML models inside Pulsar Functions as well.

12

Edge Analytics

This chapter covers:

- Using Pulsar for Edge computing
- Using Pulsar to perform edge analytics
- Performing anomaly detection on the edge using Pulsar Functions
- Performing statistical analytics on the edge using Pulsar Functions

If you are like most people, when you hear the term the *Internet of Things* (IoT) you tend to think of smart thermostats, Internet-connected refrigerators, or personal data assistants such as Alexa. While these consumer-oriented IoT devices tend to get a lot of attention, there is a subset of IoT called the *industrial Internet of Things* (IIoT) which focuses on the use of sensors that are connected to machinery and vehicles within the transport, energy and industrial sectors. Companies use the information collected from sensors that are physically embedded inside industrial equipment to monitor, automate, and predict all kinds of industrial processes and outcomes.

The data collected from these IIoT sensors has several practical applications including monitoring tens of thousands of miles of remote industrial equipment within the energy industry to ensure that there are no imminent failures that could lead to a catastrophic event resulting in a large environmental impact. Sensor data can also be gathered from non-stationary IIoT sensors, such as in a large fleet of refrigerated tractor trailers used to distribute a vaccine across the globe that must be kept below a certain temperature in order to remain effective. These sensors allow us to detect a gradual warming within any given refrigeration unit and re-route the cargo to a nearby maintenance facility for repairs, etc.

In such a situation it is important that we detect the change in temperature within the refrigeration units as soon as possible so that we can react in time to preserve the heat sensitive cargo. If we waited until the cargo arrived at its intended destination before we checked the temperature, it would be too late, and the vaccine would be useless. This

phenomenon is often referred to as the diminishing time value of data, since the value obtained from the information is at its highest point immediately after the event occurred and rapidly diminishes over time. In the case of the refrigeration unit failure, the sooner we can react to that information the better. If we are unaware of the failure for hours, then the cargo is most likely going to spoil, and the information is no longer actionable because it is too late to do anything about it. As you can see from Figure 12.1, the longer the response time to such a catastrophic event, the less impact any remedial action will have on the system.

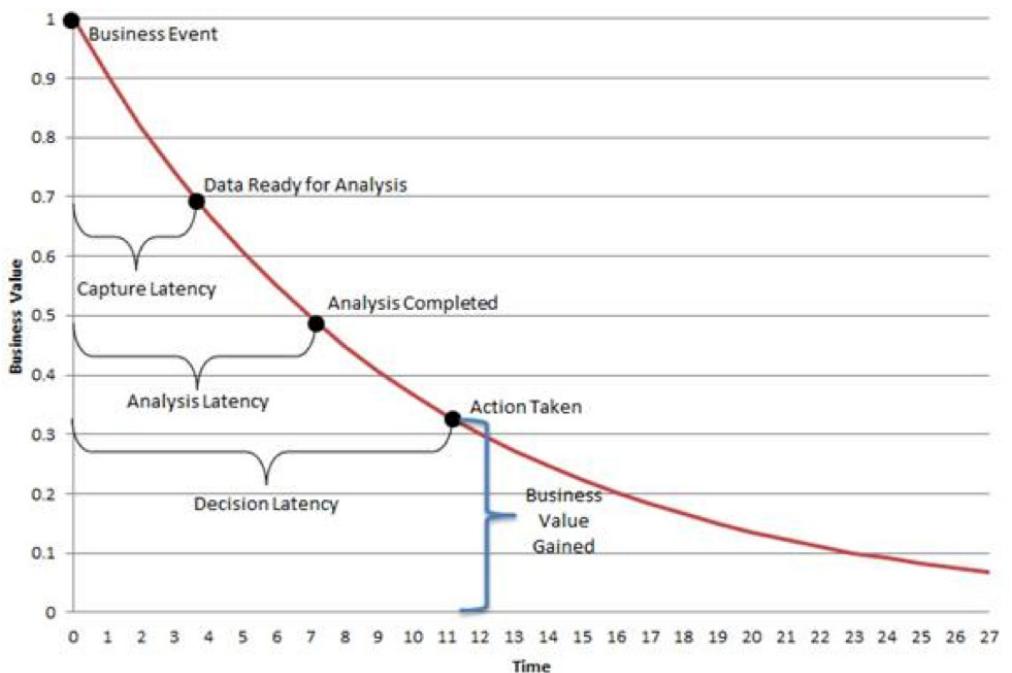


Figure 12.1: The value of any piece of information diminishes rapidly over time, and the goal of edge computing is to reduce the overall decision latency by eliminating the capture latency produced by transmitting the data from the sensor to the cloud for analysis.

The amount of time between when an event occurs and when a corresponding action is taken in response is known as the decision latency and is comprised of two components: amount of the time required to transfer of data to your analysis software which is known as the *capture latency* and the *analysis latency* which is the amount of time required to analyze the data in order to determine what action to take.

From a technological perspective, the IIoT provides the same basic capability as any other "smart" consumer IoT device which is the automated instrumentation and reporting capabilities to physical devices that previously did not have those capabilities before. For example, the defining characteristic of a "smart" thermostat is that it can communicate its

current reading and be adjusted remotely via a smartphone app. That being said, the scale of a typical IIoT deployment is significantly larger than a simple system that lets you adjust your thermostat from your phone.

With potentially millions of sensors spread across a single factory plant floor or a large fleet of tractor trailers, each of which are producing a new metric every second one can easily see that these IIoT datasets are both high-volume and high-frequency. A common approach to processing these datasets is to collect all of the individual data elements, transfer them to the cloud, and use traditional SQL-based data analysis tools such as Apache Hive or more traditional data warehouses. This ensures that the analysis is done on a complete dataset from all of the sensors so that any inter-sensor reading relationships can be observed and used for analysis e.g., the correlation between a temperature sensor and the overall plant humidity from a different sensor can be tracked and analyzed.

However, this approach has some serious disadvantages such as: significant decision latency (the time between when the event occurred and the time it gets processed); cost inefficiencies associated with having to provision sufficient network bandwidth and computing resources to process such large datasets along with the storage cost of retaining all of this information.

From a practical perspective, the amount of the time required to transfer of data from most IIoT platforms to a cloud computing environment for analysis makes it nearly impossible to perform any real-time reaction to a potentially catastrophic event. While the most dramatic examples of such an event would be the detection of faults in power plants or airplanes before they explode or crash, the speed of data analysis in most Industrial Internet of Things (IIoT) applications is critical as well.

In order to overcome this limitation, some of the data processing and analysis of IIoT data can be performed on infrastructure that is physically located closer to the source of the data itself. Bringing computation closer to the source of the data decreases the capture latency and allows applications to respond to data almost instantaneously as it's being created, rather than having to wait for the information to be transmitted to the over the internet before processing it. This practice of processing data near the edge of the network where the data is being generated, instead of in a centralized data collection point such as a data center or cloud often referred to as *edge computing*.

In this chapter I'll demonstrate how we can deploy Pulsar Functions inside an edge computing environment in order to provide near real-time data processing and analysis in order to react more quickly to events within an IIoT environment to minimize the decision latency between the time a high-value event is perceived and when the appropriate response is made.

12.1 Industrial Internet of Things Architecture

An IIoT platform acts as a bridge between the physical world of industrial equipment and the computational world of automated control systems used to monitor and react to it. As such, one of the primary goals of any IIoT system is the collection and processing the data generated by all of the physical sensors distributed across various pieces of industrial equipment. While every IIoT deployment is different, they are all comprised of the three

logical architectural layers, shown in Figure 12.2. that play an important role in the data acquisition and analysis process.

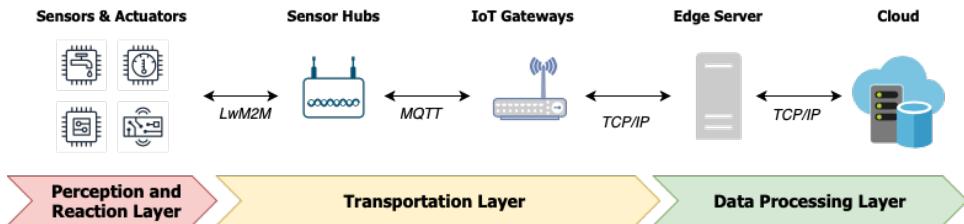


Figure 12.2: The three logical layers of an Industrial Internet of Things architecture ingest data, analyzes it, and then presents the information so that can be used by humans or by autonomous systems for making contextually relevant decisions in real time.

Within an IIoT environment, there can be millions of sensors, controllers, and actuators distributed across various pieces of industrial equipment within a single factory location. All of these sensors and devices collectively form what is commonly referred to as the *perception and reaction layer*, because they allow us to perceive what is going on within the physical world.

12.1.1 The Perception and Reaction Layer

The *perception and reaction layer* contains all of the hardware components, aka the *Things* within the Internet of Things. As the basis for every IIoT system, these connected devices are responsible for perceiving the physical conditions of the industrial equipment and surrounding environment as sensed through numerous sensors that are either embedded in the devices themselves or implemented as standalone objects. These sensors emit a continuous, real-time stream of sensor readings either over *lightweight machine-to-machine* (LwM2M) protocols such as Bluetooth, Zigbee, and Z-wave or longer-range protocols such as MQTT or LoRa if they have a wired connection.

In practice, most IIoT environments require multiple network protocols to support the various devices within the environment. Battery-powered sensors for example, can only communicate over short distances using lightweight protocols designed for short-range use. In general, the larger the distance the signal needs to travel, the more power that is required by the device to send it. In case of battery-operated devices using a longer-range protocol to send its data isn't practical.

One of the primary responsibilities of the perception and reaction layer is to perceive the environment via the sensors by capturing their readings and relaying them up to the data processing layer. The other critical responsibility of this layer reacting to the actionable insights produced by the processing layer and translating it into an immediate physical action when we detect a potentially dangerous condition. Being able to detect a potentially dangerous condition doesn't provide much business value if you can't respond to it in a meaningful way.

Several decades before the emergence of IIoT, most large manufacturers invested large amounts of time and effort into specialized software systems known as *Supervisory Control and Data Acquisition* (SCADA) systems that allow them to monitor and control their industrial equipment. These systems contain control networks that permit the automated operation of the mechanical or electro-mechanical devices known as *actuators* that can perform a variety of manual operations such as opening a pressure valve on a piece of equipment or completely turning off the power to a given machine.

By leveraging the existing control network within these SCADA systems, we are able to programmatically utilize these actuators from within our IIoT application by simply sending the correct command to the active actuator. Whereas the sensors act as the "eyes" of this layer, these actuators serve any equally important role as the "hands" that allow us to respond to the data. Without them, our ability to react to a catastrophic sensor reading would be non-existent.

The information collected within the perception and reaction layer is sent over the *transportation layer*, which as the name implies, is responsible for the secure transmission of the sensor data up to a centralized data processing layer.

12.1.2 The Transportation Layer

The sensor readings that are generated inside the perception and reaction layer and transmitted over LwM2M protocols are detected by intermediary devices known as sensor hubs that sit at the outermost edge of the transmission layer. These specialized devices can receive the sensor readings being broadcast over the LwM2M protocols by low-power devices.

The primary purpose of the sensor hubs is to provide a bridge between short-range and long-range communication technologies. Battery-enabled IoT devices communicate with the sensor hubs using short-range wireless transmission modes such as Bluetooth LE, Zigbee, and Z-wave. Upon receipt of the messages, the sensor hubs immediately relay the message over a longer-range protocol such as CoAP, MQTT, or LoRa to a device known as an IoT Gateway. Among these longer-range protocols, the *Message Queuing Telemetry Transport* (MQTT) specializes in low-bandwidth, high-latency environments, which makes it the one of the most commonly used protocols inside the Industrial IoT space.

An *IoT gateway* is a physical device that serves as the connection point between the cloud and the sensor hubs. They provide a communication bridge between the MQTT protocol and the Pulsar messaging protocol and are responsible for aggregating all of the incoming sensor readings sent over the lightweight, binary MQTT protocol and aggregating them before forwarding them to the data processing layer using Pulsar's messaging protocol.

12.1.3 The Data Processing Layer

As you can see from Figure 12.3, the *processing layer* spans two physical layers; the edge servers that are located close to the industrial equipment, and the corporate data center or cloud infrastructure. The edge servers are used to aggregate, filter and prioritize data from the massive volume of data that a large IIoT deployment typically generates in order to minimize the volume of information that needs to be forwarded to the cloud. This pre-processing of the data helps reduce transmission costs and response times.

From a hardware perspective, the edge processing layer consists of one or more traditional computers/servers that are located within the industrial location itself e.g., inside the factory. While the computing capacity at this layer might be constrained due to physical space limitations of the given location, these devices always have an internet connection that allows them to forward the data it has collected to the company's data center and/or cloud provider for further analysis and archival.

12.2 A Pulsar Based Processing Layer

Now that we have a basic understanding of an IIoT architecture, I want to demonstrate how we can use Apache Pulsar to enhance the processing capabilities of the architecture by extending the data processing layer closer proximity to the sensors and actuators than in a traditional IIoT setting.

First let's review the computing resources that are available on each of the hardware devices within the IIoT platform. As you can see, Figure 12.3 shows that inverse relationship between the available computing resources and its proximity to the industrial equipment. The sensors, actuators, and other smart devices located within the perception and reaction layer are microcontroller based, with limited memory and processing power. They are primarily battery-operated and therefore are not good candidates for performing any sort of computation. While these wireless devices can be augmented with energy harvesting devices that convert ambient energy into electrical energy, their power is best reserved for the wireless communication with the sensor hubs rather than used for computation.

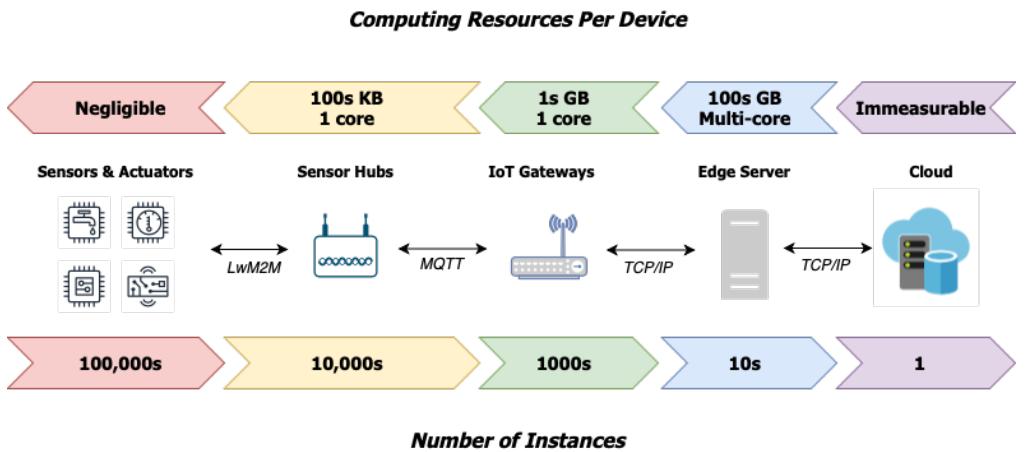


Figure 12.3: There is an inverse relationship between the number of devices deployed at each layer and the computing resources found within each device.

Sensor Hubs are typically hosted on slightly larger devices referred to as *System-on-a-Chip* (SoC) that can have some or all of the components of a traditional computer including a CPU, RAM, and external storage although at a smaller scale. But given the sheer number of

these devices their specifications are kept to a minimum in order to be economically feasible to deploy in large number. Since these devices are primarily used to receive and transmit the data, they only require a limited amount of memory to buffer the messages before re-transmitting them over a different protocol.

IoT Gateways are also hosted on System-on-a-Chip (SoC) hardware with one of the most popular platforms for these devices being the Raspberry Pi which can have up to 8GB of RAM, a quad-core CPU, and one Micro-SD card slot for up to one terabyte (1TB) of storage. These devices can also run traditional operating system software such as Linux which makes them an ideal candidate for running complex software applications such as Pulsar.

The last physical piece of hardware are the Edge servers, which are an optional feature within the IIoT architecture. Depending upon the nature of the industrial environment these devices can range in size from multiple servers with terabytes of RAM, multiple cores, and terabytes of disk storage residing inside a server closet on a factory floor, to single desktop computer on a remote drilling site. Just like the IoT Gateways, devices at this layer run more traditional operating systems and resemble what most people think of when you use the term computer.

From a computing perspective, any device that has sufficient computing resources (8GB of RAM, and a multi-core x86 CPU) to run a traditional operating system and an internet connection is a potential hardware platform for hosting a Pulsar Broker. Within an IIoT architecture, this would include not only the Edge servers and the IoT Gateway devices, but also any sensor hub that is running on a sufficiently equipped SoC device as well.

Installing a Pulsar Broker on these devices enables us to deploy Pulsar Functions directly on them in order to perform the analysis much closer to the source of the data. Doing so effectively extends the data processing layer closer to the origin of the sensor data as shown in Figure 12.4. We are effectively extending the edge closer to the industrial equipment and doing so creates a large distributed computational framework where Pulsar Functions can be deployed across two-tiers of the architecture to perform parallel computation on the data.

The key takeaway from this is the fact that we can deploy our Pulsar Functions on any of the devices inside the IIoT environment that are capable of hosting a Pulsar Broker and have them perform complex data analysis there on the edge rather than just collecting the events and forwarding upstream for processing as is done in a more traditional IIoT environment.

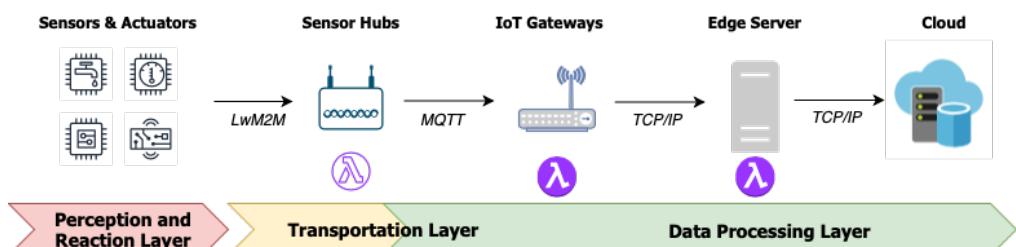


Figure 12.4: By installing Pulsar Brokers on the IoT Gateways and Edge Servers, we can extend the data processing layer closer to the source of the data itself, which will enable us to react to it much quicker.

To be fair, some IIoT vendors do provide software packages that can be used to process data on the IoT Gateway devices as well, but the features are typically limited to more rudimentary capabilities such as filtering and aggregation. Furthermore, these software packages are closed-source and do not allow for you to extend the framework to add your own data processing functions that you might want. Whereas with Pulsar Functions, you can easily create and use your own functions to perform more complex data analysis.

It is also worth noting that Apache Pulsar provides support for the MQTT protocol via a plugin which allows devices using the MQTT protocol, such as the sensor hubs, to publish messages directly to a topic on a Pulsar Broker. This allows us to use Pulsar as an IoT Gateway device without the need for another piece of software to acts as the MQTT message broker in order to consume and process the sensor data directly on the gateway itself as you can see from Figure 12.5. A Pulsar-based IoT Gateway supports bi-directional communications over the MQTT protocol which allows the Pulsar Functions to send messages to the actuators in response to potentially catastrophic events they detect.

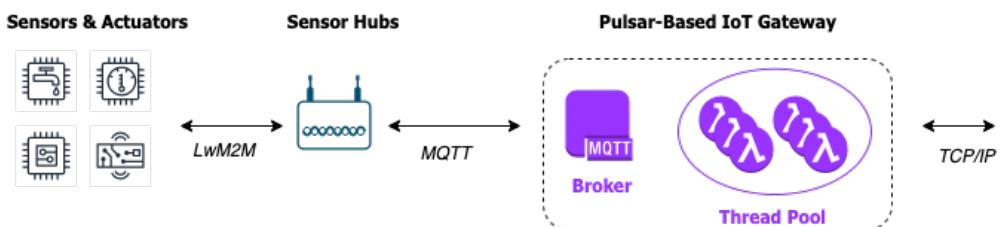


Figure 12.5: The complete functionality of an IoT Gateway can be performed using a Pulsar Broker that has the MQTT plugin enabled which allows it to receive messages from the sensor hubs while the Pulsar Functions can be administered via the TCP/IP connection.

The TCP/IP connection not only enables user to deploy, update, and administer Pulsar Functions directly on the IoT Gateway device, but it also allows the Pulsar Broker to communicate with an Apache BookKeeper based storage layer hosted in a remote location. Last and certainly not least, the TCP/IP connection allows us to communicate with other Pulsar clusters within the IIoT architecture particularly ones that have been deployed on the Edge Servers. This allows us to forward data generated on all of the IoT Gateways up to a centralized location for additional edge processing before finally being sent to the cloud for archival.

12.3 Edge Analytics

The practice of performing some or all of the data analysis on infrastructure outside of a traditional data center or cloud computing environment is commonly referred to as *edge analytics* and differs from traditional analytics in a few other key aspects that must be kept in mind when you are designing your overall analytics strategy. For starters, the analysis must be done on streaming datasets where each piece of information will be provided to your Pulsar Function only once. Since there is very limited physical disk space in an edge environment, these sensor values are not retained and thus cannot be re-read at a future

point in time. If you wanted to determine the average reading of a sensor over the previous hour in a cloud environment for example, you could simply execute a SQL query to calculate it for you from the historical data. This is not an option with edge analytics.

Another big difference is the fact that the closer you move the processing to the sensors, the less information you have in your dataset, which makes the detection of patterns between sensors that are not co-located within the range of the same IoT Gateway impossible.

12.3.1 Telemetric Data

In order to get a better understanding of the term edge analytics and what we are trying to accomplish by performing some of the data analysis on the edge, it is best to start with a basic understanding of the type of data we are processing in an IIoT environment. The overarching function of any IIoT system is the collection of sensor data so that it can be used monitor and manage the company's industrial infrastructure. Once the data is collected, it can then be analyzed for any potential events of interest that might need to be addressed.

These sensors are constantly emitting a stream of observations obtained through repeated measurements of the same variable over time such as a sensor that sends the temperature reading of a specific piece of equipment every second. These sequences of numerical data points taken at fixed intervals in chronological time order is referred to as time-series data. The entire process of collecting this time-series information in form of measurements or statistical data and forwarding it to remote systems is often referred to as *telemetry*. Like nearly all time-series datasets, this telemetric data will often have one or more of the following characteristics:

- *Trend*: When referring to the “trend” in time series data, it means that the data has a pronounce trajectory in one direction (either up or down) over a specified timeframe. A good example of a trend would be a steady long-term increase in network traffic.
- *Cycles*: Repeating and predictable fluctuations in the data that are not at a fixed frequency and thus cannot correlated to any specific time-period or interval.
- *Seasonality*: If there are regular and predictable cycles within the data that are correlated with the calendar (daily, weekly, etc.) then the data has a seasonality characteristic. This differs from a trend because the cycle fluctuates for a short period of time and is usually driven by outside factors such as a spike in network traffic to a popular e-commerce site during Cyber Monday.
- *Noise*: This refers to randomness in the data points that cannot be correlated with any explained trends. Noise is unsystematic, short-term, and needs to be filtered out in order to minimize its determinantal impact on our predictions. Consider a temperature sensor that has consistently reported a value of 200 degrees Fahrenheit over the past hour. Any sensor reading that is significantly different than the previous values we have received is most likely just noise and should be ignored. For instance, a single sensor reading of 25 degrees Fahrenheit is mostly likely not an accurate reading and should be ignored.

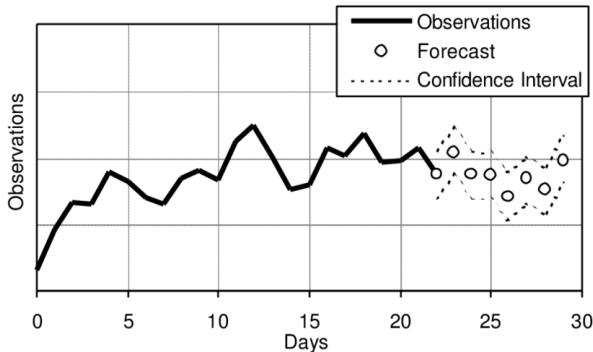


Figure 12.6: Time series forecasting is the use of a time-series data to predict future values based on previously observed values.

In the context of IIoT, edge analytics are used to detect these characteristics of the data so they can then be used to predict future values based on previously observed values as shown in Figure 12.6. The real observed values can then be compared to the predicted values to determine if some sort of action needs to be taken. For instance, if a pressure reading is trending downward this might indicate a loss of pressure in the line and a maintenance team should be dispatched to investigate.

12.3.2 Univariate and Multivariate

The other aspect of telemetric datasets is the number of variables that are being tracked within them. The most common scenario is when the data contains a set of observations of a single variable, e.g., the readings from the exact same sensor. The more formal term for this type of dataset is **univariate**, and for purposes we will assume that all of the datasets collected at the IoT Gateway layer are univariate. As you may have guessed, any dataset that tracks more than one variable are known as **multivariate** and allow us to track the relationship between multiple sensor readings over the same timeframe. Rather than containing a single value at a given point in time, these datasets contain several values. Within a Pulsar enabled IIoT architecture, multivariant datasets are generated using Pulsar Functions by combining several univariant datasets together as shown in Figure 12.7 where the values from each sensor taken at the same time are combined into a tuple of 3 values.

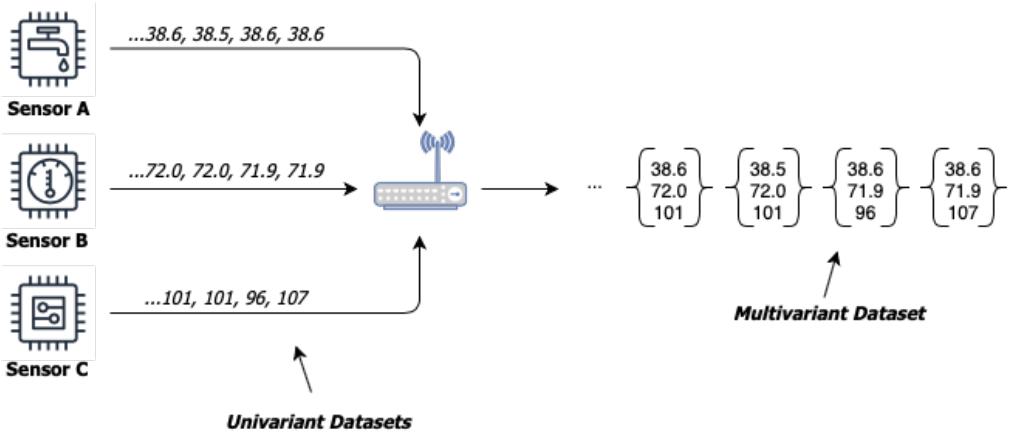


Figure 12.7: Each sensor emits a sequence of readings at a pre-determined interval. Upon receipt on the IoT Gateway, these univariate datasets are combined into a multivariate dataset that contains the values of all three sensors at a given point in time. This allows us to detect patterns and correlations between these sensor values.

Multivariate datasets can be used to perform more complex and accurate analysis by including data from multiple sources that are strongly correlated. Since there is no limit to the number of variables that can be contained within these datasets, any number of readings may be combined as needed. In fact, these datasets are well suited to serve as feature stores for any ML model you wish to deploy on the edge. As you may recall from Chapter 11, feature stores contain a set of pre-calculated values required by ML models. These feature sets are populated by external processes that relied on historical data to calculate the values.

Whereas, in a Pulsar-enabled IIoT environment these multivariate datasets are populated from the univariate datasets being collected on the edge which ensures that your ML models are using the most recent data to make their predictions.

12.4 Univariate Analysis

The endless streams of readings taken from the same sensor are the foundation of edge analytics. These univariate datasets represent the raw data used to perform the analysis of the IIoT data as a whole. Therefore, it is best to start by covering the type of analysis that can be performed on these univariate datasets. A summary of the various analytic processing commonly performed is depicted in Figure 12.8.

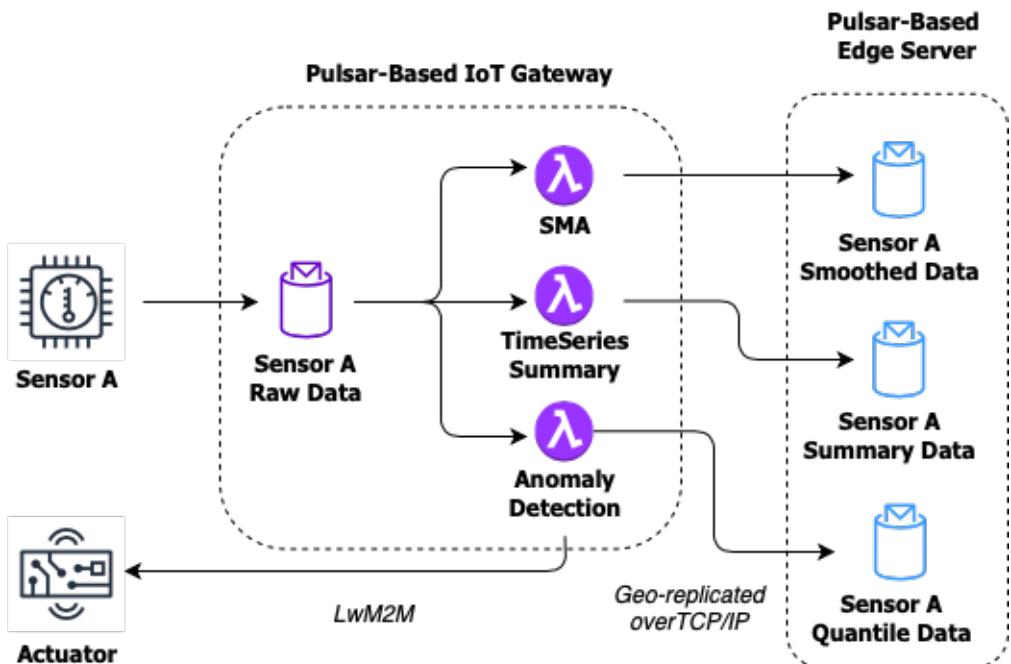


Figure 12.8: The sensor publishes its readings to the “Sensor A Raw Data” topic that is used as input to three Pulsar Functions. Two of the Functions, SMA and TimeSeriesSummary use the values to compute statistical values from the raw data before publishing these statistics to local topics that are configured to be geo-replicated to the Pulsar cluster running on the Edge Server. The third function determines if the sensor value is anomalous and can active an actuator in response to any potentially catastrophic event it detects.

This processing is accomplished using Pulsar Functions deployed on the IoT Gateways closest to the source of the data. Therefore, it is important that these Functions minimize the amount of memory and CPU required to perform their analysis.

12.4.1 Noise Reduction

I previously mentioned how there can be noise (aka randomness) within the sensor data. In order to better forecast future data values, an important pre-processing step is the reduction of the noise these univariate datasets. A common technique used to smooth out these fluctuations of the data, is to simply compute the mathematical average of the data points within a pre-defined time window to produce a value. This computed moving average is then retained rather than the raw values. Doing so minimizes the impact that any individual sensor reading has on the reported value, e.g., if you have 99 sensor readings with the same value of say 70 and one with a value of 100, then using a computed average of 70.3 would effectively “smooth” out the data and provide a value that is more indicative of the sensor reading over that interval.

While there are many different moving average models, for the sake of brevity, I will only cover the *simple moving average* in this chapter. The way that a simple moving average is calculated is by retaining the most recent subset of the time-series data such as the last 100 sensor readings. When a new reading arrives, it is used to replace the oldest value in the collection. Once the newest value has been added, the mathematical average of these remaining values is computed and returned.

Listing 12.1: A Pulsar Function to Calculate the Simple Moving Average

```
public class SimpleMovingAverageFunction implements Function<Double, Void> {

    private CircularFifoQueue<Double> values;      #A
    private PulsarClient client;        #B
    private String remotePulsarUrl, topicName;
    private boolean initialized;
    private Producer<Double> producer;

    @Override
        public Void process(Double input, Context ctx) throws Exception {
            if (!initialized) {
                initialize(ctx);
            }

            values.add(input);      #C
            double average = values.stream()
                .mapToDouble(i->i).average().getAsDouble();      #D
            publish(average);      #E
            return null;
        }

    private void publish(double average) {
        try {
            getProducer().send(average);
        } catch (PulsarClientException e) {
            e.printStackTrace();
        }
    }

    private PulsarClient getEdgePulsarClient() throws PulsarClientException {
        if (client == null) {
            client = PulsarClient.builder().serviceUrl(remotePulsarUrl).build();
        }
        return client;
    }

    private Producer<Double> getProducer() throws PulsarClientException {
        if (producer == null) {
            producer = getEdgePulsarClient()
                .newProducer(Schema.DOUBLE)
                .topic(topicName).create();
        }
        return producer;
    }

    private void initialize(Context ctx) {
        initialized = true;
        Integer size = (Integer) ctx.getUserConfigValueOrDefault("size", 100);
    }
}
```

```

        values = new CircularFifoQueue<Double> (size);
        remotePulsarUrl = ctx.getUserConfigValue("pulsarUrl").get().toString();
        topicName = ctx.getUserConfigValue("topicName").get().toString();
    }
}

```

#A The circular buffer of values that automatically removes the oldest item.
#B A pulsar client for the Pulsar cluster running on the Edge Servers.
#C Add the sensor reading to the list of values.
#D Calculate the simple moving average.
#E Publish the computed value to a topic on the Edge Server.

Fortunately, it is relatively straightforward to implement the calculation of moving averages using Pulsar Functions. As you can see from Listing 12.1, the key is using a circular buffer to retain the last N values required to calculate the moving average. You can then use this Pulsar Function to preprocess the individual sensor readings and publish the calculated SMA rather than the raw value itself. This ensures that all downstream analysis is performed on less noisy data.

12.4.2 Statistical Analysis

Moving averages aren't the only meaningful statistic that can be calculated from univariate datasets. In fact, it is quite easy to calculate just about any of the statistics that are commonly used in statistical analysis using a Pulsar Function. Take, for example, the Function code shown in Listing 12.2 which computes the following statistics in a single function: geometric mean, population variance, kurtosis, root mean square deviation, skewness, and the standard deviation.

Listing 12.2 A Pulsar Function to Calculate Multiple Statistics

```

import org.apache.commons.math3.stat.descriptive.DescriptiveStatistics;
import org.apache.commons.math3.stat.descriptive.SummaryStatistics;
import org.apache.commons.math3.stat.regression.*;      #A
import org.apache.pulsar.functions.api.Context;
import org.apache.pulsar.functions.api.Function;

public class TimeSeriesSummaryFunction implements
    Function<Collection<Double>, SensorSummary> {

    @Override
    public SensorSummary process(Collection<Double> input, Context context)
        throws Exception {      #B

        double[][] data = convert.ToDoubleArray(input);      #C
        SimpleRegression reg = calcSimpleRegression(data);
        SummaryStatistics stats = calcSummaryStatistics(data);
        DescriptiveStatistics dstats = calcDescriptiveStatistics(data);
        double rmse = calculateRSME(data, reg.getSlope(), reg.getIntercept());

        SensorSummary summary =
            SensorSummary.newBuilder()
                .setStats(TimeSeriesSummary.newBuilder()
                    .setGeometricMean(stats.getGeometricMean())      #D
                    .setKurtosis(dstats.getKurtosis())
                    .setMax(stats.getMax())

```

```

        .setMean(stats.getMean())
        .setMin(stats.getMin())
        .setPopulationVariance(stats.getPopulationVariance())
        .setRmse(rmse)
        .setSkewness(dstats.getSkewness())
        .setStandardDeviation(dstats.getStandardDeviation())
        .setVariance(dstats.getVariance())
        .build());
    .build();

    return summary;
}

private SimpleRegression calcSimpleRegression(double[][] input) {
    SimpleRegression reg = new SimpleRegression();
    reg.addData(input);
    return reg;
}

private SummaryStatistics calcSummaryStatistics(double[][] input) {
    SummaryStatistics stats = new SummaryStatistics();
    for(int i = 0; i < input.length; i++) {
        stats.addValue(input[i][1]);
    }
    return stats;
}

private DescriptiveStatistics calcDescriptiveStatistics(double[][] input) {
    DescriptiveStatistics dstats = new DescriptiveStatistics();
    for(int i = 0; i < input.length; i++) {
        dstats.addValue(input[i][1]);
    }
    return dstats;
}

private double calculateRSME(double[][] input, double slope, double intercept) {
    double sumError = 0.0;
    for (int i = 0; i < input.length; i++) {
        double actual = input[i][1];
        double indep = input[i][0];
        double predicted = slope*indep + intercept;
        sumError += Math.pow((predicted - actual),2.0);
    }
    return Math.sqrt(sumError/input.length);
}

private double[][] convert.ToDoubleArray(Collection<Double> in)
throws Exception {
    double[][] newIn = new double[in.size()][2];
    int i = 0;
    for (Double d : in) {
        newIn[i][0] = i;
        newIn[i][1] = d;
        i++;
    }
    return newIn;
}
}

```

```
#A This function relies on external libraries to perform the statistical calculations  
#B The input collection contains all of the sensor readings within the specified window  
#C Convert the data from a collection to a two-dimensional array  
#D Use the various computed statistics to populate the result
```

Obviously, these statistics cannot be computed from just a single datapoint in the time-series data, but rather need to be computed from a larger collection of data elements. Attempting to calculate the geometric mean of a single number makes no sense. Therefore, we need to specify a strategy for splitting the endless stream of metric data into finite sets of data known as *windows* that we will use to calculate these statistics. So how do we go about defining the boundaries of a data window? Within Pulsar Functions there are two policies used to control window boundaries:

- *Trigger Policy*: Controls when our function code is executed. These are the rules that the Apache Pulsar Function framework uses to notify our code that it is time to process all of the data collected in the window.
- *Eviction Policy*: Controls the amount of data retained in the window. These are the rules used to decide if a data element should be retained or evicted from the window.

Both of these policies are driven by either time or the quantity of data in the window and can be defined in terms of time or length (number of data elements). Let's explore the distinction between these two policies and how they work in concert with one another. While there are a variety of windowing techniques, the most prominent ones are tumbling and sliding windows.

Tumbling windows are contiguous, non-overlapping windows that are either of fixed-size, such as 100 elements, or taken at fixed-intervals, such as every 5-minutes. The eviction policy for tumbling windows is *always* disabled in order to allow the window to become completely full. Therefore, you only need to specify the trigger policy you want to use, either count-based or time-based. Figure 12.9 shows the behavior of a tumbling window with a length-based trigger policy set to 10, which means that at the point in time at which 10 items are in the window, the Pulsar Function code will be executed, and the window will be cleared. This behavior is irrespective of time, whether it takes 5 seconds or 5 hours for the window count to reach 10 items doesn't matter, only when the count reaches the specified length matters. Therefore, the first window takes approximately 15 seconds to become full, while the last one requires 25 seconds to elapse before the window fills up.

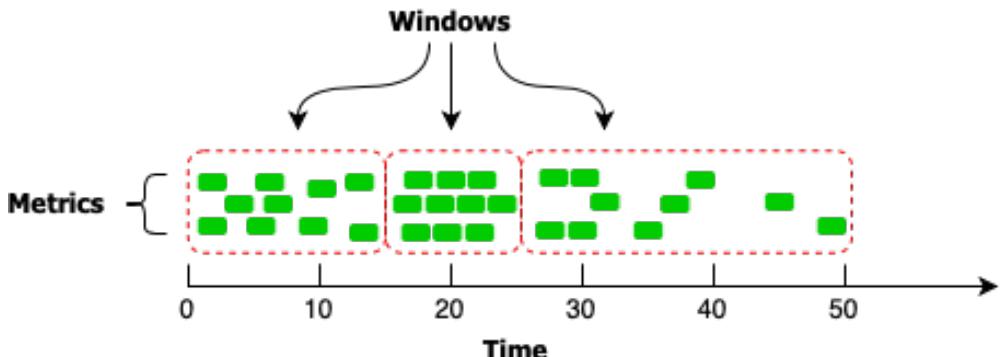


Figure 12.9: When using a count-based trigger, each tumbling window will contain the exact same number of metrics but may span different time durations of time.

The sliding window technique on the other hand, utilizes a combination of a trigger policy that defines the sliding interval and an eviction policy that limits the amount of data retained within the window for processing. Figure 12.10 shows the behavior of a sliding window with the eviction policy configured to be 20 seconds meaning that any data older than 20 seconds will not be retained or used in the calculation. The trigger policy is also configured to be 20 seconds, which means that every 20 seconds the associated Pulsar Function code would be executed, and we would have access to all of the data within the entire window length to perform our calculation.

Thus, in the scenario shown in Figure 12.10, the first window contained 15 events, while the last one contained only two. In this example, both the eviction and trigger policies were defined in terms of time, however it is also possible to define one or both of these in terms of length instead. Additionally, the window length and sliding interval don't have to be the exact same value either. For instance, if you wanted to perform the SMA using this technique, you could achieve that by setting the window length to 100 and the sliding interval to 1.

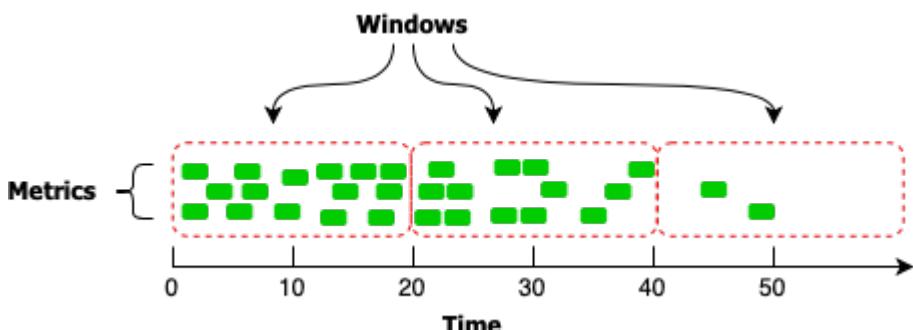


Figure 12.10: When using a time-based sliding window, each window will span the exact same amount of time and will most likely contain different numbers of metrics.

Once these statistical summaries have been calculated, they can be sent upstream to the edge servers in order to be compared to the stats computed for similar equipment within the industrial infrastructure to detect potential problems, or they could be forwarded to a database within the cloud for longer-term storage and future analysis. Having performed the calculations before storing the data in the database will save us from having to recalculate these values during the analysis phase, which can be an expensive operation.

Pulsar Functions provides the four different windowing configuration parameters shown in Table 12.1, which enables you to implement all four variations of the windows discussed in this section when used in the proper combinations.

Table 12.1: Configuring Windowing for Pulsar Functions

Time-Based Tumbling Window	--windowLengthDurationMS==xxx
Length-Based Tumbling Window	--windowLengthCount==xxx
Time-Based Sliding Window	--windowLengthDurationMS==xxx --slidingIntervalDurationMs==xxx
Length-Based Sliding Window	--windowLengthCount==xxx --slidingIntervalCount==xxx

Implementing either of these types of windowing functions in Pulsar Functions is straightforward and only requires that you specify a `java.util.Collection` as the input type. Therefore, if we wanted to run the Pulsar Function shown in Listing 12.2 to perform the statistic calculation on a sliding window of data, then all that we would have to do is submit it using the command shown in Listing 12.3 and the Pulsar Functions framework will handle the rest for us.

Listing 12.3 Performing the statistical calculation on a sliding window of data

```
$ bin/pulsar-admin functions create \  
--jar edge-analytics-functions-1.0.0.nar \  
--classname com.manning.pulsar.iiot.analytics.TimeSeriesSummaryFunction \  
--windowLengthDurationMS==20000 \ #A  
--slidingIntervalDurationMs=20000. #B
```

#A Defines an eviction policy of 20 seconds
#B Defines a trigger policy of 20 seconds

This makes it easy to utilize either of these windowing techniques without having to write the significant amount of boilerplate code necessary to perform the collection and retention of the individual events, and allows you to write clean code the focuses solely on the business logic you are trying to implement.

12.4.3 Approximation

When analyzing streaming data, there are certain types of queries that cannot be computed on the edge because they require huge amounts of both computing resources and time to generate exact results. Examples include count distinct, quantiles, most-frequent items, joins, matrix computations, and graph analysis. Therefore, these types of calculations are typically not performed on these streaming datasets at all.

However, if an approximate answer will suffice, then there is a specialized category of streaming algorithms for providing approximate values, estimates, and data samples for statistical analysis when the event stream is either too large to store in memory or the data is moving too fast to process.

These streaming algorithms utilize small data structures, known as sketches, which are usually only a few kilobytes in size to store the information. Sketch-based algorithms perform "one-touch" processing, meaning they only need to read each element in the stream once. Both these properties make these algorithms ideal candidates for deployment on edge devices. There are four families of sketching algorithms, each focused on solving a different type of problem:

- *Cardinality Sketches*: Provides approximate counts for each distinct value in the stream such as the number of page views across a number of different web pages in a given timeframe.
- *Frequent Item Sketches*: Provides a list of the most frequently seen value in the stream such as the top 10 web pages viewed in a given timeframe
- *Sampling Sketches*: uses reservoir sampling in order to provide uniform random sample of data from the stream that can be used for analysis.
- *Quantile Sketches*: Provides a frequency histogram that contains information about the distribution of the data stream values that can be used for anomaly detection.

There is an open-source library called Apache DataSketches that contains implementations of these algorithms in Java, which makes them easy to use inside a Pulsar Function such as the one shown in Listing 12.4 that uses the quantile sketch to detect anomalies in the sensor data.

QUANTILES

The word quantile is derived from the word quantity and simply refers to equal-sized groupings of something, typically a probability distribution or a series of observed values. You are already familiar with some of the more common quantiles when they are referred to by their more common terms such as halves, thirds, quarters, etc. In statistics and probability, quantiles are more formally defined as cut points that divide a probability distribution into continuous, adjacent intervals with an equal number of elements.

For example, in Figure 12.11 there are three values; Q1, Q2, and Q3 that splits the dataset into equally weighted sections. The areas under the graph between each of these points are the quantiles, in this case thirds, and contain an equal number of data points.

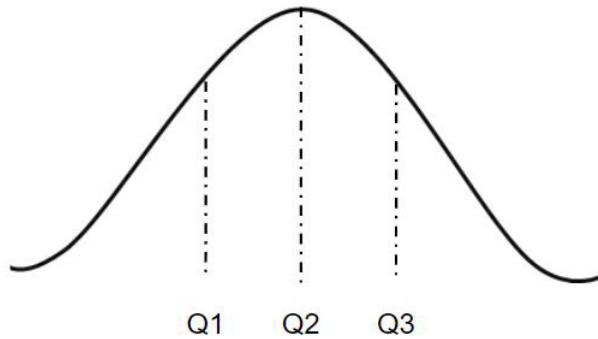


Figure 12.11: A dataset that is divided into three equal sized sections known as quantiles. Values less than or equal to Q1 are considered part of the first quantile, values between Q1 and Q2 are part of the second quantile, and those greater than or equal to Q3 are part of the third quantile.

For our use case, we will be processing an endless stream of metric data values and using them to dynamically construct a quantile. This allows us to make our decisions based solely upon the actual observed values when determining whether or not a value is anomalous or not by comparing to previous values of the same metric. When a new metric reading comes into our Pulsar Function, we will first add it to the quantile to update the distribution model, then we will calculate the *rank* of the metric reading. The rank is best described as the proportion of values in the distribution that the given value is greater than or equal to. For instance, if a metric reading is higher than 79% of the previously observed values then its rank would be 79. Ranking a value helps us determine whether a given metric reading is commonplace or not, and we have decided to use a configurable value to define the threshold that we consider anomalous.

Listing 12.4: A Pulsar Function for Anomaly Detection

```
import org.apache.datasketches.quantiles.DoublesSketch;
import org.apache.datasketches.quantiles.UpdateDoublesSketch;    #A
...
public class AnomalyDetector implements Function<Double, Void> {

private UpdateDoublesSketch sketch;
private double alertThreshold;
private boolean initialized = false;

@Override
public Void process(Double input, Context ctx) throws Exception {
    if (!initialized) {
        init(ctx);
    }
    sketch.update(input);      #B

    if (sketch.getRank(input) >= alertThreshold) {      #C
        react();    #D
    }
    return null;
}
```

```

}

protected void init(Context ctx) {
    sketch = DoublesSketch.builder().build();
    alertThreshold = (double) ctx.getUserConfigValue ("threshold");
    initialized = true;
}

protected void react() {
    // Implementation specific      #E
}
}

#A This function relies on the DataSketches library to perform the statistical calculations
#B Add the metric reading to the sketch
#C Get the metric reading's rank value and compare that to the alert threshold
#D If the metric is above the configured threshold then react
#E This will vary based on the LwM2M protocol being used by the actuator.

```

The logic for the function is fairly straight-forward. First, we update the quantile sketch by adding the data element to it. Then we request the relative rank of the values in the overall distribution. Next, we determine whether or not the value is an outlier or not by comparing the metric reading's rank to the preconfigured alert threshold. If an outlier is detected, then we use a LwM2M client to send a message to an actuator in the perception and reaction layer to perform some sort of preventative action such as turning off a machine or opening a pressure valve, etc. The logic of the react function shown in Listing 12.4 will vary based on the LwM2M protocol being used and the command(s) we need to send in response to the event.

12.5 Multivariate Analysis

Thus far we have implemented various analytical techniques on data from a single sensor. While this univariate analysis does enable us to perform anomaly detection and trend analysis on a single sensor, much more interesting analysis can be performed once you have combined the data from multiple sensors as shown in Figure 12.7. In this section, I will provide an outline of the steps required to combine data collected from different IoT Gateways, analyze it, and respond to these new insights.

12.5.1 Creating a Bi-Directional Messaging Mesh

Creating a messaging framework that can be used to transmit messages up from the IoT Gateways to the Edge Servers involves an initial configuration phase to add all of the Pulsar clusters to the same Pulsar Instance. As you may recall from Chapter 2, a Pulsar Instance can contain multiple Pulsar clusters. Being part of the same Pulsar instance is also a prerequisite for enabling geo-replication of data between Pulsar clusters, which as you saw in Figure 12.8 the preferred mechanism for forwarding the calculated statistical sensor data from the IoT Gateways to the edge server.

The first phase of this configuration is adding each of the individual IoT Gateway-based Pulsar clusters to the same Pulsar Instance as the Pulsar cluster running on the edge server. This can be easily achieved using the Pulsar admin command line interface or REST API as

shown in Listing 12.5, which shows the command used to add a single IoT Gateway cluster to the Pulsar Instance.

Listing 12.5 Adding an IoT Gateway Cluster to the Pulsar Instance

```
$ pulsar-admin clusters create iot-gateway-1 \
    --broker-url http://<IoT-Gateway-IP>:6650 \
    --url http://<IoT-Gateway-IP>:8080      #C
$ pulsar-admin clusters list      #D
```

#A Each name has to be unique

#B URL address of the TCP broker on the Gateway

#C Service URL for the Gateway

#D Confirm that the cluster was added to the list

This command has to be run just once for every IoT Gateway-based Pulsar cluster in your IIoT environment. Once a cluster has been added to the Pulsar Instance, it is able to have messages delivered to it asynchronously via Pulsar's geo-replication mechanism rather than having to write additional code to perform the data replication.

The next step in establishing geo-replication between the IoT Gateways and the Edge Servers is to define a tenant that can be used for bi-directional communication. This can be easily achieved using the Pulsar admin command line interface or REST API as shown in Listing 12.6, which shows the command create a new Pulsar tenant that can be accessed by all of the IoT Gateway clusters as well as the Pulsar cluster running on the Edge Servers.

Listing 12.6 Creating a Geo-Replicated Tenant

```
$ pulsar-admin tenants create iiot-analytics-tenant \
    --allowed-clusters iot-gateway-1, iot-gateway-2, ... \
    --admin-roles analytics-role      #A
#B
```

#A Provide a complete list of all the IoT Gateway clusters you created

#B Specifies the admin role for this namespace

The last step of the configuration process is the creation of a namespace that can be used specifically for the geo-replication of the data between the IoT Gateways and the Edge Servers. This can be easily achieved using the Pulsar admin command line interface or REST API as shown in Listing 12.7, which shows the command create a new Pulsar namespace that can be accessed by all of the IoT Gateway clusters as well as the Pulsar cluster running on the Edge Servers. Please note that this replicated namespace has to be within the tenant we created previously. (See Appendix B for details on configuring geo-replication between Pulsar clusters)

Listing 12.7 Creating a Geo-Replicated Namespace

```
$ pulsar-admin namespaces iiot-analytics-tenant/analytics-namespace \
    --clusters iot-gateway-1, iot-gateway-2, ...      #A
```

#A Provide a complete list of all the IoT Gateway clusters you created

Once you create a geo-replicated namespace, any topics that producers or consumers create within that namespace are automatically replicated across all of clusters. Therefore, any messages published to topics within the geo-replicated namespace on the IoT Gateways will automatically be sent asynchronously to Edge Server. Unfortunately, this would also result in the message also being replicated to all of the other IoT Gateways within the IIoT infrastructure, which is not what we want at all. Not only would this inter-Gateway replication waste precious network bandwidth, but it would also waste disk space on the Gateways themselves as they would have to store these messages that they receive. Since these messages will never be consumed on the other Gateways, storing them just wastes disk space as well.

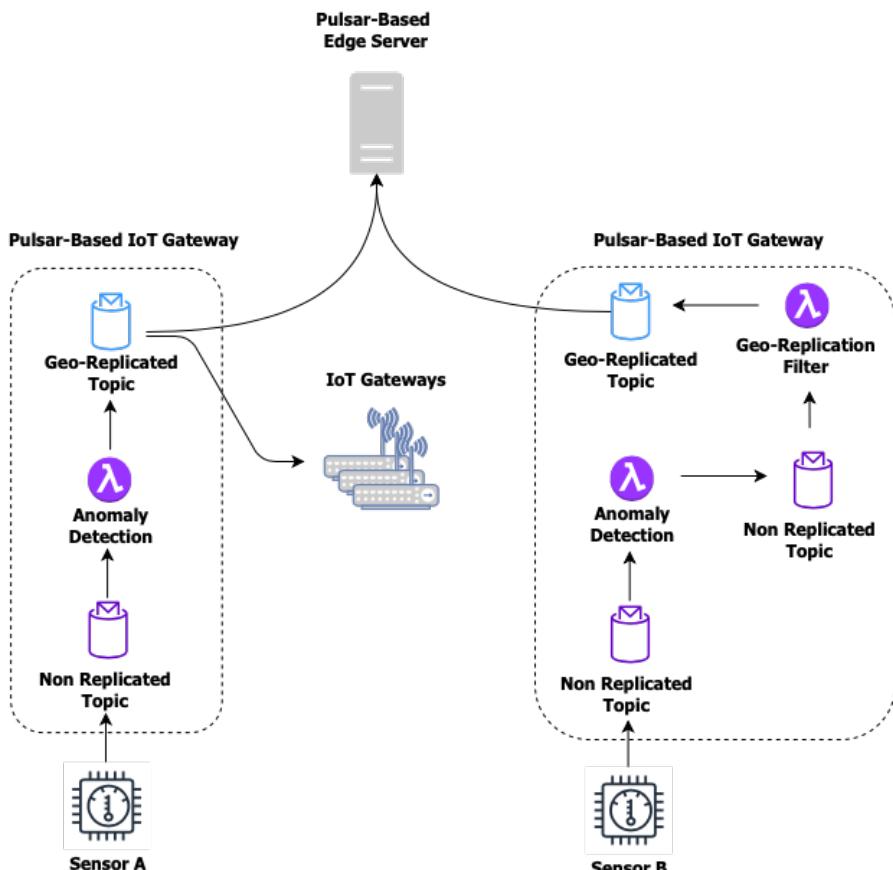


Figure 12.12: Without the use of a geo-replication filter, the data for sensor A is automatically replicated to all of the IoT Gateway clusters in addition to the Edge Server. Whereas with data for sensor B is only replicated to the Pulsar cluster running on the Edge Servers.

Therefore, we need to enable selective replication to ensure that the outbound messages that are only intended to be consumed by the edge server are only replicated to the edge server and not across all of the IoT Gateways. This can be accomplished by writing a simple Pulsar Function that consumes from a local, non-geo-replicated topic on the gateway and publishes it to a geo-replicated topic but restricts the replication to only the Edge Servers as shown on the left side of Figure 12.12

As you can see from Listing 12.8, the Pulsar Function used to forward these messages would use the existing Producer Java API to restrict the replication of the messages to only the Edge Cluster. Resulting in a message flow shown on the right side of Figure 12.11, where the message first goes to a local topic before getting forwarded to a geo-replicated topic but only to the Edge Server rather than all of the clusters.

Listing 12.8 Geo-Replication Message Filter Function

```
public class GeoReplicationFilterFunction implements Function<byte[],Void> {

    private boolean initialized = false;
    private List<String> restrictReplicationTo;

    private Producer<byte[]> producer;
    private PulsarClient client;
    private String serviceUrl;
    private String topicName;

    @Override
    public Void process(byte[] input, Context ctx) throws Exception {
        if (!initialized) {
            init(ctx);
        }
        getProducer().newMessage()
            .value(input)          #A
            .replicationClusters(restrictReplicationTo)      #B
            .send();

        return null;
    }

    private void init(Context ctx) {
        serviceUrl = "pulsar://localhost:6650";      #C
        topicName = ctx.getUserConfigValue("replicated-topic").get().toString();      #D
        restrictReplicationTo = Arrays.asList(
            ctx.getUserConfigValue("edge").get().toString());      #E
        initialized = true;
    }

    private Producer<byte[]> getProducer() throws PulsarClientException {
        if (producer == null) {
            producer = getClient().newProducer()
                .topic(topicName)
                .create();
        }
        return producer;
    }

    private PulsarClient getClient() throws PulsarClientException {
        if (client == null) {
```

```
        client = PulsarClient.builder().serviceUrl(serviceUrl).build();
    }
return client;
}
}
```

```
#A Create a new message using the input bytes
#B Restrict the clusters that the message will be replicated to
#C We are publishing to a local geo-replicated topic
#D The destination topic should be in the replicated namespace
#E This should be the name of the Pulsar cluster running on the Edge Servers
```

The Pulsar Function is writing to a geo-replicated topic on the local machine in order to allow the Pulsar geo-replication mechanism handle the forwarding of the message rather than sending it directly to the Edge Server and avoiding the synchronous call to over the network for each message.

Now that we have covered the “up” direction of the messaging mesh, let’s focus on the “down” direction of the bi-directional mesh which is focused on delivering the insights discovered on the Edge Servers through the analysis of data from multiple sensors back down to the Pulsar Functions running on the IoT Gateways. In reality, this process is fairly straightforward because all that is required is for the Edge Servers to publish to a geo-replicated topic and have the interested parties subscribe to the topic in order to have the messages delivered. Even though the topics are geo-replicated, the messages will *NOT* be sent to the IoT Gateways unless there is an active consumer of the topic on the Gateway node.

12.5.2 Multivariant Dataset Construction

Consider the scenario where you have multiple temperature sensors measuring the ambient temperature across your entire data center. Rather than comparing the current reading of a given individual sensor to just its previous readings, you want see how it compares to the previous readings of all of the other temperature sensors deployed across your data center. Obviously, this would provide a much more meaningful comparison, particularly in the case where the sensor readings gradually drifted higher or lower rather than suddenly. For instance, if one of your pieces of equipment was gradually over-heating and the temperature readings slowly rose from a safe range to a more dangerous level. In such a scenario, there might not be any single reading that is significantly large enough to be considered an anomaly, but compared to other similar sensors, all of the reading were high.

In order to perform a comparison across a much broader set of data would require the combination of data from potentially 100s of different sensors. Therefore, we must first gather all of the data into a single location in order to calculate the statistics for the sensor group as a whole rather than individually. Fortunately, the data sketches used inside the AnomalyDetector Function can be merged together rather easily to provide these types of statistics. A data sketch generated from a series of readings from a single sensor can be used to determine the ranking of any given sensor reading relative to all of the readings recorded in the sketch thus far. But if you combine 100 sketches then you can use the resulting sketch to determine the ranking of any given sensor reading relative to all the readings across 100 sensors. This is a much more meaningful value and would solve the

issue we are trying to address because we are comparing each sensor reading to all others across the entire factory.

In order to achieve this, we must first modify the existing AnomalyDetector Function as shown in Listing 12.9 to periodically send a copy of its sketch to the Edge Servers so that it can be merged with the sketches from all the other BiDirectionalAnomalyDetector Function instances running on different IoT Gateways.

Listing 12.9 Update AnomalyDetection Function

```
public class BiDirectionalAnomalyDetector implements Function<Double, Void> {

    private boolean initialized = false;
    private long publishInterval;      #A
    private String reportingTopic;    #B
    private String alertThresholdTopic; #C
    private double alertThreshold;    #D
    private String remotePulsarUrl;   #E
    private PulsarClient client;
    private Producer<byte[]> producer;   #F
    private Consumer<Double> consumer;  #G
    private ExecutorService service = Executors.newFixedThreadPool(1);      #H
    private ScheduledExecutorService executor =
        Executors.newScheduledThreadPool(1);      #I
    private UpdateDoublesSketch sketch;

    @Override
    public Void process(Double input, Context ctx) throws Exception {
        if (!initialized) {
            init(ctx);
            launchReportingThread();      #J
            launchFeedbackConsumer();    #K
        }

        synchronized(sketch) {      #L
            getSketch().update(input);

            if (getSketch().getRank(input) >= alertThreshold) {
                react();
            }
        }
        return null;
    }

    private void launchReportingThread() {
        Runnable task = () -> {
            synchronized(sketch) {      #M
                try {
                    if (getSketch() != null) {
                        getProducer().newMessage().value(getSketch().toByteArray()).send();
                        sketch.reset();      #N
                    }
                } catch(final PulsarClientException ex) { /* Handle */}
            };
            executor.scheduleAtFixedRate(task,
                publishInterval, publishInterval, TimeUnit.MINUTES);      #O
        };
    }
}
```

```

private void launchFeedbackConsumer() {
    Runnable task = () -> {
        Message<Double> msg;
        try {
            while ((msg = getConsumer().receive()) != null) {      #P
                alertThreshold = msg.getValue();      #Q
                getConsumer().acknowledge(msg);
            }
        } catch (PulsarClientException ex) {/* Handle */ }
    };
    service.execute(task);      #R
}

private UpdateDoublesSketch getSketch() {
    if (sketch == null) {
        sketch = DoublesSketch.builder().build();
    }
    return sketch;
}

private Producer<byte[]> getProducer() throws PulsarClientException {
    if (producer == null) {
        producer = getEdgePulsarClient().newProducer(Schema.BYTES)
            .topic(reportingTopic).create();
    }
    return producer;
}

private Consumer<Double> getConsumer() throws PulsarClientException {
    if (consumer == null) {
        consumer = getEdgePulsarClient().newConsumer(Schema.DOUBLE)
            .topic(alertThresholdTopic).subscribe();
    }
    return consumer;
}

private void react() {
    // Implementation specific
}

private PulsarClient getEdgePulsarClient() throws PulsarClientException {
...
}

protected void init(Context ctx) {
...
}

```

#A Property that defines the how often the local data sketch gets published (in minutes)
#B Property that defines the topic name used to send the local data sketch to the edge server
#C Property that defines the topic name used to receive the updated alert threshold from the edge server
#D Calculated alert threshold provided by the edge server.
#E The URL for the Pulsar Broker running on the edge server.
#F Producer for sending the data sketched to the edge server
#G Consumer for receiving updated alert threshold values from the edge server
#H Local thread pool where the consumer thread can run in the background

```

#I Local thread pool that invokes the publishing thread at a fixed interval, e.g., every 5 minutes
#J Start the data sketch publishing thread just once
#K Start the alert threshold consuming thread just once
#L Create an exclusive lock on the data sketch when writing the data
#M Create an exclusive lock on the data sketch when publishing the data
#N Clear all of the data from the sketch
#O Schedule the publishing task to run at the fixed interval specified by the configuration properties
#P Wait for incoming alert threshold messages
#Q Update the alert threshold with the provided value
#R Launch the alert threshold consuming thread in the background

```

The `BiDirectionalAnomalyDetector` Function still listens for incoming sensor readings and adds them to a local data sketch object before comparing the sensor reading to the anomaly threshold to determine if immediate action must be taken. However, it also creates two additional background threads in order to communicate with the `SketchConsolidator` Function that is running on the Edge Server. This Function relies upon the Java `ScheduledThreadExecutor` to ensure that the local data sketches are published at a periodic interval, e.g., every five minutes. While the other thread is used to continuously monitor the feedback topic for any updates to alert threshold value.

Next, we must create a new Pulsar Function like the one shown in Listing 12.10 that will run on the Edge Servers that will receive these inbound sketches and merge them together to produce a larger, more accurate sketch that encompasses data from all of the sensors within a given sensor family.

Listing 12.10 Data Sketch Merging Function

```

import org.apache.datasketches.memory.Memory;
import org.apache.datasketches.quantiles.DoublesSketch;
import org.apache.datasketches.quantiles.DoublesUnion;
import org.apache.pulsar.functions.api.Context;
import org.apache.pulsar.functions.api.Function;

public class SketchConsolidator implements Function<byte[], Double> {

    private DoublesSketch consolidated;      #A

    @Override
    public Double process(byte[] bytes, Context ctx) throws Exception {
        DoublesSketch iotGatewaySketch =
            DoublesSketch.wrap(Memory.wrap(bytes));      #B

        DoublesUnion union = DoublesUnion.builder().build();      #C
        union.update(iotGatewaySketch);      #D
        union.update(consolidated);      #E
        consolidated = union.getResult();      #F
        return consolidated.getQuantile(0.99);      #G
    }
}

```

```

#A Data sketch that contains data from all of the sensors
#B Convert the incoming bytes to a data sketch
#C Builder a new object used to merge multiple data sketches
#D Add the incoming data sketch to the union object
#E Add the existing consolidated data sketch to the union object

```

```
#F Update the consolidated data sketch to be equal to the result of the merge
#G Publish the newly calculated threshold value for the 99th percentile.
```

The interaction between these two Pulsar Functions is depicted in Figure 12.13 which shows both of the communication channels used to send data up from the BiDirectionalAnomalyDetector Functions running on all of the IoT Gateways to the SketchConsolidator Function running on the Edge Servers and the channel used to send the newly calculated threshold from the SketchConsolidator Function back down to the BiDirectionalAnomalyDetector Function instances running on the IoT Gateways.

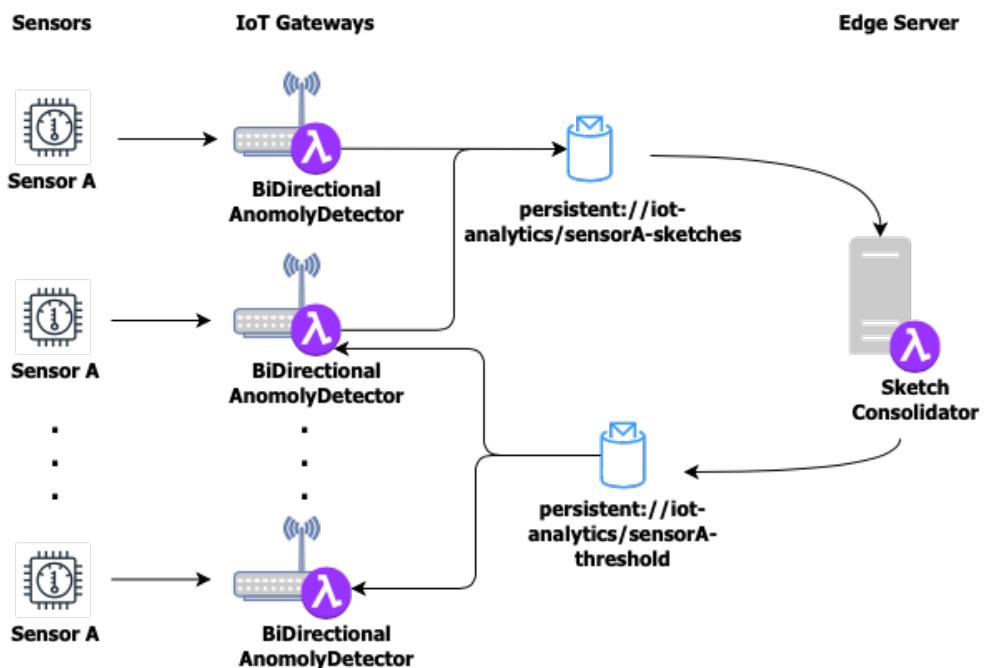


Figure 12.13: A copy of the Bi-Directional Anomaly Detector Function will run on each IoT Gateway and publish their locally calculated data sketches to the same geo-replicated topic. The Sketch Consolidator Function will consume these sketches and merge them together before publishing the cutoff value for the 99th percentile value to a different geo-replicated topic. The Bi-Directional Anomaly Detector Functions will consume messages from this topic and use the published value as the new anomaly threshold for the sensor reading.

The SketchConsolidator Function should be configured to listen to the geo-replicated topic where all of the BiDirectionalAnomalyDetector Functions will be publishing their respective sketches. As you can see from the code in Listing 12.10, once the data sketches have been merged, we use the newly created object to determine the exact value that represents the threshold of the 99th percentile of all sensor readings. We then send this value back to the BiDirectionalAnomalyDetector Functions rather than the entire sketch (to

save space and bandwidth) so that they can use this newly calculated value as their alert threshold instead of the locally calculated threshold.

12.6 Beyond the Book

As I wrap up this final chapter of the book, I hope you have enjoyed reading it and found it to be informative, enlightening, and thought-provoking. It has been a pleasure to interact with many of you throughout the MEAP process via the online discussion forum and I appreciated all of the feedback you have provided. Not only was it great to know that so many of you found the book to be of use, but more importantly how you intend to use Apache Pulsar to harness the power of streaming data within your organizations.

As with all technologies, Apache Pulsar will continue to evolve rapidly thanks to its growing and vibrant developer and user communities. In fact, several new features have been added since I started writing this book such as support for transactions. It is a testament of Pulsar's technological strength to be so widely adopted across a diverse set of companies and industries. However, this evolution will inevitably make the content of the book become increasingly dated over time and thus you should refer to the following resources for the most up-to-date information and new features:

- The Apache Pulsar [project page](#) and [documentation](#).
- The Apache Pulsar slack channel: apache-pulsar.slack.com, which I and several of the project committers monitor on a daily basis. The heavily used channel contains a wealth of information for beginners, and a concentrated community of developers who are actively using Apache Pulsar on a daily basis.
- Several blog posts, including those on the [Splunk web site](#) that are written by many of the committers to the Apache Pulsar project.

For those of you that are looking to introduce stream computing into your organization or considering using a message-based microservices architecture for future applications in order to take advantage of emerging cloud computing paradigm, let me offer the following advice on how to go about convincing your company to consider adopting Apache Pulsar; focus on the benefits of using Pulsar such as the fact that it is a cloud-native technology designed to scale the computing and storage layers independently which leads to more efficient use of expensive cloud resources. Another advantage is the fact it can serve as both a queue-based messaging system like RabbitMQ and a streaming messaging platform like Kafka in a single system.

Another approach is to bring Apache Pulsar in as an underlying technology for a brand new imitative within your organization such as microservices. You can begin your organizations foray into microservices application development using Apache Pulsar Functions as the underlying technology. Your development team will benefit from the simplicity of the programming model Pulsar Functions provides without having to use a proprietary API. If your company is using one of the cloud vendor serverless computing technologies such as AWS Lambdas, you can stress the fact that Pulsar Functions provides the same functionality at a fraction of the cost and without vendor-lock in. Furthermore, all application development and testing can be done locally for free rather than on costly AWS computing resources.

If your organization has an incumbent technology already in place such as Apache Kafka, then you would be wise to heed the advice of Mark Twain when he said, "It's easier to fool people than to convince them that they have been fooled." Which succinctly captures people's reluctance to accept the fact that may have made a bad choice. So rather than framing the conversation as a competition between the incumbent technology and Pulsar in which you must convince your organization that they made a bad decision, you should instead focus on the positives that Pulsar brings to your organization that the other technology cannot. In this way, Pulsar can be seen as a supporting technology that can co-exist within the organization rather than a replacement technology that would require a significant amount of change across the entire organization. Such a wholesale replacement strategy will be met with a large amount of resistance by those who decided upon the incumbent technology and those who invested a lot of time and effort in developing solutions based upon it. You will have a greater chance of success by focusing on Pulsar's strengths (that were covered in Chapter 1) rather than the incumbent technology's weaknesses.

Thank you again for your interest in Apache Pulsar, I hope this book has inspired you to start using Apache Pulsar in some of your projects, and I look forward to interacting with you inside one of the many forums within the Pulsar open-source community!

12.7 Summary

- The amount of time between when an event occurs and when you respond to it is known as the time value of data. This time-value decreases rapidly over time, so being able to respond to it quickly is important.
- Pulsar Functions can be used to provide near real-time analytics on IoT Data within an edge computing environment that consists primarily of resource constrained devices such as IoT Gateways.
- Running Pulsar Functions on these IoT Gateways maximizes the time-value of the data.
- Pulsar's geo-replication mechanism can be used to create bi-directional communication network between Pulsar clusters running on the IoT Gateways and Pulsar clusters running on the Edge Servers.

Appendix A:

Running Pulsar on Kubernetes

Kubernetes is a popular open-source platform for deploying and running containerized applications at scale. It originated inside Google as a solution for managing their extensive infrastructure by automating many of manual processes involved in deploying, managing and scaling their applications across multiple hosts. For more information on Kubernetes, I highly recommend *Kubernetes in Action* by Marko Lukša.

The primary purpose of Kubernetes is to schedule containers to run on a cluster of physical or virtual machines based upon the available computing resources and the resource requirements of every container. A container is simply a ready-to-run software package that contains everything needed to run an application. As we saw in Chapter 3, Docker is one of the most popular container technologies. So naturally Kubernetes can be used to schedule and run Docker containers, including those generated by the Apache Pulsar project. This allows you to run an entire Pulsar cluster, and all of its components such as Zookeeper, BookKeeper, and the Pulsar Proxy entirely on a Kubernetes cluster. This appendix walks you through the process of doing this.

A.1 Create a Kubernetes Cluster

A cluster is the foundational base for running your containerized applications. In Kubernetes, a cluster consists of at least one *cluster master* and multiple worker machines called *nodes*, as shown in Figure A.1. The cluster master machine hosts the Kubernetes control plane, which performs all the administrative functions for the cluster. While the nodes, are the machines that will host the containers themselves.

The computing resources from all of the nodes are registered with cluster master and form a “resource pool” from which all the containers draw from. For instance, if your particular container is hosting a database application and it requires 8GB of RAM and 4 CPU cores, then the cluster master would have to find a node with sufficient resources available to meet this request and run the container on it. These claimed resources would then also be subtracted from the resource pool to indicate that they are already committed to a container.

Once the clusters resource pool is exhausted, no more containers can be hosted until more resources are available.

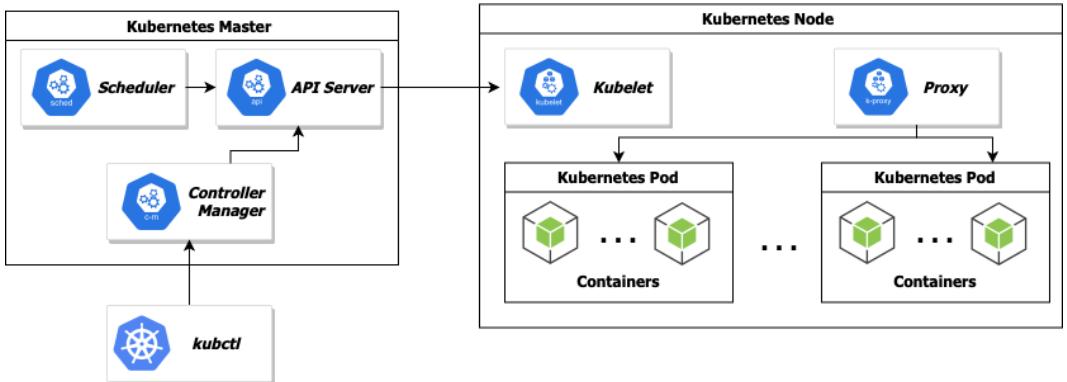


Figure A.1: The Kubernetes master node is used to control all of the Kubernetes nodes in the node pool. Each Kubernetes Node can host multiple Pods, which in turn can host one or more application containers.

With Kubernetes, resources can be easily added by adding more nodes to the cluster. This effectively allows you to scale your cluster up based on your needs in a seamless manner. This feature is so appealing that nearly all cloud-vendors offer some sort of Kubernetes option for hosting your applications. In addition, there is a large open-source implementation of Kubernetes known as OpenShift, which allows you to host a Kubernetes cluster on your own physical hardware. While both of these options are good choices for production applications, they do impose a rather high barrier for local development. Most people don't want to pay the cost of hosting a large Kubernetes cluster simply for development or testing purposes, which is why Minikube is a popular option for developers.

Pulsar was designed specifically to run in a containerized environment such as Kubernetes, where you can easily increase or decrease the number of Pulsar Broker containers and/or BookKeeper bookies based on your demand.

A.1.1 Install Prerequisites

As a prerequisite for working with Kubernetes, you will need to install the Kubernetes command-line tool called *kubectl*, that allows you to run commands against Kubernetes clusters. You will need this tool in order to deploy applications, inspect and manage cluster resources, and view logs. If you don't already have *kubectl* installed, you should [download it](#) and follow the instructions for your operating system.

Listing A.1 Installing kubectl on a Macbook

```
brew install kubectl    #A  
...  
==> Downloading https://homebrew.bintray.com/bottles/kubernetes-cli-  
1.19.1.catalina.bottle.tar.gz    #B
```

```
==> Pouring kubernetes-cli-1.19.1.catalina.bottle.tar.gz
==> Caveats
Bash completion has been installed to:
  /usr/local/etc/bash_completion.d

zsh completions have been installed to:
  /usr/local/share/zsh/site-functions
==> Summary
🍺 /usr/local/Cellar/kubernetes-cli/1.19.1: 231 files, 49MB
```

```
#A Using Homebrew to install kubectl
#B Downloading and installing version 1.19.1
```

If you have a Mac like I do, then you can use the Homebrew package manager to install it using a single line as shown in Listing A.1. If you are using a different operating system, please consult the online documentation for installation instructions specific to your OS. You must use a kubectl version that is within one minor version difference of your cluster. Therefore, it is best to use the latest version of kubectl in order to avoid any compatibility issues.

A.1.2 Minikube

Once kubectl has been installed, the next step is to create a Kubernetes cluster to host the Pulsar cluster. While all of the major cloud vendors provide Kubernetes environments that are well suited for production use, in this appendix I will use a more cost-effective alternative known as Minikube, that allows me to run a Kubernetes cluster on my development machine.

Minikube is a tool that runs a single-node Kubernetes cluster on your personal computer. It is well suited for day-to-day development tasks that require access to a containerized application such as Pulsar. It is a good choice if you want to develop and test your application inside a Kubernetes environment in order to familiarize yourself with the Kubernetes API.

Listing A.2 Installing Minikube on a Macbook

```
brew install minikube    #A
.
.
==> Downloading https://homebrew.bintray.com/bottles/minikube-1.13.0.catalina.bottle.tar.gz
#B
Already downloaded:
  /Users/david/Library/Caches/Homebrew/downloads/b4e7b1579cd54deea3070d595b60b315ff724
  4ada9358412c87ecfd061819d9b--minikube-1.13.0.catalina.bottle.tar.gz
==> Pouring minikube-1.13.0.catalina.bottle.tar.gz
==> Caveats
Bash completion has been installed to:
  /usr/local/etc/bash_completion.d

zsh completions have been installed to:
  /usr/local/share/zsh/site-functions
==> Summary
🍺 /usr/local/Cellar/minikube/1.13.0: 8 files, 62.2MB
```

```
#A Using Homebrew to install minikube
#B Downloading and installing version 1.13.0
```

If you don't already have Minikube installed, you should [download it](#) and follow the instructions for your operating system. If you have a Mac like I do, then you can use the Homebrew package manager to install it using a single line as shown in Listing A.2. If you are using a different operating system, please consult the online documentation for installation instructions specific to your OS. After Minikube has been installed, the next step is to create a Kubernetes cluster using the commands shown in Listing A.3. The first command creates the cluster itself and specifies the resources it will claim from my laptop for its resource pool.

Listing A.3 Creating a Kubernetes cluster using Minikube

```
minikube start \
--memory=8192 \      #A
--cpus=4 \           #B
--kubernetes-version=v1.19.0    #C

kubectl config use-context minikube   #D

#A Reserve 8GB of RAM for the cluster
#B Reserve 4 cores for the cluster
#C Specify the version of Kubernetes we will be using
#D Set kubectl to use Minikube
```

In order for the kubectl tool to find and access a Kubernetes cluster, it must first be configured to point to the Kubernetes cluster that you wish to interact with. This association is controlled by a kubeconfig file, which is created automatically when you deploy a Minikube cluster and is located at `~/.kube/config`. You can use the `kubectl config use-context <cluster-name>` command as shown in Listing A.3 to configure the kubectl tool to point to the newly created Minikube cluster. You can confirm that the kubectl is properly configured by running the `kubectl cluster-info` command, which will return basic information about the Kubernetes cluster.

A.2 The Pulsar Helm Chart

Now that we have a Kubernetes cluster up and running, we can deploy containerized applications on top of it. This can be accomplished with a deployment configuration file that contains all the information needed to create all the containers required by your application. These deployment configuration files are simple YAML files that conform to a specific structure as shown in Listing A.4, which show the configuration for a single Nginx-based web server that listens on port 80 for incoming requests.

Listing A.4 A Kubernetes Deployment Configuration File

```
apiVersion: apps/v1    #A
kind: Deployment      #B
metadata:
  name: mysite       #C
  labels:
    name: mysite
spec:
  replicas: 1        #D
```

```

template:
  metadata:
    labels:
      app: mysite
  spec:
    containers: #E
      - name: mysite
        image: nginx #F
        resources: #G
          limits:
            memory: "128Mi"
            cpu: "500m"
        ports:
          - containerPort: 80 #H

```

#A Specifies the API version of the configuration file
#B Specifies the resource type defined in the configuration file
#C The application name
#D The number of pods to create
#E Specifies all of the containers inside each pod
#F The docker image name to use
#G The resources required for the nginx container
#H The exposed port for the container.

Once you have created this file, you can then use the `kubectl apply -f filename` command to deploy it to your Kubernetes cluster. While this approach is relatively straightforward, it is a bit tedious to have to create and edit all of these verbose files manually. As you can see the deployment file for a simple, single-container application requires 22 lines of YAML. You can just image how big and complex the deployment file is for an application as complex as Pulsar which requires multiple instances of multiple containers (Brokers, Bookies, Zookeeper, etc.) is going to be.

Kubernetes-orchestrated container applications can be complex to deploy. Developers can use incorrect inputs for configuration files or not have the expertise to rollout these apps from YAML templates, etc. Therefore, a deployment tool known as Helm was created to simplify the deployment of containerized applications to Kubernetes.

A.2.1 What is Helm?

Helm is a package manager for Kubernetes that allows developers to easily package, configure, and deploy applications and services onto Kubernetes clusters. It is analogous to Linux package managers such as **yum** or **apt** because they all allow you to deploy a software package, along with all its dependencies with a simple command.

We will be using Helm to install our pulsar cluster, so if you don't already have Helm installed, you should install it now. If you have a Mac like I do, then you can use the Homebrew package manager to install it using a single line as shown in Listing A.5. If you are using a different operating system, please consult the [online documentation](#) for installation instructions specific to your OS.

Listing A.5 Installing Helm on a MacBook

```
brew install helm      #A
...
==> Downloading https://homebrew.bintray.com/bottles/helm-3.3.1.catalina.bottle.tar.gz
#B
Already downloaded:
  /Users/david/Library/Caches/Homebrew/downloads/77e13146a8989356ceaba3a19f6ee6a342427
  d88975394c91a263ae1c35a3eb6--helm-3.3.1.catalina.bottle.tar.gz
==> Pouring helm-3.3.1.catalina.bottle.tar.gz
==> Caveats
Bash completion has been installed to:
  /usr/local/etc/bash_completion.d

zsh completions have been installed to:
  /usr/local/share/zsh/site-functions
==> Summary
🍺 /usr/local/Cellar/helm/3.3.1: 56 files, 40.3MB
```

#A Using Homebrew to install helm

#B Downloading and installing version 3.3.1

Helm allows us to package Kubernetes applications into packages of pre-configured Kubernetes resources known as **Charts**. Helm Charts provide “push button” deployment and deletion of apps, making development and deployment of Kubernetes applications easier for those with little or no container or microservices experience.

ANATOMY OF A HELM CHART

A Helm chart is basically a collection of files inside a directory. The directory name is used as the name of the chart. Within this directory, the helm chart directory contains a self-descriptor file named Chart.yaml, a values.yaml file, and one or more manifest files that are stored in the chart's template folder as shown in Listing A.6.

Listing A.6 The Helm Chart Directory Layout

```
package-name/
  charts/
    templates/    #A
    Chart.yaml    #B
    values.yaml   #C
    requirements.yaml #D
```

#A Folder of manifest files

#B The self-descriptor file

#C Default values used in the templates

#D Optional list of dependencies

The Helm chart uses the YAML templates for application configuration with a separate value.yaml file to store all the values which are injected into the template YAML at the time of installation. Essentially, Helm charts can be thought of as Kubernetes files that can be parameterized.

When your chart is ready for deployment, you can use the `helm package <chartname>` command to create a tar-gzipped file containing all the files. Once all this is packaged into a Helm chart, anyone can use it using the `helm install` command and providing custom values

to the configurations via an external values file or as an argument to the helm install command and those values are used while creating the Kubernetes application by running the `helm install <chartname>` command.

A.2.2 The Pulsar Helm Chart

Now I have covered what Helm charts are, and how they can be used to deploy an entire application. You will be glad to know that there is a Helm chart for Apache Pulsar that is included in the open-source distribution, and you can easily access the chart by cloning the repo using git, as shown in Listing A.7.

Listing A.7 Downloading the Pulsar Helm Chart

```
git clone https://github.com/apache/pulsar-helm-chart    #A  
cd pulsar-helm-chart    #B
```

#A Clone the Helm chart repo

#B Change into the folder that the repo was cloned into

Once you have cloned the repo, you can examine the contents of the Helm chart inside the charts subfolder as shown in Listing A.8. As expected, the directory structure conforms to the Helm directory structure we saw earlier in Listing A.6, with `Chart.yaml` and `values.yaml` files at the base level, along with a directory of template files.

Listing A.8 The Pulsar Helm Chart Directory Layout

```
ls ./charts/pulsar/    #A  
Chart.yaml      templates      values.yaml  
  
ls ./charts/pulsar/templates/*.yaml    #B  
.charts/pulsar/templates/autorecovery-configmap.yaml  
.charts/pulsar/templates/autorecovery-service.yaml  
.charts/pulsar/templates/autorecovery-statefulset.yaml  
.charts/pulsar/templates/bookkeeper-cluster-initialize.yaml  
.charts/pulsar/templates/bookkeeper-configmap.yaml  
.charts/pulsar/templates/bookkeeper-pdb.yaml  
.charts/pulsar/templates/bookkeeper-podmonitor.yaml  
.charts/pulsar/templates/bookkeeper-service.yaml  
.charts/pulsar/templates/bookkeeper-statefulset.yaml  
.charts/pulsar/templates/bookkeeper-storageclass.yaml  
.charts/pulsar/templates/broker-cluster-role-binding.yaml  
.charts/pulsar/templates/broker-configmap.yaml  
.charts/pulsar/templates/broker-pdb.yaml  
.charts/pulsar/templates/broker-podmonitor.yaml  
.charts/pulsar/templates/broker-rbac.yaml  
.charts/pulsar/templates/broker-service-account.yaml  
.charts/pulsar/templates/broker-service.yaml  
.charts/pulsar/templates/broker-statefulset.yaml  
.charts/pulsar/templates/dashboard-deployment.yaml  
.charts/pulsar/templates/dashboard-ingress.yaml  
.charts/pulsar/templates/dashboard-service.yaml  
.charts/pulsar/templates/function-worker-configmap.yaml  
.charts/pulsar/templates/grafana-admin-secret.yaml
```

```

./charts/pulsar/templates/grafana-configmap.yaml
./charts/pulsar/templates/grafana-deployment.yaml
./charts/pulsar/templates/grafana-ingress.yaml
./charts/pulsar/templates/grafana-service.yaml
./charts/pulsar/templates/keytool.yaml
./charts/pulsar/templates/namespace.yaml
./charts/pulsar/templates/prometheus-configmap.yaml
./charts/pulsar/templates/prometheus-deployment.yaml
./charts/pulsar/templates/prometheus-pvc.yaml
./charts/pulsar/templates/prometheus-rbac.yaml
./charts/pulsar/templates/prometheus-service.yaml
./charts/pulsar/templates/prometheus-storageclass.yaml
./charts/pulsar/templates/proxy-configmap.yaml
./charts/pulsar/templates/proxy-ingress.yaml
./charts/pulsar/templates/proxy-pdb.yaml
./charts/pulsar/templates/proxy-podmonitor.yaml
./charts/pulsar/templates/proxy-service.yaml
./charts/pulsar/templates/proxy-statefulset.yaml
./charts/pulsar/templates/pulsar-cluster-initialize.yaml
./charts/pulsar/templates/pulsar-manager-admin-secret.yaml
./charts/pulsar/templates/pulsar-manager-configmap.yaml
./charts/pulsar/templates/pulsar-manager-deployment.yaml
./charts/pulsar/templates/pulsar-manager-ingress.yaml
./charts/pulsar/templates/pulsar-manager-service.yaml
./charts/pulsar/templates/tls-cert-internal-issuer.yaml
./charts/pulsar/templates/tls-certs-internal.yaml
./charts/pulsar/templates/toolset-configmap.yaml
./charts/pulsar/templates/toolset-service.yaml
./charts/pulsar/templates/toolset-statefulset.yaml
./charts/pulsar/templates/zookeeper-configmap.yaml
./charts/pulsar/templates/zookeeper-pdb.yaml
./charts/pulsar/templates/zookeeper-podmonitor.yaml
./charts/pulsar/templates/zookeeper-service.yaml
./charts/pulsar/templates/zookeeper-statefulset.yaml
./charts/pulsar/templates/zookeeper-storageclass.yaml

```

#A Examine the structure of generated Pulsar Helm chart directory
#B List all of the generated templates

As we can see from Listing A.8 there are quite a few templates that encapsulate the bulk of the chart logic. Let's examine the templates associated with the Pulsar Brokers to get a better understanding of the details these templates contain.

Listing A.9 The Pulsar Broker Deployment Configuration File

```

cat ./charts/pulsar/templates/broker-service.yaml    #A
...
{{- if .Values.components.broker --}}
apiVersion: v1
kind: Service
metadata:
  name: "{{ template "pulsar.fullname" . }}-{{ .Values.broker.component }}"
  namespace: {{ .Values.namespace }}
  labels:
    {{- include "pulsar.standardLabels" . | nindent 4 --}}
      component: {{ .Values.broker.component }}
  annotations:

```

```

{{ toYaml .Values.broker.service.annotations | indent 4 }}
spec:
  ports:
    # prometheus needs to access /metrics endpoint
    - name: http
      port: {{ .Values.broker.ports.http }}      #B
      {{- if or (not .Values.tls.enabled) (not .Values.tls.broker.enabled) --}}
    - name: pulsar
      port: {{ .Values.broker.ports.pulsar }}      #C
      {{- end --}}
      {{- if and .Values.tls.enabled .Values.tls.broker.enabled }}      #D
    - name: https
      port: {{ .Values.broker.ports.https }}      #E
    - name: pulsarssl
      port: {{ .Values.broker.ports.pulsarssl }}      #F
      {{- end --}}
    clusterIP: None
    selector:
      app: {{ template "pulsar.name" . }}
      release: {{ .Release.Name }}
      component: {{ .Values.broker.component }}
    {{- end --}}

```

#A The file containing the Pulsar Broker service definition

#B The HTTP port to use

#C The data port to use

#D Whether the Broker should use TLS or not

#E The secured HTTPS port to use

#F The secured data port to use

As you can see from Listing A.9, the Pulsar Broker definition file depends upon parameterized values for configuration. As you may suspect, these values are provided in the `values.yaml` file that was generated for us when we ran the script to produce the Pulsar Helm chart. Listing A.10 shows the corresponding section of the `values.yaml` file that contains the definitions for the Pulsar Broker.

Listing A.10 The Pulsar Broker Related Values in `values.yaml`

```

## Pulsar: Broker cluster
## templates/broker-statefulset.yaml
##
broker:
  # use a component name that matches your grafana configuration
  # so the metrics are correctly rendered in grafana dashboard
  component: broker
  replicaCount: 3      #A
  # If using Prometheus-Operator enable this PodMonitor to discover broker scrape targets
  # Prometheus-Operator does not add scrape targets based on k8s annotations
  podMonitor:
    enabled: false
    interval: 10s
    scrapeTimeout: 10s
  ports:      #B
    http: 8080
    https: 8443
    pulsar: 6650
    pulsarssl: 6651

```

```

# nodeSelector:
  # cloud.google.com/gke-nodepool: default-pool
...
  resources:    #C
  requests:
    memory: 512Mi
    cpu: 0.2
## Broker configmap
## templates/broker-configmap.yaml      #D
##
configData:
  PULSAR_MEM: >
    -Xms128m -Xmx256m -XX:MaxDirectMemorySize=256m    #E
  PULSAR_GC: >
    -XX:+UseG1GC
    -XX:MaxGCPauseMillis=10
    -Dio.netty.leakDetectionLevel=disabled
    -Dio.netty.recycler.linkCapacity=1024
    -XX:+ParallelRefProcEnabled
    -XX:+UnlockExperimentalVMOptions
    -XX:+DoEscapeAnalysis
    -XX:ParallelGCThreads=4
    -XX:ConcGCThreads=4
    -XX:G1NewSizePercent=50
    -XX:+DisableExplicitGC
    -XX:-ResizePLAB
    -XX:+ExitOnOutOfMemoryError
    -XX:+PerfDisableSharedMem    #F
  managedLedgerDefaultEnsembleSize: "2"      #G
  managedLedgerDefaultWriteQuorum: "2"        #H
  managedLedgerDefaultAckQuorum: "2"          #I

```

#A Specifies a total of three Broker instances

#B Section that specifies the various port values

#C Section that specifies the pod resources

#D The associated broker configuration map

#E The JVM memory settings for the Broker pods

#F The JVM garbage collection settings for the Broker pods

#G The ensemble size for the Pulsar ledger

#H The write quorum size for the Pulsar ledger

#I The ack quorum for the Pulsar ledger.

As you can see from Listing A.10, these settings are on the small side in terms of resources. This is because the default Pulsar Helm chart is designed specifically for Minikube based deployment. You can modify these values to suit your own needs.

A.3 Using the Pulsar Helm Chart

Now that we have downloaded and examined the Pulsar Helm chart, the next step is to use it to provide our Pulsar cluster. The first step in this process is to add the Pulsar Helm chart to your local Helm repository and initialize it as shown in Listing A.11. This will allow your local Helm client to locate and download the Pulsar Helm chart.

Listing A.11 Adding the Pulsar Helm Chart to your Helm Repository

```
helm repo add apache https://pulsar.apache.org/charts    #A

./scripts/pulsar/prepare_helm_release.sh \
--create-namespace \    #B
--namespace pulsar \   #C
--release pulsar-mini   #D

namespace/pulsar created
generate the token keys for the pulsar cluster    #E
The private key and public key are generated to
    /var/folders/zw/x39hv0dd7133w9v9cgnt1lvr0000gn/T/tmp.QT3EjywR and
    /var/folders/zw/x39hv0dd7133w9v9cgnt1lvr0000gn/T/tmp.YkhhbAyG successfully.
secret/pulsar-mini-token-asymmetric-key created
generate the tokens for the super-users: proxy-admin,broker-admin,admin
generate the token for proxy-admin
secret/pulsar-mini-token-proxy-admin created
generate the token for broker-admin
secret/pulsar-mini-token-broker-admin created
generate the token for admin
secret/pulsar-mini-token-admin created    #F
-----

The jwt token secret keys are generated under:    #G
- 'pulsar-mini-token-asymmetric-key'

The jwt tokens for superusers are generated and stored as below:    #H
- 'proxy-admin':secret('pulsar-mini-token-proxy-admin')
- 'broker-admin':secret('pulsar-mini-token-broker-admin')
- 'admin':secret('pulsar-mini-token-admin')
```

#A Add the Pulsar Helm repo to your local Helm repo

#B Instruct Helm to create the Kubernetes namespace

#C The name of the Kubernetes namespace to create

#D The Pulsar release name

#E Generated the public and private token files

#F Generated the tokens for the various admin users

#G Generated the JWT secret

#H Generated the JWT access tokens.

The final step in the process is to use Helm to install the Pulsar cluster as shown in Listing A.12. It is important to specify initialize=true when installing a Pulsar release for the first time because it will ensure that the cluster metadata for both BookKeeper and Pulsar is properly initialized.

Listing A.12 Install Pulsar using the Helm Chart

```
helm install \
--set initialize=true \    #A
--values examples/values-minikube.yaml \    #B
pulsar-mini \    #C
apache/pulsar    #D

kubectl get pods -n pulsar -o name    #E
pod/pulsar-mini-bookie-0
pod/pulsar-mini-bookie-init-94r5z
```

```
pod/pulsar-mini-broker-0
pod/pulsar-mini-grafana-6746b4bf69-bjtff
pod/pulsar-mini-prometheus-5556dbb808-m8287
pod/pulsar-mini-proxy-0
pod/pulsar-mini-pulsar-init-dmztl
pod/pulsar-mini-pulsar-manager-6c6889dff-q9t5q
pod/pulsar-mini-toolset-0
pod/pulsar-mini-zookeeper-0
```

#A Request that the cluster metadata be initialized

#B The values file to use

#C The unique name for this cluster

#D The Helm chart to use

#E List all the pods created for the Pulsar cluster

After Helm has completed the installation process, you can use the `kubectl` tool to list all of the pods created for the Pulsar cluster and validate that the necessary services are up and running, get the IP addresses, etc.

A.3.1 Administering Pulsar on Kubernetes

Once you have deployed a Pulsar cluster to a Kubernetes environment, one of your first concerns will be how to administer the pulsar cluster. Fortunately, the Pulsar Helm chart creates a pod named `pulsar-mini-toolset-0` that contains the `pulsar-admin` CLI tool which is already configured to interact with the deployed Pulsar cluster.

Consequently, all that is required to administer the cluster is to use the `kubectl exec` command to access the pod and execute the commands directly against the cluster as shown in Listing A.13.

Listing A.13 Administering Pulsar on Kubernetes

```
kubectl exec -it -n pulsar pulsar-mini-toolset-0 /bin/bash      #A
bin/pulsar-admin tenants create manning      #B
bin/pulsar-admin tenants list      #C
"manning"
"public"
"pulsar"
```

Since the `pulsar-admin` CLI tool is the same for both the Kubernetes cluster and the Docker standalone container, the `docker exec` and `kubectl exec` commands can be used interchangeably throughout this book if you choose to follow the examples using Kubernetes rather than Docker. For more details on the `pulsar-admin` CLI, please refer to the documentation.

A.3.2 Configuring Clients

The main challenge with connecting to a Pulsar cluster inside a K8s environment is finding the ports that the cluster is listening on. The default binary port, 6650 and HTTP admin port,

8080 are not exposed outside of the K8s environment. Therefore, you need first determine where these node ports are mapped to.

By default, the Pulsar Helm chart exposes the Pulsar cluster through a Kubernetes Load Balancer. In Minikube, you can use the command shown in Listing A.14 to check the proxy service. This output from this command will tell us which node ports that Pulsar cluster's binary port and HTTP port are mapped to. The port after 80: is the HTTP port while the port after 6650: is the binary port.

Listing A.14 Determining the Pulsar Client Ports

```
$kubectl get services -n pulsar | grep pulsar-mini-proxy    #A  
pulsar-mini-proxy           LoadBalancer   10.110.67.72      <pending>  
  80:30210/TCP,6650:32208/TCP   4h16m       #B  
  
$minikube service pulsar-mini-proxy -n pulsar --url     #C  
http://192.168.64.3:30210    #D  
http://192.168.64.3:32208    #E
```

#A Command to determine port mappings

#B The output tells us port 80 is mapped to port 30210, and port 6650 is mapped to port 32208

#C Command to find the IP address of the exposed ports inside minikube

#D The Proxy's HTTP URL

#E The Proxy's binary URL

At this point, you have service URL's you need to connect your clients to the Pulsar cluster running inside Minikube, and you can use them, along with required security tokens that we generated earlier when configuring your Pulsar clients to interact with the cluster.

Appendix B:

Geo-Replication

Geo-replication is a common mechanism used to provide disaster recovery in multi-datacenter deployments. Unlike other pub-sub messaging systems that require additional processes to mirror messages between data centers, geo-replication is automatically performed by Pulsar brokers and can be enabled, disabled, or dynamically changed at runtime. Traditional geo-replication mechanisms typically fall into one of two categories; synchronous and asynchronous. Apache Pulsar comes with multi-datacenter replication as an integrated feature that supports both of these geo-replication strategies. In the following examples, I will assume that we are deploying our Pulsar instance across the three cloud provider regions: **us-west**, **us-central** and **us-east**.

B.1 Synchronous Geo-Replication

A *synchronous* geo-replicated Pulsar installation consists of a cluster of bookies running across multiple regions, a cluster of brokers also distributed across all regions, and single global Zookeeper installation to form a single global logical instance across all available regions as shown in Figure B.1. The global “stretched” Zookeeper ensemble is critical to supporting this approach because it is used to store the managed ledgers.

In the synchronous geo-replication case, when the client issues a write request to a Pulsar cluster in one geographical location, the data is written to multiple Bookies in different geographical locations within the same call. The write request is only acknowledged to the client when the configured number of the data centers have issued a confirmation that the data has been persisted. While this approach provides the highest level of data guarantees it also incurs the cost of the cross-datacenter network latency for each message.

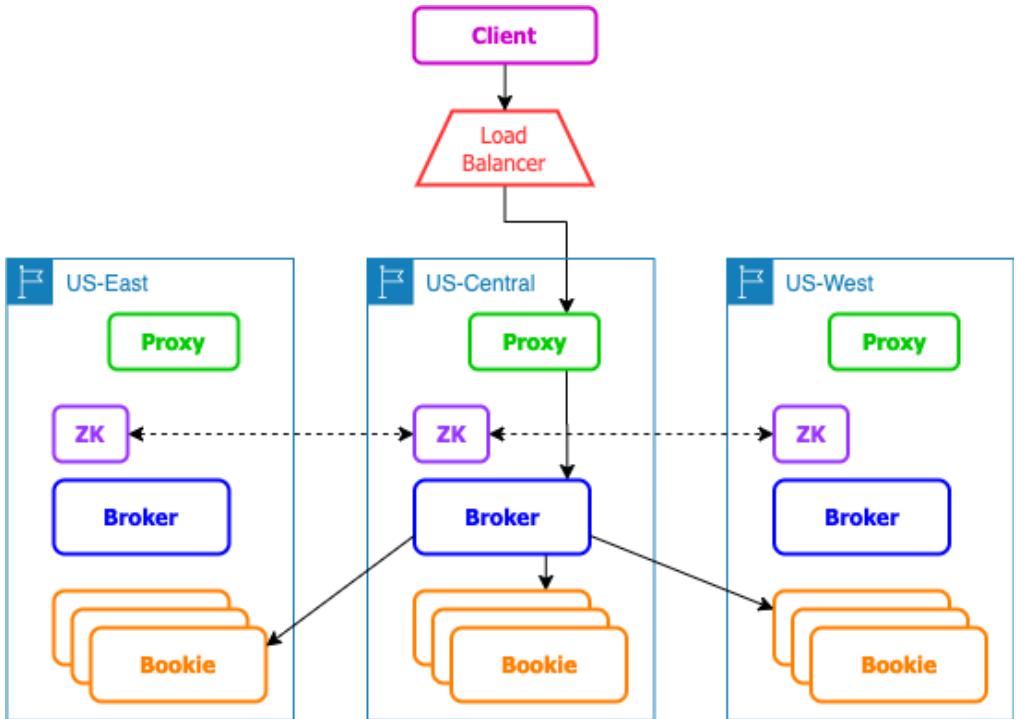


Figure B.1: Clients access a synchronously geo-replicated cluster via a single load balancer, which forward the publish request to one of the Pulsar proxies. The Proxy routes the request to the Broker that owns the topic which then publishes the data across the regions based on the placement policy that is configured.

Synchronous geo-replication is actually achieved by Apache BookKeeper in the storage layer for Pulsar and relies upon a placement policy to distribute the data across multiple data centers and guarantee availability constraints. You can enable either the rack-aware or region-aware placement policy depending on whether you are running in a bare metal or cloud environment respectively by modifying the broker configuration file (`broker.conf`) as shown in Listing B.1.

B.1: Enabling the Region-Aware Policy

```

# Set this to true if your cluster is spread across racks inside one
# datacenter or across multiple AZs inside one region
bookkeeperClientRackawarePolicyEnabled=true

# Set this to true if your cluster is spread across multiple datacenters or
# cloud provider regions.
bookkeeperClientRegionawarePolicyEnabled=true

```

When you enable the region-aware placement policy for example, BookKeeper will choose bookies from different regions when forming a new Bookie ensemble, which ensures that the

topic data will be distributed evenly across all of the available regions. Note that only one of these settings will be honored at runtime, with region-awareness taking precedence if both are set to true.

The use of a single Zookeeper cluster to implement synchronous geo-replication also requires some additional configuration changes in order to have the geographically dispersed Broker and Bookie components work together as a single cluster. Configuring ZooKeeper for such a scenario involves adding a `server.N` line to the `conf/zookeeper.conf` file for each node in the ZooKeeper cluster, where N is the number of the ZooKeeper node as shown in Listing B.2 which uses one Zookeeper node per region.

Listing B.2 Single Zookeeper Configuration for Synchronous Geo-Replication

```
server.1=zk1.us-west.example.com:2888:3888  
server.2=zk1.us-central.example.com:2888:3888  
server.3=zk1.us-east.example.com:2888:3888
```

In addition to modifying the `conf/zookeeper.conf` file in the `conf` directory of each Pulsar installation, you will also need to modify the `zkServers` property in the `conf/bookkeeper.conf` file to list each of the Zookeeper servers as shown in Listing B.3.

Listing B.3 Bookkeeper Configuration for Synchronous Geo-Replication

```
zkServers= zk1.us-west.example.com:2181, zk1.us-central.example.com:2181, zk1.us-east.example.com:2181
```

Similarly, you will need to update the `zookeeperServers` property in BOTH the `conf/discovery.conf` and `conf/proxy.conf` files to be a comma separated list of the Zookeeper servers as well since both the Pulsar Proxy and Service discovery mechanism depend upon Zookeeper to provide them with up-to-date metadata about the Pulsar cluster.

Synchronous geo-replication provides stronger data consistency guarantees than asynchronous replication since the data is always synchronized across the datacenters, making it easier to run your applications independent of where the messages are published. A synchronous geo-replicated Pulsar cluster can continue to function like normal even if an entire datacenter goes down, with the outage being entirely transparent to the applications that are accessing the cluster via a load balancer, etc. This makes synchronous geo-replication good for mission-critical use cases that are able to tolerate a slightly higher publish latency.

B.2 Asynchronous Geo-Replication

An *asynchronous* geo-replicated Pulsar installation consists of a two or more independent Pulsar clusters running in different regions. Each Pulsar cluster contains their own respective set of Brokers, Bookies, and Zookeeper nodes that are completely isolated from one another. In asynchronous geo-replication, when messages are produced on a Pulsar topic, they are first persisted to the local cluster and then replicated asynchronously to the remote clusters. This replication process occurs via inter-Broker communication as shown in Figure B.2

With asynchronous geo-replication the message producer doesn't wait for a confirmation from multiple Pulsar clusters, instead the producer receives a response immediately after the nearest cluster successfully persists the data. The data is then replicated to the other Pulsar clusters, in an asynchronous fashion in the background. Under normal conditions messages are replicated at the same time that they are dispatched to local consumers.

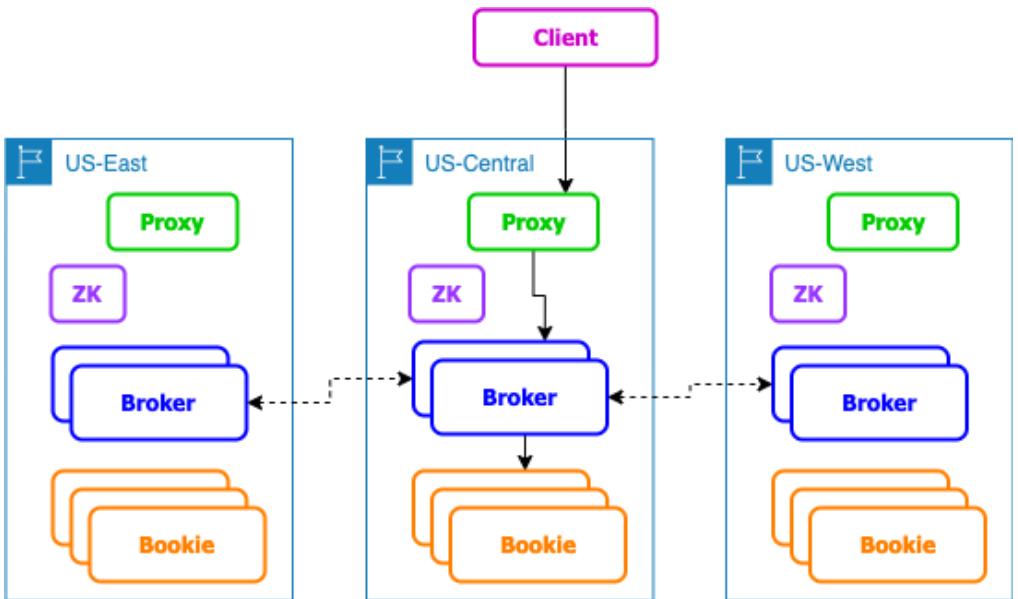


Figure B.2: Clients access an asynchronously geo-replicated cluster via the closest proxy. The Proxy routes the request to the Broker that owns the topic which then publishes the data to the Bookies in the same region. The Broker then replicates the incoming data to the brokers in the other regions.

While asynchronous geo-replication provides lower latency because the client doesn't have to wait for responses from the other data centers, it also provides weaker consistency guarantees due to asynchronous replication. Given that there is always a replication lag in asynchronous replication, there will always be some amount of data that hasn't been replicated from source to destination at any given point in time. Therefore, if you choose to implement this pattern your application must be able to tolerate some data loss in exchange for lower publish latency. Typically, the end-to-end replication latency is bounded by the *network round-trip time* (RTT) between the remote regions.

It is worth noting that asynchronous geo-replication is enabled on a per-tenant basis in Pulsar rather than a cluster-wide basis, allowing you to configure replication only for those topics for which it is needed. This allows each individual department or group to maintain control over their data replication policies. Asynchronous geo-replication is managed at the namespace level which provides more granular control over the datasets that get replicated. This is particularly useful in cases where you are not permitted to have our data leave a particular region due to regulatory and/or security reasons.

B.2.1 Configuring Asynchronous Geo-Replication

As you may recall from Chapter 2, a Pulsar Instance is comprised of one or more Pulsar clusters that act together as a single unit and can be administered from a single location, as shown in Figure B.3. In fact, one of the biggest reasons for using a Pulsar Instance is to enable geo-replication and only clusters within the same instance can be configured to replicate data amongst themselves. Therefore, enabling asynchronous geo-replication requires us to first create a Pulsar Instance.

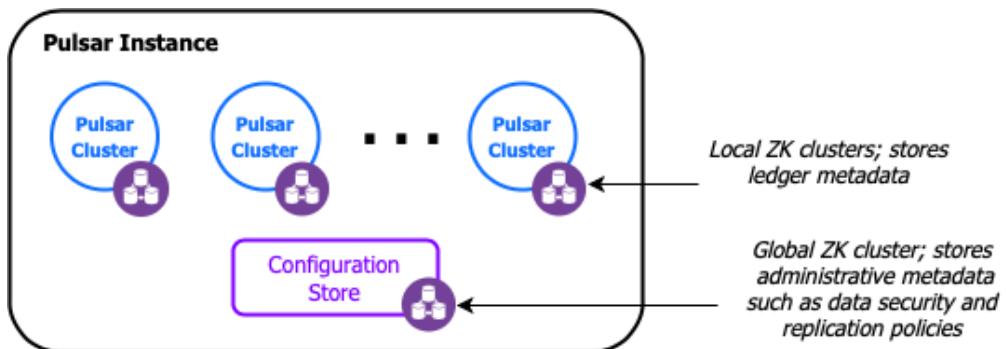


Figure B.3: A Pulsar Instance can consist of multiple, geographically dispersed clusters.

A Pulsar Instance employs an instance-wide Zookeeper cluster called the *configuration store* to retain information that pertains to multiple clusters, such as geo-replication, and tenant-level security policies. This allows you to define and manage these policies in a single location. While the complete documentation is available online, I wanted to highlight a few of these steps in the next section.

It is worth noting that the instance-wide Zookeeper instance should be deployed in such a manner as to make it completely independent from the individual Pulsar clusters so that in the event of a failure on the part of the instance-wide Zookeeper ensemble, the individual clusters will be able to continue to function without interruption.

DEPLOYING THE CONFIGURATION STORE

In addition to installing the individual clusters, creating a multi-cluster Pulsar instance involves deploying a separate ZooKeeper quorum to use as the configuration store. This configuration store should be implemented with its own dedicated Zookeeper quorum spread across at least 3 regions. Given the very low expected load on the configuration store servers, you can share the same hosts used for the local ZooKeeper quorum but will have to do so as either separate Zookeeper processes or K8s pods depending on your deployment environment. You will also have to use different TCP port in order to avoid port conflicts.

Listing B.4 Zookeeper Configuration for the Configuration Store Quorum

```
tickTime=2000
dataDir=/var/lib/zookeeper      #A
clientPort=2185      #B
initLimit=5
syncLimit=2
server.1=zk2.us-west.example.com:2185:2186  #C
server.2=zk2.us-central.example.com:2185:2186
server.3=zk2.us-east.example.com:2185:2186
```

#A Use a different location for storing the transaction log
#B Use a different port than the local ZK instance
#C Quorum consists of servers from across 3 regions listening on the same port.

Setting up a separate ZooKeeper quorum is fairly straightforward and well-documented on the Apache Zookeeper documentation page. Each Zookeeper server is contained in a single JAR file, so installation consists downloading the jar, unpacking it, and creating a configuration file. The default location for this configuration file is `conf/zoo.cfg`. All of servers in the new Zookeeper quorum should have the exact same configuration file as shown in Listing B.4

INITIALIZING CLUSTER METADATA

Now that the secondary Zookeeper quorum is up and running, the next step is to populate the configuration store with information about all the clusters that will be included in the Pulsar instance. This metadata can be initialized by using the `initialize-cluster-metadata` command of the pulsar CLI tool as shown in Listing B.5

Listing B.5 Initializing the Cluster Metadata

```
$ /pulsar/bin/pulsar initialize-cluster-metadata \
  --cluster us-west \    #A
  --zookeeper zk1.us-west.example.com:2181 \    #B
  --configuration-store zk1.us-west.example.com:2184 \    #C
  --web-service-url http://pulsar.us-west.example.com:8080/ \    #D
  --web-service-url-tls https://pulsar.us-west.example.com:8443/ \    #E
  --broker-service-url pulsar://pulsar.us-west.example.com:6650/ \    #F
  --broker-service-url-tls pulsar+ssl://pulsar.us-west.example.com:6651/
```

#A The name of the cluster that will be used when setting up replication
#B The local ZK connection string
#C The connection string for the configuration store

The command associates all the various connection URLs to a given cluster name, and stores that information inside the configuration store. This information is used when replication is enabled to connect the brokers that need to exchange data between them, e.g., replication data from us-west to us-east. You will need to run this command for every Pulsar cluster you are adding to the instance.

CONFIGURE THE SERVICES TO USE THE CONFIGURATION STORE

After you have populated the configuration store with all the metadata associated with the Pulsar clusters in your instance, you will need to modify a couple of configuration files on

EVERY cluster to enable geo-replication. Since geo-replication is accomplished via broker-to-broker communication, the most important one is the `conf/broker.conf` configuration file as shown in Listing B.6.

Listing B.6 Updated broker.conf for Asynchronous Geo-Replication

```
# Local ZooKeeper servers
zookeeperServers=zk1.us-west.example.com:2181,zk2.us-west.example.com:2181,zk3.us-
west.example.com:2181      #A

# Configuration store quorum connection string.
configurationStoreServers=zk2.us-west.example.com:2185,zk2.us-
central.example.com:2185,zk2.us-east.example.com:2185      #B

clusterName=us-west      #C
```

#A Use the local ZK quorum as before
#B Use the second ZK quorum for the configuration store
#C Specify the name of the cluster that the broker belongs to.

Make sure that you set the `zookeeperServers` parameter to reflect the local quorum and the `configurationStoreServers` parameter to reflect the configuration store quorum. You also need to specify the name of the cluster to which the broker belongs using the `clusterName` parameter, taking care to use the value you specified in the `initialize-cluster-metadata` command. Finally, make sure that the broker and web service ports match the values you provided in the `initialize-cluster-metadata` command as well. Otherwise, the replication process will fail because it the source Broker will be attempting to communicate over the wrong port.

If you are using the service discovery mechanism that is included with Pulsar, then you need to change a few parameters in the `conf/discovery.conf` configuration file. Specifically, you must set the `zookeeperServers` parameter to the ZooKeeper quorum connection string of the cluster and the `configurationStoreServers` setting to the configuration store quorum connection string using the same values that you did in the broker configuration file.

Once you have finished updating all of the necessary configuration files, all of these services will need to be restarted after these changes are made in order for the new properties to take effect.

B.3 Asynchronous Geo-Replication Patterns

With asynchronous replication, Pulsar provides tenants a great degree of flexibility for customizing their replication strategy. That means that an application is able to set up active-active and full-mesh replication, active-standby replication, and aggregation replication across multiple data centers. Let's take a quick look at how to implement each of these patterns inside of Pulsar.

B.3.1 Multi-Active Geo-Replication

Asynchronous geo-replication is controlled on a per-tenant basis in Pulsar. This means geo-replication can only be enabled between clusters when a tenant has been created that allows access to all of the clusters involved. In order to configure *Multi-Active Geo-Replication*, you need to specify which clusters a tenant has access to via the pulsar-admin CLI as shown in Listing B.7 which shows the command to create a new tenant and grant it permission to access the us-east and us-west clusters only.

Listing B.7 Granting Tenant Access to Clusters

```
$ ./pulsar/bin/pulsar-admin tenants create customers \      #A
  --allowed-clusters us-west,us-east \    #B
  --admin-roles test-admin-role
```

#A Create a new tenant named customers

#B Grant the tenant permission to access these 2 clusters only.

Now that the tenant has been created, we need to configure the geo-replication at the namespace level. Therefore, we will first need to create the namespace using the pulsar-admin CLI tool and then assign the namespace to a cluster—or multiple clusters—using the set-clusters command as shown in Listing B.8.

Listing B.8 Assigning a Namespace to a Cluster

```
$ ./pulsar/bin/pulsar-admin namespaces create customers/orders
$ ./pulsar/bin/pulsar-admin namespaces set-clusters customers/orders \
  --clusters us-west,us-east,us-central
```

By default, once replication is configured between two or more clusters as shown in Listing B.8 all of the messages published to topics inside the namespace in one cluster are asynchronously replicated to all the other clusters in the list. Therefore, the default behavior is effectively full-mesh replication of all the topics in the namespace with messages getting published in multiple directions as shown in Figure B.4. When you only have two clusters, then the default behavior can be thought of as an active-active cluster configuration where the data is available on both clusters to serve clients and in the event of a single cluster failure, all of the clients can be re-directed to the remaining active cluster with interruption.

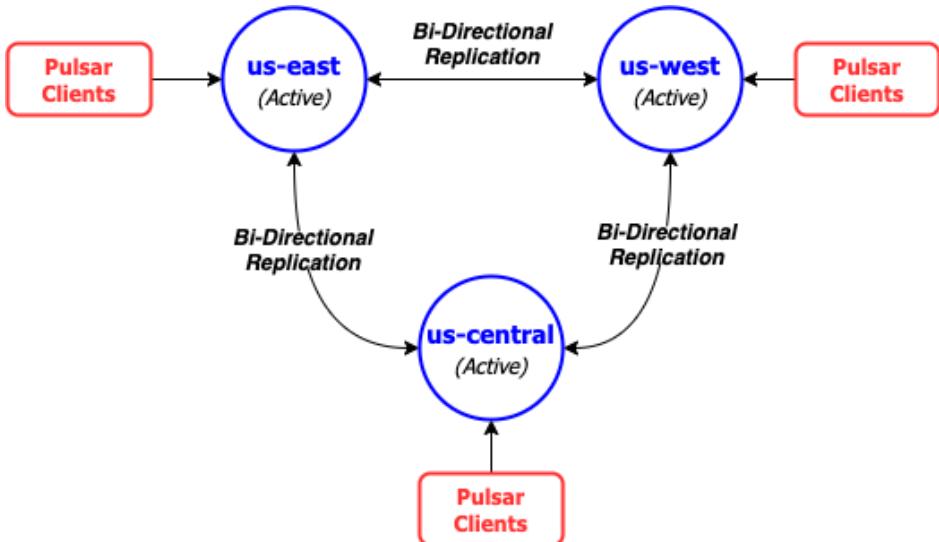


Figure B.4: The default behavior is full-mesh geo-replication between all clusters. Messages published to a topic in the namespace to the us-east cluster will be forwarded to both the us-west and us-central clusters.

Besides full-mesh (active-active) geo-replication, there are a few other replication patterns you can use. Another common one for disaster recovery is the *active-standby replication pattern*.

B.3.2 Active-Standby Geo-Replication

In this situation you are looking to keep an up-to-date copy of the cluster at a different geographical location so that you can resume operations in the event of a failure with a minimal amount of data loss or recovery time.

Since Pulsar doesn't provide a means for specifying one-way replication of namespaces, the only way to accomplish this configuration is by restricting the clients to a single cluster known as the "active" cluster and having them all failover to the standby cluster only in the event of a failure. Typically, this can be accomplished via a load balancer or other network-level mechanism that makes the transition transparent to the clients as shown in Figure B.5. Pulsar clients publish messages to the active cluster which are then replicated to the standby cluster for backup.

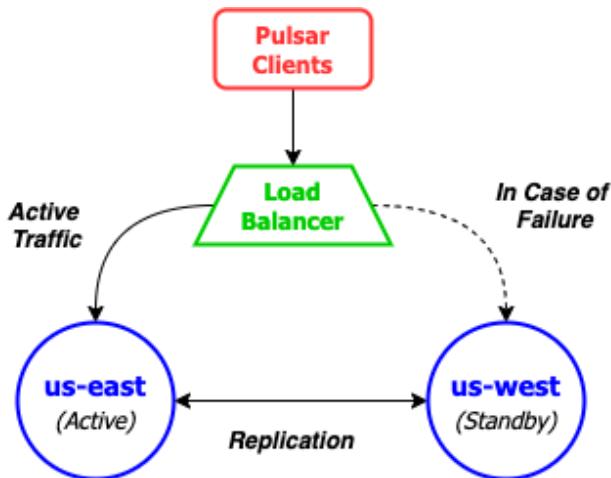


Figure B.5: You can use asynchronous geo-replication to implement an active-standby scenario where all of the data within a given namespace is forwarded to a cluster that will be used only in the event of a failure.

As you may have noticed, the replication of the pulsar data will still be done bi-directionally, which means that the us-west cluster will attempt to send the data it receives during the outage to the us-east cluster. This might be problematic if the failure is related to one or more components within the Pulsar cluster, or the network for the us-east cluster is unreachable, etc. Therefore, you should consider adding selective replication code inside your Pulsar producers to prevent the us-west cluster from attempting to replicate messages to the us-east cluster which is most likely dead.

You can restrict replication selectively, by directly specifying a replication list for a message at the application level. The code in Listing B.9 shows an example of producing a message that will only be replicated to the us-west cluster, which is the behavior you want in this active-standby scenario.

Listing B.9 Selective Replication Per Message

```

List<String> restrictDatacenters = Lists.newArrayList("us-west");

Message message = MessageBuilder.create()
    ...
    .setReplicationClusters(restrictDatacenters)
    .build();

producer.send(message);

```

Sometimes you want to funnel messages from multiple clusters into a single location for aggregation purposes. One such example would be gathering all the payment data collected from across all the geographical regions for processing and collection, etc.

B.3.3 Aggregation Geo-Replication

Assume we have three clusters all actively serving the GottaEat customers in their respective regions, and a fourth Pulsar cluster named “internal” that is completely isolated from the web and only accessible by internal employees that is used to aggregate the data from all of the customer-serving Pulsar clusters as shown in Figure B.6. In order to implement Aggregation Geo-Replication across these 4 clusters, you will need to use the commands shown in Listing B.10 which first creates the “E-payments” tenant and grants access to all the clusters.

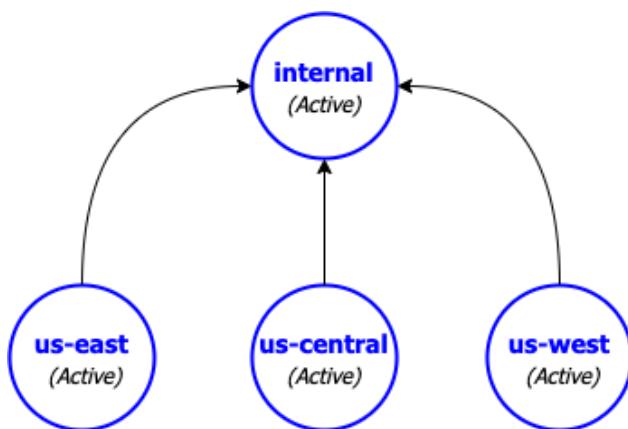


Figure B.6: An aggregation geo-replication configuration to funnel messages from 3 customer-facing Pulsar clusters to an internal Pulsar cluster for aggregation and analysis.

Next you will need to create a namespace for each of the customer services clusters, e.g., E-payments/us-east-payments. You CANNOT use one such as E-payments/payments because that would lead to full mesh replication if you attempted to use it instead since every cluster would have that namespace. Thus, a per-cluster namespace is required in order for this to work.

Listing B.10 Aggregator Geo-Replication

```
/pulsar/bin/pulsar-admin tenants create E-payments \ #A
--allowed-clusters us-west,us-east,us-central,internal

/pulsar/bin/pulsar-admin namespaces create E-payments/us-east-payments #B
/pulsar/bin/pulsar-admin namespaces create E-payments/us-west-payments
/pulsar/bin/pulsar-admin namespaces create E-payments/us-central-payments

/pulsar/bin/pulsar-admin namespaces set-clusters \ #C
E-payments/us-east-payments --clusters us-east,internal

/pulsar/bin/pulsar-admin namespaces set-clusters \ #D
E-payments/us-west-payments --clusters us-west,internal

/pulsar/bin/pulsar-admin namespaces set-clusters \ #E
```

```
E-payments/us-central-payments --clusters us-central,internal
```

#A Create the global tenant for Payments
#B Create the cluster-specific namespaces
#C Configure use-east to internal replication
#D Configure use-west to internal replication
#E Configure use-central to internal replication

If you decide to implement this pattern and you intend to run identical copies of an application across all the customer servicing cluster, then be sure to make the topic name configurable so that the application running on us-east knows to publish messages to topics inside the us-east-payments namespace, etc. Otherwise, the replication will not work.