redhat.

RED HAT®

AMQ Streams

1.0

Stelios Kousouris
Senior Applications Development Architect

Introduction to Apache Kafka

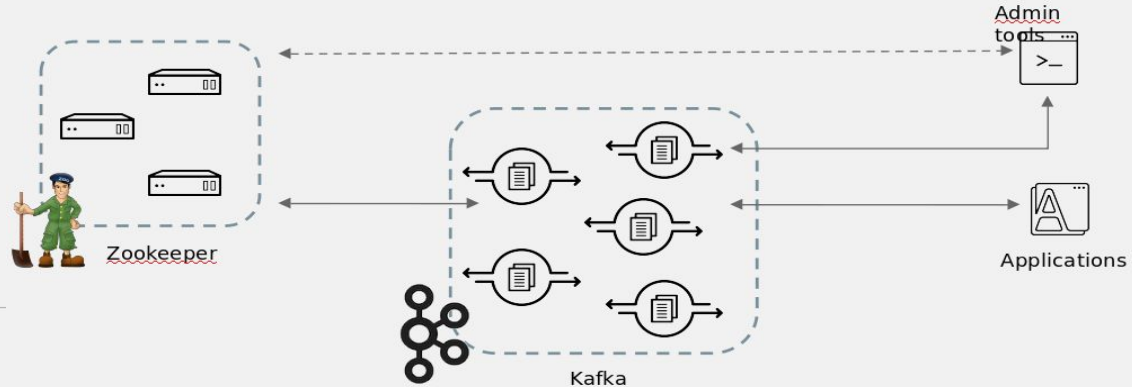
Apache Kafka ecosystem

- Apache Kafka has an ecosystem consisting of many components / tools
  - Kafka Core
    - Broker
    - Clients library (Producer, Consumer, Admin)
    - Management tools
  - Kafka Connect
  - Kafka Streams
  - Mirror Maker

Apache Kafka ecosystem

Apache Kafka components

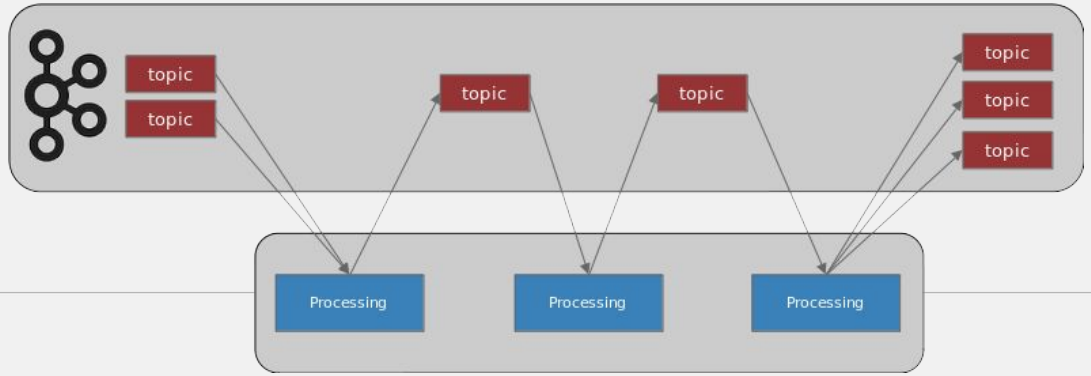
- **Kafka Broker**
  - Central component responsible for hosting topics and delivering messages
  - One or more brokers run in a cluster alongside with a Zookeeper ensemble
- **Kafka Producers and Consumers**
  - Java-based clients for sending and receiving messages
- **Kafka Admin tools**
  - Java- and Scala- based tools for managing Kafka brokers
  - Managing topics, ACLs, monitoring etc.



Apache Kafka ecosystem

Apache Kafka components

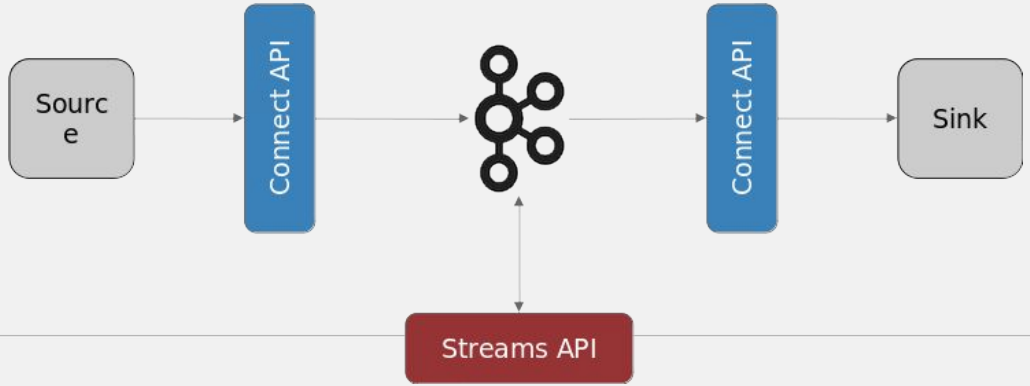
- **Kafka Streams**
  - Stream processing framework
  - Streams are Kafka topics (as input and output)
  - Scaling the stream application horizontally
  - Creates a topology of processing nodes (filter, map, join etc) acting on a stream
  - Low level processor API
  - High level DSL
  - Using “internal” topics (when re-partitioning is needed or for “stateful” transformations)
  - Scala wrapper (New in Kafka 2.0)



Apache Kafka ecosystem

Apache Kafka components

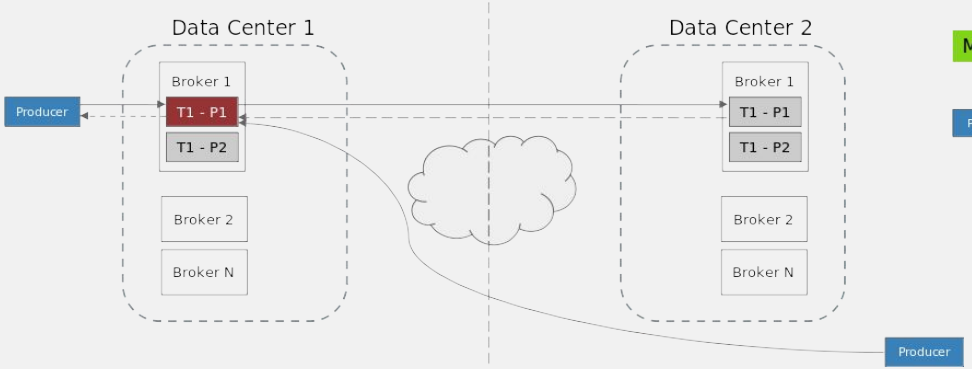
- **Kafka Connect**
  - Framework for transferring data between Kafka and other data systems
  - Facilitate data conversion, scaling, load balancing, fault tolerance, …
  - Connector plugins are deployed into Kafka connect cluster
  - Well defined API for creating new connectors (with Sink/Source)
  - Apache Kafka itself includes only *FileSink* and *FileSource* plugins (reading records from file and posting them as Kafka messages / writing Kafka messages to files)
  - Many additional plugins are available outside of Apache Kafka

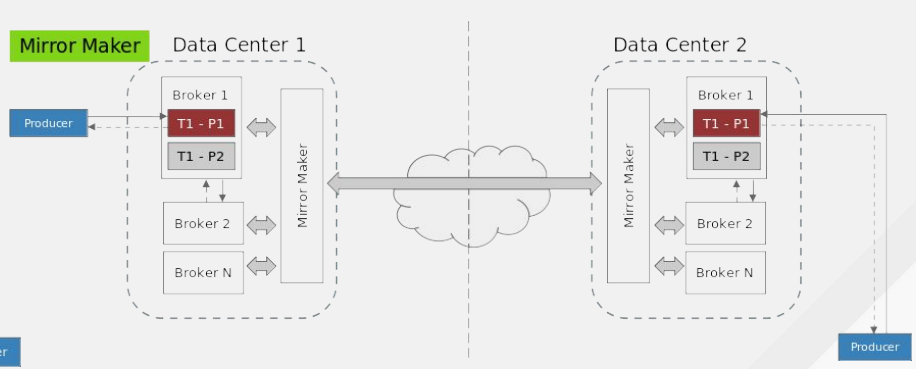


Apache Kafka ecosystem

Apache Kafka components

- **Mirror Maker**
  - Kafka clusters do not work well when split across multiple datacenters
  - Low bandwidth, High latency
  - For use within multiple datacenters it is recommended to setup independent cluster in each data center and mirror the data
  - Tool for replication of topics between different clusters



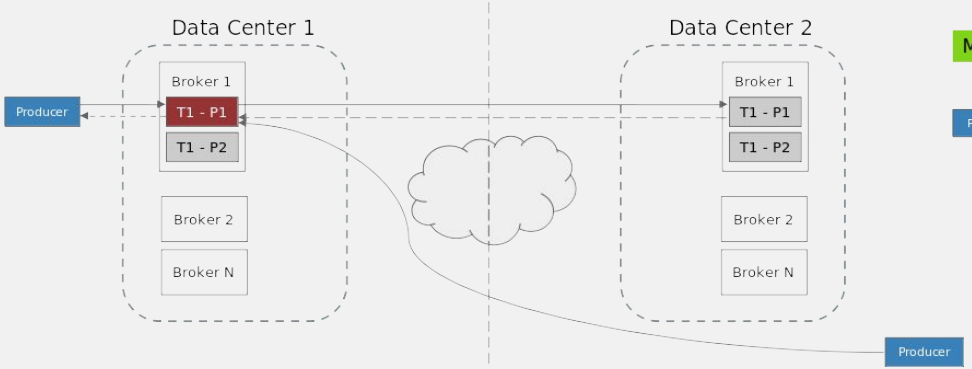


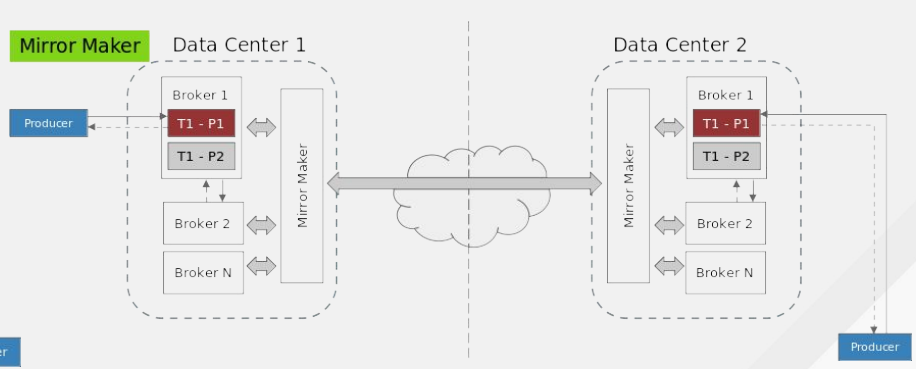
redhat.

Apache Kafka ecosystem

Apache Kafka components

- **Mirror Maker**
  - Kafka clusters do not work well when split across multiple datacenters
  - Low bandwidth, High latency
  - For use within multiple datacenters it is recommended to setup independent cluster in each data center and mirror the data
  - Tool for replication of topics between different clusters





redhat.

Zookeeper

Apache Kafka internals

- Electing controller
  - Zookeeper is used to elect a controller, make sure there is only one and elect a new one it if it crashes
- Cluster membership
  - Which brokers are alive and part of the cluster? This is also managed through ZooKeeper
- Topic configuration
  - Which topics exist, how many partitions each has, where are the replicas, who is the preferred leader, what configuration overrides are set for each topic
- Quotas and ACLs
  - How much data is each client allowed to read and write and who is allowed to do that to which topic

Security

Apache Kafka internals

- Encryption between clients and brokers and between brokers
  - Using SSL
- Authentication of clients (and brokers) connecting to brokers
  - Using SSL (mutual authentication)
  - Using SASL (with PLAIN, Kerberos or SCRAM-SHA as mechanisms)
- Authorization of read/writes operation by clients
  - ACLs on resources such as topics
  - Authenticated “principal” for configuring ACLs
  - Pluggable
- It’s possible to mix encryption/no-encryption and authentication/no-authentication

Security

Apache Kafka internals

- Zookeeper
  - No encryption support in latest stable version
  - Support for SASL authentication
    - Using Kerberos or locally stored credentials
- Kafka treats information in public
  - Every user can see it
  - ACL protection can be enabled for changing the data

Supported in 1.0

Supported

Platforms

- AMQ Streams 1.0 server components
  - Broker, Zookeeper, Kafka Connect, MirrorMaker
  - RHEL7 / x86_64 (OpenJDK 8, OracleJDK 8, IBM JDK 8)
- AMQ Streams 1.0 client components
  - Kafka Consumer, Producer and Admin clients
  - Kafka Streams
  - Supported on RHEL7 / x86_64 (OpenJDK 8, OracleJDK 8, IBM JDK 8)

Supported

Protocol & API Versions

- Apache Kafka is backwards compatible with older clients / brokers
  - The protocol / API is versioned according to Kafka release (e.g. 1.1, 1.0, 0.10.1)
  - Latest Kafka still supports many old API versions (from 0.8.0 up to 1.1)
- AMQ Streams 1.0 will support only the latest version
  - The older versions will still work with AMQ Streams 1.0
  - But Red Hat support will be offered only for the latest version
  - In next releases, support for newer versions will be added as they are implemented in Kafka

Supported Components

Broker, MirrorMaker

- Kafka Broker
  - Fully supported on RHEL 7
- MirrorMaker
  - Fully supported

Supported Components

Clients

- Apache Kafka project contains only Java client libraries
  - Consumer, Producer and Admin APIs
  - Streams API
  - Will be supported by AMQ Streams
- Clients for other languages are usually open-source, but are not part of Apache Kafka
  - Many clients are bindings against librdkafka library (C language)
  - These clients will not be supported by AMQ Streams 1.0
    - Users are free to use them on their own

Supported Components

Kafka Connect

- Apache Kafka Connect will be supported in both distributed and standalone modes
- Only the FileSink and FileSoruce plugins will be supported

Supported Components

Apache Zookeeper

- Apache Zookeeper is a dependency of Kafka
- Just an implementation detail for AMQ Streams
- Zookeeper will be supported only for use by Kafka
  - No support will be provided for using Zookeeper with other applications
  - Plans for removing Zookeeper dependency are being discussed upstream
    - Once Zookeeper is not needed by Apache Kafka, we will drop it as well

UnSupported

- Several notable parts of wider Kafka ecosystem which will not be supported in 1.0
  - Non-Java clients
  - Kafka Connect plugins
  - Confluent REST proxy (https://github.com/confluentinc/kafka-rest)
  - Confluent Schema Registry (https://github.com/confluentinc/schema-registry)
  - KSQL (https://github.com/confluentinc/ksql)

After 1.0 - Looking to support

- Security
  - Improve authentication and authorization options
  - Support LDAP for authentication
  - Support Red Hat SSO for authentication and authorization
  - Support authentication and authorization using tokens
- AMQP Bridge
  - Based on https://github.com/strimzi/amqp-kafka-bridge
  - Advantages
    - Integrate with other AMQ products using AMQP
    - Use our supported AMQP clients (as an alternative for non-Java Kafka clients which we don’t support)
    - No need to expose the whole Kafka cluster
  - Further plans to extend the Bridge with HTTP and MQTT support

After 1.0 - Looking to support

- Schema Registry
  - Possibly based on https://github.com/jhalliday/perspicuus
  - Advantages
  - Support for different schemas
  - JSON, AVRO, Protocol Buffers
- Cluster balancing
  - Support automated cluster balancer
    - Automatically balance the cluster
      - Move topics between nodes to achieve best performance
      - Based on Memory, CPU, Disk or Network IO
    - Reusing one of existing projects
    - Build from scratch using OptaPlanner for optimization

Why & Where should you use AMQ Streams

Why use AMQ Streams

- Scalability and Performance
  - Designed for horizontal scalability
  - Cluster sizes from few brokers up to 1000s of brokers
    - 3 nodes usually seen as minimum for production (HA, message durability)
    - Most clusters are under 50 nodes
    - Different approaches: One big cluster vs. several small clusters
  - Scaling has minimal impact on throughput and latency
  - Adding nodes to running cluster is easy
- Message ordering guarantee
  - Messages are written to disk in the same order as received by the broker
  - Messages are read from disk from the requested offset
  - Kafka protocol makes sure that one consumer can read only one partition
  - Note: Order is guaranteed only within a single partition!
  - Note: No synchronization between producers!

Why use AMQ Streams

- Message rewind / replay
  - Limited only by available disk space
    - Amount of stored messages has no impact on performance
  - Topic / Partition size has no direct impact on performance
  - Allows to reconstruct application state by replaying the messages
  - Combined with compacted topics allows to use Kafka as key-value store
  - Event sourcing (https://martinfowler.com/eaaDev/EventSourcing.html)
  - Parallel running (https://en.wikipedia.org/wiki/Parallel_running)

AMQ Streams

vs AMQ 7 Broker

- Use AMQ Streams when
  - You need scalability to achieve highest possible throughput
  - You need large number of parallel consumers
  - Your messages are cattle and not pets
  - You need message ordering or replay features
  - You can reuse some of the components available in Kafka ecosystem instead (e.g. Kafka Connect plugins)

AMQ Streams

vs AMQ 7 Broker

- Use AMQ Broker when
  - You care about individual messages
  - You use request/response patterns
  - You want to use standard clients and protocols
  - You need TTL or DLQ semantics
  - You need more sophisticated routing patterns

AMQ Streams on Openshift

AMQ Streams on OpenShift

What is it?

- Based on OSS project called Strimzi
- Provides:
  - Docker images for running Apache Kafka and Zookeeper
  - Tooling for managing and configuring Apache Kafka clusters and topics
- Follows the Kubernetes operator model
- OpenShift 3.9 and higher

AMQ Streams on OpenShift

What is Strimzi?

- Open source project focused on running Apache Kafka on Kubernetes and OpenShift
  - Web site: http://strimzi.io/

AMQ Streams on OpenShift

Why is Strimzi?

- Kafka brokers inherently stateful.
- Updating and scaling a Kafka cluster requires careful orchestration to ensure that messaging clients are unaffected and no records are lost.
- By design, Kafka clients connect to all the brokers in the cluster.
- This is part of what gives Kafka its horizontal scaling and high availability, but when running on OpenShift, this means the Kafka cluster cannot simply be put behind a load-balanced service like other services. Instead services have to be orchestrated in parallel with cluster scaling.
- Running Kafka also requires running a Zookeeper cluster, which has many of the same challenges as running the Kafka cluster.

AMQ Streams on OpenShift - How?

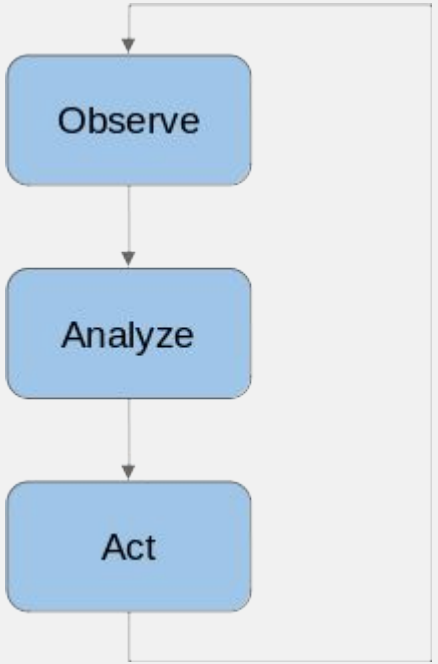
Operator

- Application-specific controller is used to create, configure and manage other complex application
  - The controller contains specific domain / application knowledge
  - Usually used for stateful applications (databases, ...) which are non-trivial to operate on Kubernetes / OpenShift
- Controller operates based on input from ~~Config Maps~~ or **Custom Resource Definitions**
  - User describes the desired state
  - Controller applies this state to the application

AMQ Streams on OpenShift - How?

Operator

- Observe
  - Monitor the current state of the application
- Analyze
  - Compare the actual state to the desired state
- Act
  - Resolve any differences between actual and desired state



AMQ Streams on OpenShift - How?

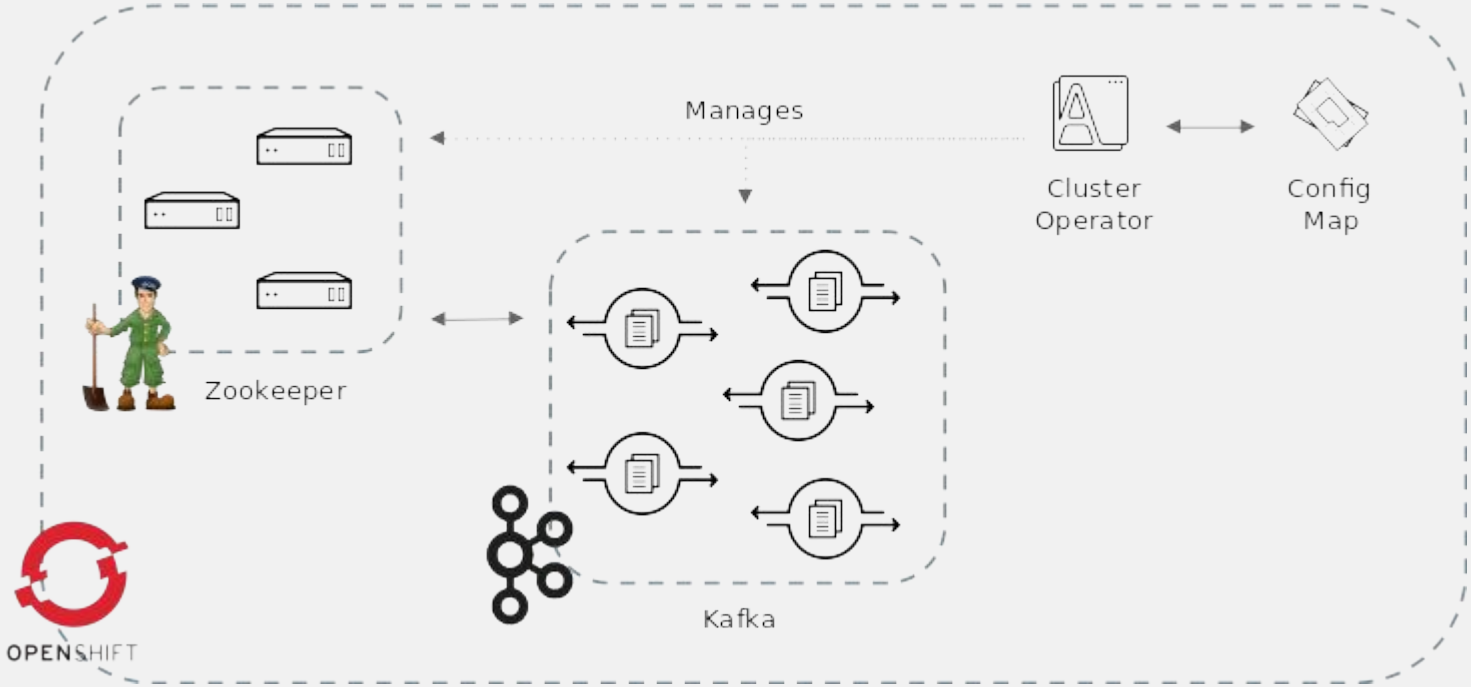
Custom Resource Definitions

- Flexible data structure
- Possibility to set permissions for the CRD resources

Cluster Operator

Cluster Operator

Creating and managing Apache Kafka clusters



Cluster Operator

Supported clusters

- Cluster Operator can deploy and manage two kinds of cluster
  - Kafka cluster
  - Cluster of Kafka brokers
  - Includes Zookeeper deployment
  - Using Stateful Sets for managing Kafka and Zookeeper
- Kafka Connect cluster
  - Distributed Kafka Connect cluster
  - Using Deployment
  - S2I support for adding additional plugins

Cluster Operator

Supported features

- Cluster Operator currently allows to configure
  - Number of Zookeeper, Kafka and Kafka Connect nodes
  - Configuration of Kafka and Kafka Connect
  - Storage
    - Persistent versus Ephemeral
    - Storage size
  - Metrics exports for Prometheus
  - Healthchecks

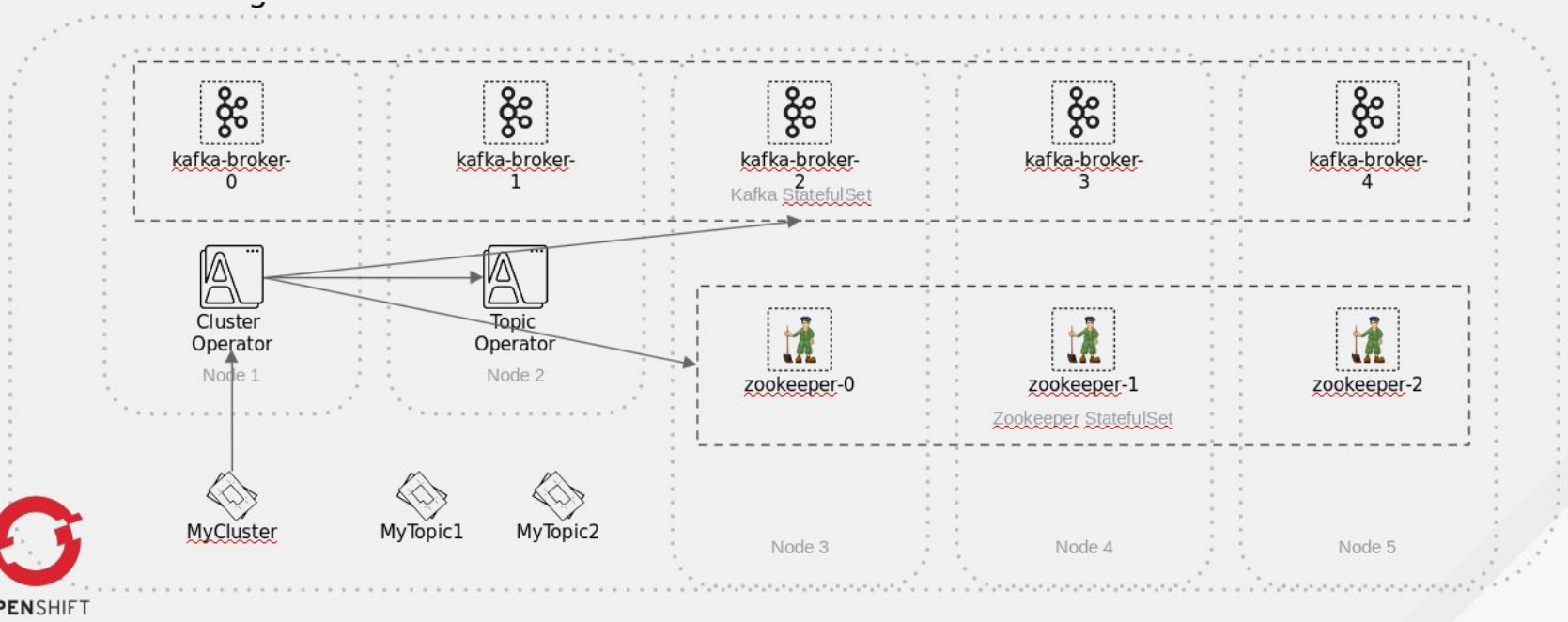
Cluster Operator

Creating a cluster

- One Cluster Operator can manage several clusters in parallel
  - Can cover one or more projects / namespaces
- To deploy new cluster
  - (Deploy the cluster controller)
  - Create a Config Map describing the cluster
  - The Controller will see the Config Map and start deploying the cluster
  - The cluster will be deployed and ready to use

Cluster Operator

Creating a cluster



OPENSHIFT

redhat.

Cluster Operator

Managing cluster

- Clusters can be modified by modifying the Config Map
  - Scale-up / scale-down
  - Kafka configuration
- Custer Operator will update the cluster to match the desired state described in Config Map
- Update does not allow to change storage configuration
  - Such operation cannot be done without losing data

Cluster Operator

Deleting cluster

- Clusters can be deleted by deleting the CRDs
  - All cluster resources will be removed
  - Persistent Volume Claims will be deleted according to user configuration
- Deleting the CRDs is irreversible

Cluster Operator

Kafka Connect and Source 2 Image

- Kafka Connect can be deployed with Source 2 Image support (S2I)
  - By default Kafka Connect contains only FileSync and FileSource plugins which are not much useful in distributed mode
  - To make it more useful, additional plugins need to be added
  - New Docker image with additional plugins can be build using S2I
    - User prepares a binary of the plugin and triggers new build
    - The build adds the plugin to the original Kafka Connect image
    - Rolling update is triggered to use the new image

Cluster Operator

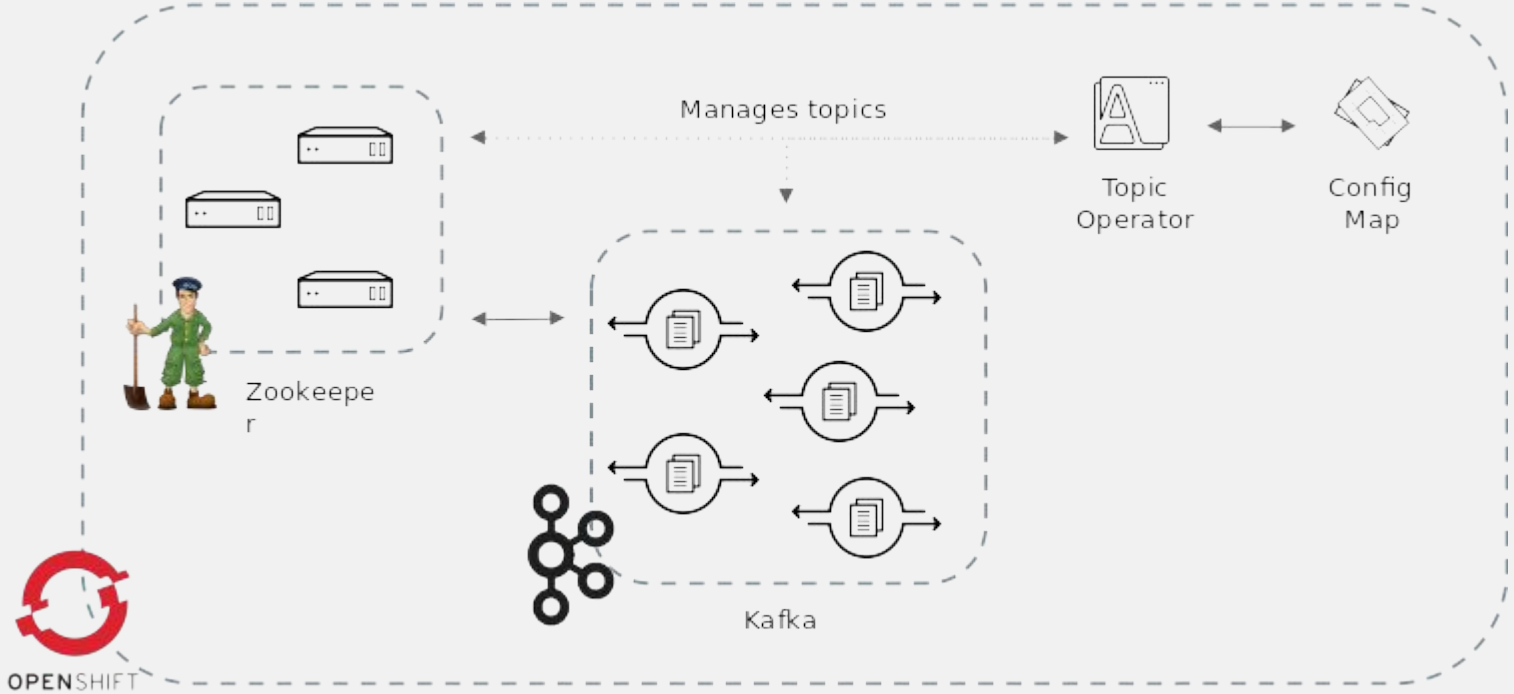
Kafka Connect and Source 2 Image

- Kafka Connect can be deployed with Source 2 Image support (S2I)
  - By default Kafka Connect contains only FileSync and FileSource plugins which are not much useful in distributed mode
  - To make it more useful, additional plugins need to be added
  - New Docker image with additional plugins can be build using S2I
    - User prepares a binary of the plugin and triggers new build
    - The build adds the plugin to the original Kafka Connect image
    - Rolling update is triggered to use the new image

Topic Operator

Topic Operator

Creating and managing Kafka topics



Topic Operator

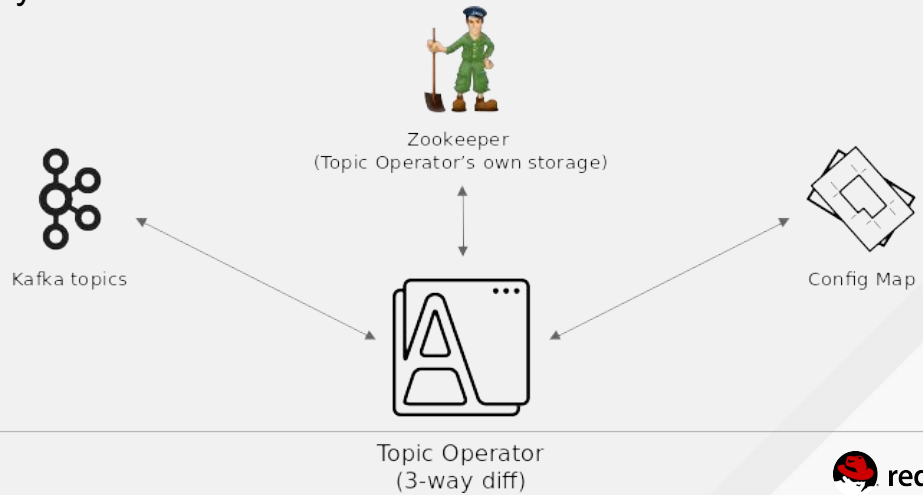
Creating and managing Kafka topics

- Topic Operator is managing Kafka topics
- Some Kafka components (Streams, Connect) often create their own topics
  - Bi-directional synchronization
  - Changes done directly in Kafka / Zookeeper are applied to CRDs
  - Changes done in Config Maps are applied to Kafka topics
- Topic Operator solves this by using 3-way diff
  - Our own Zookeeper-based store
  - Apache Kafka / Zookeeper
  - CRDs

Topic Operator

Creating and managing Kafka topics

- Topic Operator is managing Kafka topics
- Some Kafka components (Streams, Connect) often create their own topics
  - Bi-directional synchronization
  - Changes done directly in Kafka / Zookeeper are applied to CRDs
  - Changes done in Config Maps are applied to Kafka topics
- Topic Operator solves this by using 3-way diff
  - Our own Zookeeper-based store
  - Apache Kafka / Zookeeper
  - CRDs



Topic Operator

Creating and managing Kafka topics

- Kafka gives user more freedom for naming than OpenShift
  - CRDs configuring topics contain “name” field to define the name which is not allowed for OpenShift resource
  - If such topic is created by Kafka, it will be mapped to a Config Map named after specially encoded name
- We recommend to use only topic names which are allowed as OpenShift resource names

User Operator

User Operator

Creating and managing Kafka users/acls

- The User Operator provides synchronization between a *KafkaUser* custom resource in OpenShift and Kafka's own user and Access Control List data structures.
- Allows user to provision the user accounts their application needs at the same time, and in the same way as the application itself: As OpenShift CRD resources.
- User Operator watches for *KafkaUser* custom resource creation and then ensures they are configured in the KAFKA cluster.
- It does not sync changes from KAFKA cluster to Openshift CRDs (unlike Topic Operator)
- In addition manages authorization rule by including description of the user’s rights on specific topics

User Operator

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: KafkaUser
metadata:
  name: my-user
  labels:
        strimzi.io/cluster: my-cluster
spec:
  authentication:
        type: tls
  authorization:
        type: simple
        acls:
        - resource:
        type: topic
        name: my-topic
        operation: Read
        - resource:
        type: topic
        name: my-topic
        operation: Describe
        - resource:
        type: group
        name: my-hello-world-consumer
        operation: Read
        - resource:
        type: topic
        name: my-topic
        operation: Write
        - resource:
        type: topic
```

redhat.

THANK YOU

plus.google.com/+RedHat

facebook.com/redhatinc

linkedin.com/company/red-hat

twitter.com/RedHatNews

youtube.com/user/RedHatVideos