KIP-877: Mechanism for plugins and connectors to register metrics

Status

Current state: Under Discussion

Discussion thread: here

JIRA: Unable to render Jira issues macro, execution error.

Please keep the discussion on the mailing list rather than commenting on the wiki (wiki discussions get unwieldy fast).

Motivation

Kafka exposes many pluggable API for users to bring their custom plugins. For complex and critical plugins it's important to have metrics to monitor their behavior. Plugins wanting to emit metrics can use the Metrics class from the Kafka API but when creating a new Metrics instance it does not inherits the tags from the component it depends on (for example from a producer for a custom partitioner), or the registered metrics reporters. As most plugins are configurable, a workaround is to reimplement the metric reporters logic and in some case for tags too but that is cumbersome. Also by creating a separate Metrics instance, these metrics are separate from the client's and in case multiple clients are running in the same JVM, for example multiple producers, it can be hard to identify the specific client that is associated with some plugin metrics.

This issue also applies to connectors and tasks in Kafka Connect. For example MirrorMaker2 creates its own Metrics object and has logic to add the metric reporters from the configuration.

In this proposal, a "plugin" is an interface users can implement and that is instantiated by Kafka. For example, a class implementing org.apache.kafka.server.policy.CreateTopicPolicy is considered a plugin as it's instantiated by brokers. On the other hand a class implementing org.apache.kafka.clients.producer.Callback is not considered a plugin as it's instantiated in user logic.

Public Interfaces

I propose introducing a new interface: `Monitorable`. If a plugin implements this interface, the `withPluginMetrics()` method will be called when the plugin is instantiated (after `configure()` if the plugin also implements `Configurable`). This will allow plugins to add their own metrics to the component (broker, producer, consumer, etc) that instantiated them.

Monitorable.java

```
package org.apache.kafka.common.metrics;

public interface Monitorable {

    /**
     * Set the PluginMetrics instance from the component that instantiates the plugin.
     * Plugins can register and unregister metrics using the given PluginMetrics
     * at any point in their lifecycle prior to their close method being called.
     *
     * Plugin must call the close() method on this PluginMetrics instance in their close()
     * method to clear all metrics they registered.
     */
    void withPluginMetrics(PluginMetrics metrics);

}
```

The PluginMetrics interface has methods to add and remove metrics and sensors. Plugins will only be able to remove metrics and sensors they created. Metrics created via this class will have their group set to "plugins" and include tags that uniquely identify the plugin.

```
/**
 * Implementations are thread safe so plugins may use the PluginMetrics instance from multiple threads
 */
public interface PluginMetrics extends Closeable {

    /**
     * Create a {@link MetricName} with the given name, description and tags. The group will be set to "plugins"
     * Tags to uniquely identify the plugins are automatically added to the provided tags
     *
     * @param name        The name of the metric
     * @param description A human-readable description to include in the metric
     * @param tags        additional key/value attributes of the metric
     */
    MetricName metricName(String name, String description, Map<String, String> tags);

    /**
     * Add a metric to monitor an object that implements {@link MetricValueProvider}. This metric won't be
associated with any
     * sensor. This is a way to expose existing values as metrics.
     *
     * @param metricName The name of the metric
     * @param metricValueProvider The metric value provider associated with this metric
     * @throws IllegalArgumentException if a metric with same name already exists.
     */
    void addMetric(MetricName metricName, MetricValueProvider<?> metricValueProvider);

    /**
     * Remove a metric if it exists.
     *
     * @param metricName The name of the metric
     * @throws IllegalArgumentException if the plugin has not already created a metric with this name
     */
    void removeMetric(MetricName metricName);

    /**
     * Create a sensor with the given unique name. The name must only be unique for the plugin, so different
plugins can use the same names.
     *
     * @param name The sensor name
     * @return The sensor
     * @throws IllegalArgumentException if a sensor with same name already exists for this plugin
     */
    Sensor sensor(String name);

    /**
     * Remove a sensor (if it exists) and its associated metrics.
     *
     * @param name The name of the sensor to be removed
     * @throws IllegalArgumentException if the plugin has not already created a sensor with this name
     */
    void removeSensor(String name);
}
```

The PluginMetrics interface implements Closeable. Calling the close() method removed all metrics and sensors created by this plugin. It will be the responsibility of the plugin that creates metrics to call close() of the PluginMetrics instance they were given to remove their metrics.

New methods will be added to AbstractConfig so new plugin instances implementing Monitorable can get a PluginMetrics instance.

AbstractConfig.java

```
/**
 * Get a configured instance of the give class specified by the given configuration key. If the object
implements
 * Configurable configure it using the configuration. If the object implements Monitorable, set the appropriate
PluginMetrics for that instance.
 *
 * @param key The configuration key for the class.
 * @param t The interface the class should implement.
 * @param metrics The metrics registry to use to build the PluginMetrics instance if the class implements
Monitorable.
 * @return A configured instance of the class.
 */
public <T> T getConfiguredInstance(String key, Class<T> t, Metrics metrics);

/**
 * Get a configured instance of the give class specified by the given configuration key. If the object
implements
 * Configurable configure it using the configuration. If the object implements Monitorable, set the appropriate
PluginMetrics for that instance.
 *
 * @param key The configuration key for the class.
 * @param t The interface the class should implement.
 * @param configOverrides override origin configs.
 * @param metrics The metrics registry to use to build the PluginMetrics instance if the class implements
Monitorable.
 * @return A configured instance of the class.
 */
public <T> T getConfiguredInstance(String key, Class<T> t, Map<String, Object> configOverrides, Metrics
metrics);

/**
 * Get a list of configured instances of the given class specified by the given configuration key. The
configuration
 * may specify either null or an empty string to indicate no configured instances. If an instance implements
Monitorable,
 * set the appropriate PluginMetrics for that instance. In both cases, this method returns an empty list to
indicate no configured instances.
 * @param key The configuration key for the class.
 * @param t The interface the class should implement.
 * @param configOverrides Configuration overrides to use.
 * @param metrics The metrics registry to use to build the PluginMetrics instances for each class that
implements Monitorable.
 * @return The list of configured instances.
 */
public <T> List<T> getConfiguredInstances(String key, Class<T> t, Map<String, Object> configOverrides, Metrics
metrics);

/**
 * Get a list of configured instances of the given class specified by the given configuration key. The
configuration
 * may specify either null or an empty string to indicate no configured instances. If an instance implements
Monitorable,
 * set the appropriate PluginMetrics for that instance. In both cases, this method returns an empty list to
indicate no configured instances.
 * @param key The configuration key for the classes.
 * @param classNames The list of class names of the instances to create.
 * @param t The interface the class should implement.
 * @param configOverrides Configuration overrides to use.
 * @param metrics The metrics registry to use to build the PluginMetrics instances for each class that
implements Monitorable.
 * @return The list of configured instances.
 */
public <T> List<T> getConfiguredInstances(String key, List<String> classNames, Class<T> t, Map<String, Object>
configOverrides, Metrics metrics)
```

The Converter interface will extend Closeable (like HeaderConverter already does), the logic of the Converter class stays untouched. The Connect runtime will be updated to call close() when the converter instances can be released.

Converter.java

```
public interface Converter extends Closeable {

    @Override
    default void close() throws IOException {
        // no-op
    }
}
```

Proposed Changes

When instantiating a class, if it implements `Monitorable`, the `withPluginMetrics()` method will be called. If the class is also `Configurable`, `withPluginMetrics()` will be always called after `configure()`. Metrics registered by plugins will inherit the prefix/namespace from the current `Metrics` instance, these are: kafka.producer, kafka.consumer, kafka.connect, kafka.streams and kafka.server. Tags will be added to metrics and sensors tags created via the PluginMetrics interface to unique identify each instance.

For all plugins apart from Connectors, Tasks, Converters, Transformations and Predicates, a tag containing the configuration name (config) will be added. For example, metrics registered by a custom Serializer named MySerializer configured via key.serializer will have the following name: kafka.producer:type=plugins,client-id=producer-1,config=key.serializer,class=MySerializer,

For Connectors and Converters, the name of the connector will be added as a tag (connector). Tasks will add the name of the connector (connector) and the task id (task) as tags. Transformations and Predicates will have the connector name, the task id and their alias (transformation/predicate) added as tags. For example:

- for a task: kafka.connect:type=plugins,connector=my-sink,task=0
- for a predicate: kafka.connect:type=plugins,connector=my-sink,task=0,predicate=my-predicate

For configurations that accept a list of classes, for example interceptor.classes, if the same class is provided multiple times, their metrics may collide. This is deemed highly unlikely to occur as there are no use cases where providing multiple times the same class is useful.

This proposal supersedes KIP-608 which only aimed at providing this functionality to Authorizer plugins. KIP-608 was adopted in 2020 but never implemented. This KIP proposes to alter the names of the metrics from KIP-608 to the following so all plugins use the same naming format:

Full Name	Type	Description
kafka.server:type=plugins,config=authorizer.class.name,class=MyAuthorizer,name=acls-total-count	Gauge	Total ACLs created in the broker
kafka.server:type=plugins,config=authorizer.class.name,class=MyAuthorizer,name=authorization-request-rate-per-minute	Rate	Total number of authorization requests per minute
kafka.server:type=plugins,config=authorizer.class.name,class=MyAuthorizer,name=authorization-allowed-rate-per-minute	Rate	Total number of authorization allowed per minute
kafka.server:type=plugins,config=authorizer.class.name,class=MyAuthorizer,name=authorization-denied-rate-per-minute	Rate	Total number of authorization denied per minute

Supported plugins

The goal is allow this feature to be used by all plugins that are Closeable, AutoCloseable or have a close() methods apart from MetricsReporter since instances are created before the Metrics instance. Also all Connect connector plugins will be supported. The javadoc of all supported plugins will be updated to mention they are able to implement the Monitorable interface to define their own metrics.

Common

- ConfigProvider: config.providers
- AuthenticateCallbackHandler: sasl.client.callback.handler.class, sasl.login.callback.handler.class, sasl.server.callback.handler.class
- Login: sasl.login.class
- SslEngineFactory: ssl.engine.factory.class

Broker

- KafkaPrincipalBuilder: principal.builder.class
- ReplicaSelector: replica.selector.class
- AlterConfigPolicy: alter.config.policy.class.name
- Authorizer: authorizer.class.name
- ClientQuotaCallback: client.quota.callback.class
- CreateTopicPolicy: create.topic.policy.class.name
- RemoteLogMetadataManager: remote.log.metadata.manager.class.name
- RemoteStorageManager: remote.log.storage.manager.class.name

Producer

- Serializer: key.serializer, value.serializer
- Partitioner: partitioner.class
- ProducerInterceptor: interceptor.classes

Consumer

- Deserializer: key.deserializer, value.deserializer
- ConsumerInterceptor: interceptor.classes

Connect

- ConnectorClientConfigOverridePolicy: connector.client.config.override.policy
- Converter: key.converter, value.converter
- HeaderConverter: header.converter
- ConnectRestExtension: rest.extension.classes
- Connector
- Task
- Transformation
- Predicate

Streams

- Serde: default.key.serde, default.list.key.serde.inner, default.list.key.serde.type, default.list.value.serde.inner, default.list.value.serde.type, windowed.inner.class.serde

Unsupported Plugins

The following plugins will not support this feature:

Class	Config	Reason
KafkaMetricsReporter	kafka.metrics.reporters	This interface is technically not part of the public API. Since MetricsReporter will not support it, it makes sense to not add it to this one either.
ConsumerPartitionAssignor	partition.assignment.strategy	This interface does not have a close() method. This interface is being deprecated by KIP-848, hence I don't propose updating it.
DeserializationExceptionHandler	default.deserialization.exception.handler	This interface does not have a close() method.
ProductionExceptionHandler	default.production.exception.handler	This interface does not have a close() method.
TimestampExtractor	default.timestamp.extractor	This interface does not have a close() method.
RocksDBConfigSetter	rocksdb.config.setter	This interface does not have a close() method.
SecurityProviderCreator	security.providers	This interface does not have a close() method. The instance is passed to java.security.Security and we don't control its lifecycle.
ReplicationPolicy	replication.policy.class	MirrorMaker currently uses its own mechanism to emit metrics. This interface does not have a close() method.
ConfigPropertyFilter	config.property.filter.class	MirrorMaker currently uses its own mechanism to emit metrics.
GroupFilter	group.filter.class	MirrorMaker currently uses its own mechanism to emit metrics.
TopicFilter	topic.filter.class	MirrorMaker currently uses its own mechanism to emit metrics.
ForwardingAdmin	forwarding.admin.class	MirrorMaker currently uses its own mechanism to emit metrics.

If we decide that some of these plugins would benefit from being able to emit metrics, we could make the necessary API changes in the future to support them.

Example usage

For example if we create a custom ProducerInterceptor

```
public class MyInterceptor<K, V> implements ProducerInterceptor<K, V>, Monitorable {

    private Sensor sensor;
    private PluginMetrics metrics;

    public void withPluginMetrics(PluginMetrics metrics) {
        this.metrics = metrics;
        sensor = metrics.sensor("onSend");
        MetricName rate = metrics.metricName("rate", "Average number of calls per second.", Collections.
emptyMap());
        MetricName total = metrics.metricName("total", "Total number of calls.", Collections.emptyMap());
        sensor.add(rate, new Rate());
        sensor.add(total, new CumulativeCount());
    }

    @Override
    public ProducerRecord<K, V> onSend(ProducerRecord<K, V> record) {
        sensor.record();
        return record;
    }

    @Override
    public void close() {
        try {
            if (metrics != null) metrics.close();
        } catch (IOException e) { // Even though ProducerInterceptor extends AutoCloseable which has "void
close() throws Exception;", ProducerInterceptor has its own close() method without "throws"!
            throw new RuntimeException(e);
        }
    }
}
```

If the producer using this plugin has its client-id set to producer-1, the metrics created by this plugin will have the following name: kafka.producer:type=plugins,client-id=producer-1,config=interceptor.classes,class=MyInterceptor, and these attributes: rate and total.

Compatibility, Deprecation, and Migration Plan

This is a new feature so it has no impact on deprecation. The only significant API change is for Converter that will now be Closeable. The default close() method should ensure that existing implementations still work.

Test Plan

This feature will be tested using unit and integration tests. For each supported plugin, we will verify they can implement the withPluginMetrics() and that metrics have the correct tags associated.

Rejected Alternatives

- Create a dedicated Metrics instance for plugins: A dedicated instance could have its own prefix/namespace (for example kafka.consumer.plugins). This would allow grouping metrics from all plugins but it requires instantiating another Metrics instance and new metrics reporters.
- Let plugins create their own Metrics instance: Instead of passing the Metrics instance to plugins we could pass all the values necessary (metrics reporters, configs, etc ...) to create and configure a Metrics instance. This is impractical as it requires passing a lot of values around and plugins still have to have logic to use them.
- Provide the Metrics instance to Kafka Connect Connectors and Tasks via their context: We would have 2 different mechanisms, one for Connectors/Tasks and one of all other plugins, for the same feature. Also using the Connector and Task contexts has an impact on compatibility.
- Update MirrorMaker to use this new mechanism for creating its metrics. This will cause metrics to have different names. If needed this should be tackled in a separate KIP.