KIP-714: Client metrics and observability

Status

Current state: *Approved*

Discussion thread: *here and now here*

JIRA: KAFKA-15601

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

Motivation

Being able to centrally monitor and troubleshoot problems with Kafka clients is becoming increasingly important as the use of Kafka is expanding within organizations as well as for hosted Kafka services. The typical Kafka client user is now an application owner with little experience in operating Kafka clients, while the cluster operator has profound Kafka knowledge but little insight in the client application. This is of particular importance for hosted Kafka services.

Troubleshooting Kafka problems is currently an organizationally complex issue, with different teams or even organizations running the client applications and the brokers. While some organizations may already have custom collection of existing client metrics in place, most do not and metrics are typically not available when the problem needs to be analysed. Enabling metrics after-the-fact may not be possible without code change to the application, or at least a restart, which typically means the required metrics data is lost.

While the broker already tracks request-level metrics for connected clients, there is a gap in the end-to-end monitoring when it comes to visibility of client internals, be it queue sizes, internal latencies, error counts, and application behavior such as message processing rates. These are covered by Kafka client metrics.

This proposal aims to provide a generic and standardized interface through which a cluster operator can request metrics, clients can push metrics to the broker, a plugin interface for how those metrics are handled by the broker, and a minimum set of standard pre-defined metrics that a Kafka client should expose through this interface. A supporting client should have metrics enabled by default and not require any extra configuration, and will provide at least the required set of standard metrics as outlined later in this proposal.

One of the key goals of this KIP is to have the proposed metrics and telemetry interface generally available and enabled by default in all of the mainstream Kafka clients, not just the Java client in the Apache Kafka project, allowing troubleshooting and monitoring as needed without interaction from cluster end-users. While metrics are to be enabled by default on the clients, the brokers still need to be configured with a metrics plugin, and metrics subscriptions must be configured on the cluster before any metrics are sent and collected. The default configuration for an Apache Kafka cluster does not include any metrics subscriptions.

This KIP defines two kinds of metrics: **standard metrics** and **required metrics**.

A **standard metric** is a Kafka client metric which clients that support this KIP should support. The standard metrics are a subset of the Kafka client metrics, selected because they measure concepts which are likely to apply well to any client which implements the Kafka protocol. The Apache Kafka Java client does by definition support all of the standard metrics. Other client implementations should support standard metrics which conceptually make sense.

A **required metric** is a Kafka client metric which clients that support this KIP must support. The required metrics are a subset of the standards metrics, selected because they have the highest value for monitoring and troubleshooting client performance and whose concepts apply well to all clients which implement the Kafka protocol. The Apache Kafka Java client does by definition support all of the required metrics.

The Apache Kafka client provides the reference implementation of these metrics. To implement these metrics in other clients, the definition of the metrics is provided by the Apache Kafka documentation and ultimately the code.

Defining standard and required metrics makes the monitoring and troubleshooting of clients from various client types because the operator can combine the same metric from all client types. For example, request latencies for all client types are reported in a consistent way. Care has been taken to make the standard metrics as generic as possible, and they should fit most Kafka client implementations.

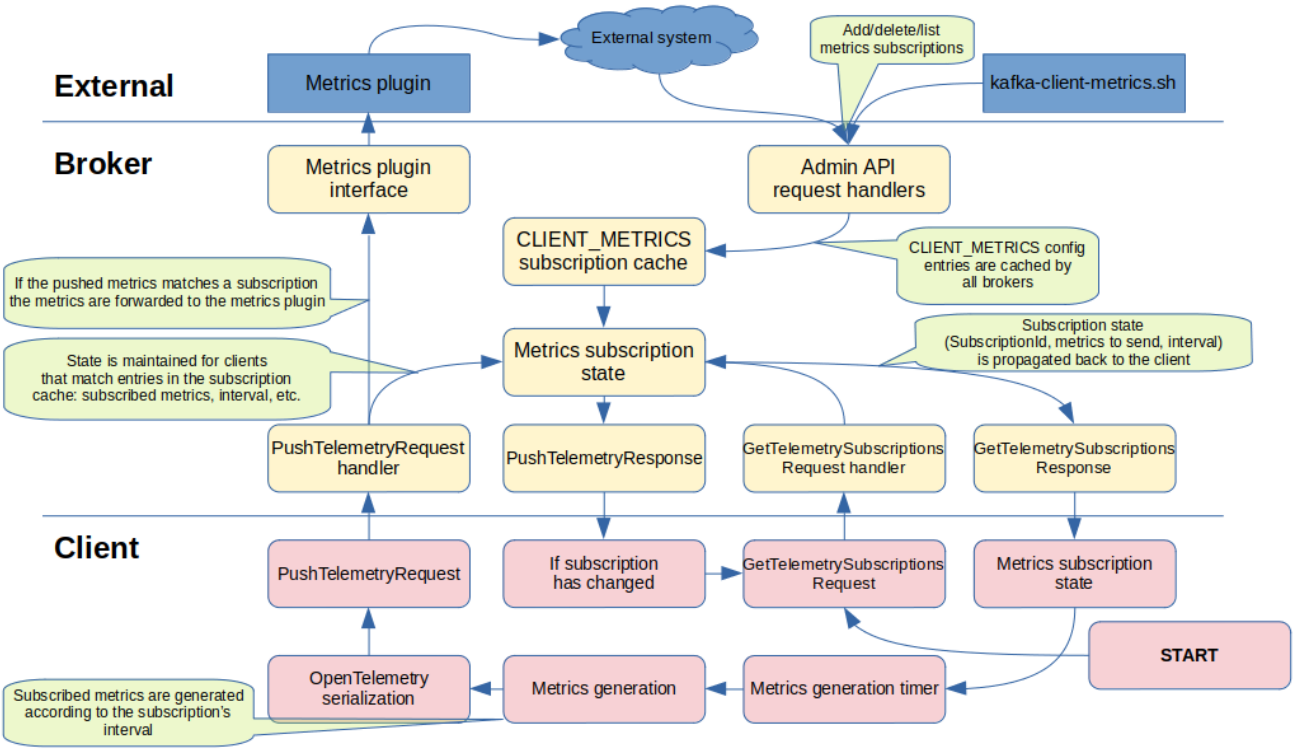
User privacy is an important concern and extra care is taken in this proposal to not expose any information that may compromise the privacy of the client user.

Proposed Changes

Overview

This feature is made up of the following components:

- **GetTelemetrySubscriptions RPC** - protocol request used by the client to acquire its initial Client instance ID and to continually retrieve updated metrics subscriptions.
- **PushTelemetry RPC** - protocol request used by the client to push metrics to any broker it is connected to.
- **Standard and required metrics** - a set of standardized metrics that all supporting clients should provide.
- **AdminAPI config** - the AdminAPI configuration interface with a new `CLIENT_METRICS` resource type is used to manage metrics subscriptions.
- **Client metrics plugin / extending the MetricsReporter interface** - a broker plugin interface that performs something meaningful with the metrics. This plugin will typically forward the metrics to a time-series database. It is recommended that any broker-side processing is kept to a minimum for scalability and performance reasons.



The metric collection is opt-in on the broker and opt-out on the client.

For metrics to be collected, a `MetricsPlugin` which implements the `ClientTelemetry` interface must be configured on the brokers, and at least one metrics subscription must be configured through the Admin API. Only then will metrics subscriptions be propagated to clients, and only then will clients push metrics to the broker. It is thus up to the cluster operator to explicitly enable client metrics collection.

Because this KIP includes a new kind of ConfigResource, this KIP is only supported for KRaft clusters. This avoids doing the work to store these new resources in ZooKeeper, given that ZooKeeper support is already deprecated.

Client metrics subscription

A client metrics subscription is the way that the operator chooses which client metrics to collect from the client. It consists of:

- A list of metric name prefixes (in the telemetry metric name format described later)
- The push interval which specifies the frequency of pushing the metrics from the client
- Matching criteria which can be used to filter which clients match the subscription, such as client version

Client identification and the client instance id

The value of metrics collection is limited if the metrics can't be tied to an entity, in this case a client instance. While aggregate metrics do provide some value in trends and superficial monitoring, for metrics to be truly useful, there needs to be a way to identify a particular client instance.

While Kafka has a per-request client.id (which most Kafka clients allow to be optionally configured), it is in no way unique and can't be used, despite its name, to identify a single client instance.

Other seemingly unique identifiers such as authentication principals or client IP source address and port are either not guaranteed to be unique (there may not be authentication involved or not specific to a single client instance), or not a singleton (a client will have multiple connections to the cluster, each connection with its own unique address+port combination).

Since this KIP intends to have metrics enabled by default in supporting clients, we can't rely on the client.id being properly set by the application owner. We have to resort to some other mechanism for a client instance to uniquely identify itself. There are ideally two entities that we would like to identify in metrics:

- Client instance - a single instance of a producer, consumer, Admin client, etc.
- Application process instance - a single application runtime, which may contain multiple client instances such as a Kafka Streams application with a mix of Admin, producer and consumer instances.

As part of the initial metrics handshake, the broker generates a unique **client instance id** and returns it to the client. The client must use the returned client instance id for the remaining lifetime of the client instance, regardless of which broker it communicates with. The client instance id is therefore a temporary, unique, generated identifier for identifying a client instance for the purpose of metrics collection.

The Kafka Java client provides an API for the application to read the generated client instance id to assist in mapping and identification of the client based on the collected metrics. Other clients implementing this proposal could provide an equivalent interface or use an alternative such as logging the client instance id as soon as it has acquired it from the cluster. This allows live and post-mortem correlation between collected metrics and a client instance.

As it is not feasible for a Kafka client instance to automatically generate or acquire a unique identity for the application process it runs in, and as we can't rely on the user to configure one, we treat the application instance id as an optional future nice-to-have that may be included as a metrics label if it has been set by the user. This allows a more direct mapping of client instance to application instance and vice versa. However, due to these constraints and the need for zero-configuration on the client, adding an application instance id configuration property is outside the scope of this proposal.

Mapping

Mapping the client instance id to an actual application instance running on a (virtual) machine can be done by inspecting the metrics resource labels, such as the client source address and source port, or security principal, all of which are added by the receiving broker. This will allow the operator together with the user to identify the actual application instance.

Metrics naming and format

The OpenTelemetry specification will be used as the metrics serialization format as well as to define how metrics are constructed when it comes to naming, metric type, semantics, and so on.

See the OpenTelemetry metrics specification for more information.

Encoding

While future protocol-level requests could allow for other metrics formats, Kafka will only guarantee to support OpenTelemetry. It is required for a client that claims support of this KIP to use the OpenTelemetry v0.19 protobuf format.

Metric payloads are encoded as OpenTelemetry MetricsData v1 protobuf objects.

Naming

This KIP introduces a new naming scheme for Kafka client metrics called **telemetry metric names**. Client metric subscriptions are defined in terms of telemetry metric names. Telemetry metric names follow the OpenTelemetry conventions for metric naming. Every existing Kafka client metric's name can be transformed into an equivalent telemetry metric name.

This proposal does not define any new client metrics, it just introduces a new scheme for naming them. The telemetry metric name format is being introduced to give a hierarchical naming convention that meets OpenTelemetry rules, enabling interoperability with that ecosystem.

The existing Kafka client metric names consist of two parts: a group name and a metric name. The telemetry metric name is derived by concatenation of:

- The prefix `"org.apache.kafka."`
- The group name with `"-metrics"` removed, and every '-' replaced with '.'
- The metric name with every '-' replaced with '.'

The following examples illustrate the derivation of the telemetry metric names from Kafka metric names:

Kafka metric name	Telemetry metric name
"connection-creation-rate", group="producer-metrics"	`"org.apache.kafka.producer.connection.creation.rate"`
"rebalance-latency-max", group="consumer-coordinator-metrics"	`"org.apache.kafka.consumer.coordinator.rebalance.latency.max"`

Other vendor or implementation-specific metrics can be added according to the following examples, using an implementation-specific reverse domain name as the namespace to avoid clashes:

Implementation-specific metric name	Telemetry metric name
Client "io.confluent.librdkafka" Metric name "client.produce.xmitq.latency"	`"io.confluent.librdkafka.client.produce.xmitq.latency"`
Python client "com.example.client.python" Metric name "object.count"	`"com.example.client.python.object.count"`

Metrics may also hold any number of attributes which provide the multi-dimensionality of metrics. These are similarly derived from the tags of the Kafka metrics, and thus the properties of the equivalent JMX MBeans, replacing '-' with '_'. For example:

Kafka metric name	Telemetry metric name

"request-latency-avg", group="producer-node-metrics", client-id={client-id}, node-id={node-id}	`"org.apache.kafka.producer.node.request.latency.avg"` Attribute keys: `"client_id"` and `"node_id"`

Sparse metrics

To keep metrics volume down, it is recommended that a client only sends metrics with a recorded value.

Metric aggregation temporality

Metrics are to be sent as either DELTA or CUMULATIVE values, depending on the value of DeltaTemporality in the GetTelemetrySubscriptionsResponse. Clients must support both temporalities.

CUMULATIVE metrics allow for missed/dropped transmits without loss of precision at the cost of increased processing and complexity required in upstream systems.

See OTLP specification for more information on temporality.

Serialized size

As an example, the serialized size prior to compression of all producer and standard metrics defined in this KIP for a producer producing to 50 partitions gives approximately 100 kB.

Compression

As metric names and labels are highly repetitive, it is recommended that the serialized metrics are compressed prior to sending them to the broker if the serialized uncompressed size exceeds 1000 bytes.

The broker will return a prioritized list of supported compression types in the **GetTelemetrySubscriptionsResponse.AcceptedCompressionTypes** array, the client is free to pick any supported compression type but should pick the first mutually supported type in the returned list. If the AcceptedCompressionTypes array is empty the client must send metrics uncompressed. The default compression types list as returned from the broker should be: *ZStd, LZ4, GZip, Snappy*.

The **PushTelemetryRequest.CompressionType** must then be set to the corresponding compression type value as defined for MessageV2.

Preliminary tests indicate that a compression ratio up to 10x is possible for the standard metrics using ZStd.

Decompression of the metrics data will be performed by the broker prior to passing the data to the metrics plugin.

Client metrics and metric labels

This KIP defines two kinds of metrics: **standard metrics** and **required metrics**.

A **standard metric** is a Kafka client metric which clients that support this KIP should support. The standard metrics are a subset of the Kafka client metrics, selected because they measure concepts which are likely to apply well to any client which implements the Kafka protocol. The Apache Kafka client does by definition support all of the standard metrics. Other client implementations should support standard metrics which conceptually make sense.

A **required metric** is a Kafka client metric which clients that support this KIP must support. The required metrics are a subset of the standards metrics, selected because they have the highest value for monitoring and troubleshooting client performance and whose concepts apply well to all clients which implement the Kafka protocol. The Apache Kafka client does by definition support all of the required metrics.

Defining standard and required metrics makes the monitoring and troubleshooting of clients from various client types easier because the operator can combine the same metric from all client types. For example, request latencies for all client types are reported in a consistent way. Care has been taken to make these standard metrics as generic as possible, and they should fit most Kafka client implementations.

There are many more Kafka client metrics beyond the standard and required metrics, some of which are quite specific to the way that the Java client works. Any of these other metrics could conceivably be included in client metrics subscriptions, but there's no expectation that non-Java clients would implement them.

Metric types

The OTLP metric data point types in the following tables correspond to the OpenTelemetry v1 metrics protobuf message types. A short summary:

- Sum - Monotonic total count meter (Counter). Suitable for total number of X counters, e.g., total number of bytes sent.
- Gauge - Non-monotonic current value meter (UpDownCounter). Suitable for current value of Y, e.g., current queue count.

Standard producer metrics

All standard telemetry metric names begin with the prefix `"org.apache.kafka."`. This is omitted from the table for brevity. The required metrics are **bold**

Telemetry metric name	OTLP metric data point type	Labels	Description	Existing Kafka metric name
`producer.connection.creation.rate`	Gauge		The rate of connections established per second.	“connection-creation-rate”, group=” producer-metrics”
`producer.connection.creation.total`	Sum		The total number of connections established.	“connection-creation-total”, group=” producer-metrics”
`producer.node.request.latency.avg`	Gauge	node_id	The average request latency in ms for a node.	“request-latency-avg”, group=”producer-node-metrics”
`producer.node.request.latency.max`	Gauge	node_id	The maximum request latency in ms for a node.	“request-latency-max”, group=” producer-node-metrics”
`producer.produce.throttle.time.avg`	Gauge		The average time in ms a request was throttled by the broker.	“produce-throttle-time-avg”, group= “producer-metrics”
`producer.produce.throttle.time.max`	Gauge		The maximum time in ms a request was throttled by the broker.	“produce-throttle-time-max”, group= “producer-metrics”
`producer.record.queue.time.avg`	Gauge		The average time in ms record batches spent in the send buffer.	“record-queue-time-avg”, group= “producer-metrics”
`producer.record.queue.time.max`	Gauge		The maximum time in ms record batches spent in the send buffer.	“record-queue-time-max”, group= “producer-metrics”

Standard consumer metrics

All standard telemetry metric names begin with the prefix `"org.apache.kafka."`. This is omitted from the table for brevity. The required metrics are **bold**.

Telemetry metric name	OTLP metric data point type	Labels	Description	Existing metric name
`consumer.connection.creation.rate`	Gauge		The rate of connections established per second.	“connection-creation-rate”, group= “consumer-metrics”
`consumer.connection.creation.total`	Sum		The total number of connections established.	“connection-creation-total”, group=”consumer-metrics”
`consumer.node.request.latency.avg`	Gauge	node_id	The average request latency in ms for a node.	“request-latency-avg”, group= “consumer-node-metrics”
`consumer.node.request.latency.max`	Gauge	node_id	The maximum request latency in ms for a node.	“request-latency-max”, group= “consumer-node-metrics”
`consumer.poll.idle.ratio.avg`	Gauge		The average fraction of time the consumer’s poll() is idle as opposed to waiting for the user code to process records.	“poll-idle-ratio-avg”, group= “consumer-metrics”
`consumer.coordinator.commit.latency.avg`	Gauge		The average time taken for a commit request.	“commit-latency-avg”, group= “consumer-coordinator-metrics”
`consumer.coordinator.commit.latency.max`	Gauge		The maximum time taken for a commit request.	“commit-latency-max”, group= “consumer-coordinator-metrics”
`consumer.coordinator.assigned.partitions`	Gauge		The number of partitions currently assigned to this consumer.	“assigned-partitions”, group= “consumer-coordinator-metrics”
`consumer.coordinator.rebalance.latency.avg`	Gauge		The average time taken for group rebalance.	“rebalance-latency-avg”, group= “consumer-coordinator-metrics”
`consumer.coordinator.rebalance.latency.max`	Gauge		The maximum time taken for a group rebalance.	“rebalance-latency-max”, group= “consumer-coordinator-metrics”
`consumer.coordinator.rebalance.latency.total`	Sum		The total time taken for group rebalances.	“rebalance-latency-total”, group= “consumer-coordinator-metrics”
`consumer.fetch.manager.fetch.latency.avg`	Gauge		The average time taken for a fetch request.	“fetch-latency-avg”, group= “consumer-fetch-manager-metrics”
`consumer.fetch.manager.fetch.latency.max`	Gauge		The maximum time taken for a fetch request.	“fetch-latency-max”, group= “consumer-fetch-manager-metrics”

Standard client resource labels

The following labels should be added by the client as appropriate before metrics are pushed.

Label name	Description
client_rack	client.rack (if configured)

group_id	group.id (consumer)
group_instance_id	group.instance.id (consumer)
group_member_id	group member id (if any, consumer)
transactional_id	transactional.id (producer)

Broker-added resource labels

The following labels should be added by the broker plugin as metrics are received.

Label name	Description
client_instance_id	The generated CLIENT_INSTANCE_ID.
client_id	client.id as reported in the Kafka protocol header.
client_software_name	The client's implementation name as reported in ApiVersionRequest.
client_software_version	The client's version as reported in ApiVersionRequest.
client_source_address	The client connection's source address.
client_source_port	The client connection's source port.
principal	Client's security principal. Content depends on authentication method.
node_id	Receiving broker's node-id.

If the received metrics have resource labels which clash with those added by the broker, the broker's overwrite the received values.

Client behavior

A client that supports this metric interface and identifies a supporting broker (through detecting at least GetTelemetrySubscriptionsRequestV0 in the ApiVersionResponse) will start off by sending a GetTelemetrySubscriptionsRequest with the ClientInstanceId field set to Null to one randomly selected connected broker to gather its client instance id, the subscribed metrics, the push interval, accepted compression types, and so on. This handshake with a Null ClientInstanceId is **only performed once** for a client instance's lifetime. Subsequent GetTelemetrySubscriptionsRequests must include the ClientInstanceId returned in the first response, regardless of broker.

If a client attempts a subsequent handshake with a Null ClientInstanceId, the receiving broker may not already know the client's existing ClientInstanceId. If the receiving broker knows the existing ClientInstanceId, it simply responds the existing value back to the client. If it does not know the existing ClientInstanceId, it will create a new client instance ID and respond with that.

Upon receiving the GetTelemetrySubscriptionsResponse, the client shall update its internal metrics collection to match the received subscription (absolute update) and update its push interval timer according to the received PushIntervalMs. The first metrics push should be randomized between 0.5 * PushIntervalMs and 1.5 * PushIntervalMs. This is to ensure that not all clients start pushing metrics at the same time after a cluster comes back up after some downtime.

If GetTelemetrySubscriptionsResponse.RequestedMetrics indicates that no metrics are desired (RequestedMetrics is Null), the client should send a new GetTelemetrySubscriptionsRequest after the PushIntervalMs has expired. This is to avoid having to restart clients if the cluster metrics configuration is disabled temporarily by operator error or maintenance such as rolling upgrades. The default PushIntervalMs is 300000 ms (5 minutes).

If GetTelemetrySubscriptionsResponse.RequestedMetrics is non-empty but does not match any metrics the client provides, then the client should send a PushTelemetryRequest at the indicated PushIntervalMs interval with an empty metrics blob. This is needed so that a broker metrics plugin can differentiate between non-responsive or buggy clients and clients that don't have metrics matching the subscription set.

Threading

The client will re-use the existing threads that are used for network communication. The existing logic in the `poll` method of the `NetworkClient` class issues internal network requests for metadata when needed. The `NetworkClient` class has been augmented to now also issue internal network requests for telemetry based on the poll interval in the subscription.

The threads that invoke the `NetworkClient`'s `poll` method are:

- `KafkaAdminClient`: the "admin client runnable" thread
- `KafkaConsumer`: (existing code): both the "heartbeat" and the application threads
- `KafkaConsumer`: (consumer threading refactor code as a part of KIP-848 effort): the "background" threads
- `KafkaProducer`: the "sender" thread

Connection selection

The client may send telemetry requests to any broker, but shall prefer using an already available connection rather than creating a new connection to keep the number of cluster connections down.

It should also keep using the same broker connection for telemetry requests until the connection goes down, at which time it may choose to reconnect and continue using the same broker, or switch over to another broker connection. Using a persistent connection for PushTelemetryRequests is important so that metrics throttling can be properly performed by the receiving broker, and also avoids maintaining metrics state for the client instance id on multiple brokers.

Client termination

When a client with an active metrics subscription is being shut down, it should send its final metrics without waiting for the PushIntervalMs time.

To avoid the receiving broker's metrics rate-limiter discarding this out-of-profile push, the `PushTelemetryRequest.Terminating` field must be set to true. A broker must only allow one such unthrottled metrics push for each combination of client instance ID and SubscriptionId.

In the event that the client's metric subscription has changed and the final metrics push fails with error code `UNKNOWN_SUBSCRIPTION_ID` , the terminating client can choose to obtain a new subscription ID by sending a GetTelemetrySubscriptionsRequest and then immediately sending a PushTelemetryRequest with the Terminating flag set to true, or it can choose to abandon sending a final metrics push.

Error handling

The following error codes can be returned in `PushTelemetryResponse` .

Error code	Reason	Client action
`INVALID_REQUEST` (42)	Client sent a PushTelemetryRequest when it has already sent a Terminating request.	Log an error and stop pushing metrics. This is viewed as a problem in the client implementation of metrics serialization that is not likely to be resolved by retrying.
`INVALID_RECORD` (87)	Broker failed to decode or validate the client's encoded metrics.	Log an error and stop pushing metrics. This is viewed as a problem in the client implementation of metrics serialization that is not likely to be resolved by retrying.
`TELEMETRY_TOO_LARGE` (NEW)	Client sent a PushTelemetryRequest larger than the maximum size the broker will accept.	Reduce the size of the metrics payload so its size does not exceed GetTelemetrySubscriptionsResponse.TelemetryMaxBytes.
`UNKNOWN_SUBSCRIPTION_ID` (NEW)	Client sent a PushTelemetryRequest with an invalid or outdated SubscriptionId. The configured subscriptions have changed.	Immediately send a GetTelemetrySubscriptionsRequest to update the client's subscriptions and get a new SubscriptionId.
`UNSUPPORTED_COMPRESSION_TYPE` (76)	Client's compression type is not supported by the broker.	Immediately send a GetTelemetrySubscriptionsRequest to get an up-to-date list of the broker's supported compression types (and any subscription changes).

Retries should preferably be attempted on the same broker connection, in particular for `UNKNOWN_SUBSCRIPTION_ID`, but another broker connection may be utilized at the discretion of the client.

How errors and warnings are propagated to the application is client- and language-specific. Simply logging the error is sufficient.

Java client dependencies

The OpenTelemetry metrics serialization is used as the payload of the telemetry metrics sent to the broker. This will require us to include the Java Bindings for the OpenTelemetry Protocol (OTLP) as a dependency for the Java client.

implementation("io.opentelemetry.proto:opentelemetry-proto:0.19.0-alpha")

The OTLP Java bindings library is a minimal dependency and itself requires only the Google Protobuf serialization library:

```
implementation("com.google.protobuf:protobuf-java:3.18.0")
```

This will allow us to build up Java objects in memory that reflect that specification and easily serialize them to the required transport wire format.

The Gradle build process will be updated to *shadow* the OTLP Java bindings library and its dependencies to avoid in-JVM versioning conflicts.

Receiving broker behavior

We allow clients to send the GetTelemetrySubscriptions and PushTelemetry requests to any connected broker that reports support for both these APIs in its ApiVersionResponse.

If `GetTelemetrySubscriptionsRequest.ClientInstanceId` is Null the broker will generate a unique id for the client and return it in the response. Subsequent requests from the client to **any broker** must have the ClientInstanceId set to this returned value.

The broker should add additional metrics labels to help identify the client instance before forwarding the metrics to an external system. These are labels such as the namespace, cluster id, client principal, client source address and port. As the metrics plugin may need to add additional information on top of this, the generic metrics receiver in the broker will not add these labels but rely on the plugins to do so. This avoids deserializing and serializing the received metrics multiple times in the broker.

See **Broker-added labels** below for the list of labels that should be added by the plugin.

The broker only reports support for GetTelemetrySubscriptions and PushTelemetry requests in its ApiVersionResponse if it has a `MetricsReporter` that implements the `ClientTelemetry` interface. This means that clients will not attempt to push metrics to brokers that are not capable of receiving them. If there is no client metrics receiver plugin configured on the broker, the client will not send GetTelemetrySubscriptions or PushTelemetry RPCs, and existing behavior is preserved.

PushTelemetryRequest handling

Validation

Validation of the encoded metrics is the task of the ClientMetricsReceiver, if the compression type is unsupported the response will be returned with ErrorCode set to UnsupportedCompressionType. Should decoding or validation of the binary metrics blob fail the ErrorCode will be set to InvalidRecord.

Throttling and rate-limiting

There are two mechanisms at play to protect brokers from rogue or buggy clients that:

1. Standard request throttling - will mute the client connection if user quotas (size and/or request rate) are exceeded.
2. Metrics PushIntervalMs rate-limiting - ensures the client does not push telemetry more often than the configured PushIntervalMs (subscription interval). As this rate-limiting state is maintained by each broker the client is sending telemetry requests to it is possible for the client to send at most one accepted out-of-profile per connection before the rate-limiter kicks in. The metrics plugin itself may also put constraints on the maximum allowed metrics payload.

The receiving broker's standard quota-based throttling should operate as usual for PushTelemetryRequest, but in addition to that the PushTelemetryRequest is also subject to rate-limiting based on the calculated next desired PushIntervalMs interval derived from the configured metrics subscriptions. Should the client send a push request prior to expiry of the previously calculated PushIntervalMs the broker will discard the metrics and return a PushTelemetryResponse with the ErrorCode set to THROTTLING_QUOTA_EXCEEDED.

The one exception to this rule is when the client sets the PushTelemetryRequest.Terminating field to true indicating that the client is terminating, in this case the metrics should be accepted by the broker, but a consecutive request must ignore the Terminating field and apply rate-limiting as if the field was not set. The Terminating flag may be reused upon the next expiry of PushIntervalMs.

In case the cluster load induced from metrics requests becomes unmanageable the remedy is to temporarily remove or limit configured metrics subscriptions.

Metrics subscription

Metrics subscriptions are configured through the standard Kafka Admin API configuration interface with the new config resource type `CLIENT_METRICS` .

The ConfigResource has a name and is of type `CLIENT_METRICS` . The resource name does not have significance to the metrics system other than to group metrics subscriptions in the configuration interface.

The configuration is made up of the following ConfigEntry names:

- `metrics` - a comma-separated list of telemetry metric name prefixes, e.g., `"org.apache.kafka.producer.node.request.latency., org.apache.kafka.consumer.coordinator.rebalance.latency.max"`. Whitespace is ignored.
- `interval.ms` - metrics push interval in milliseconds. Defaults to 300000 ms (5 minutes) if not specified.
- `match` - Client matching selector that is evaluated as a list of an anchored regular expressions (i.e., "something.*" is treated as "^something.*$"). Any client that matches **all** of the selectors will be eligible for this metrics subscription. The regular expressions are compiled and executed using Google RE2/J. Initially supported selectors are:
    - `client_instance_id` - CLIENT_INSTANCE_ID UUID string representation.
    - `client_id` - client's reported client.id in the GetTelemetrySubscriptionsRequest.
    - `client_software_name` - client software implementation name.
    - `client_software_version` - client software implementation version.
    - `client_source_address` - client connection's source address from the broker's point of view.
    - `client_source_port` - client connection's source port from the broker's point of view.

For example, using the standard `kafka-configs.sh` tool for create a metrics subscription:

```
 $ kafka-configs.sh --bootstrap-server $BROKERS \
        --entity-type client-metrics \
        --entity-name "basic_producer_metrics" \
        --alter \
        --add-config "metrics=[org.apache.kafka.producer., org.apache.kafka.consumer.coordinator.rebalance.
latency.max],interval.ms=15000,match=[client_instance_id=b69cc35a-7a54-4790-aa69-cc2bd4ee4538]"
```

There is also a new `kafka-client-metrics.sh` tool which is described later that has easier syntax.

As the assumption is that the number of CLIENT_METRICS configuration entries will be relatively small (<100), all brokers with a configured metrics plugin will monitor and cache the configuration entries for the CLIENT_METRICS resource type.

As a GetTelemetrySubscriptionsRequest is received for a previously unknown client instance id, the CLIENT_METRICS config cache is scanned for any configured metric subscriptions whose match selectors match that of the client. The resulting matching configuration entries are compiled into a list of subscribed metrics which is returned in GetTelemetrySubscriptionsResponse.RequestedMetrics along with the minimum configured collection interval (this can be improved in future versions by including a per-metric interval so that each subscribed metric is collected with its configured interval, but in its current form longer-interval metrics are included “for free” if there are shorter-interval metrics in the subscription set). The CRC32C checksum is also calculated based on the compiled metrics and is returned as the SubscriptionId in the response, as well as stored in the per-client-instance cache on the broker to track configuration changes.

This client instance specific state is maintained in broker memory up to MAX(60*1000, PushIntervalMs * 3) milliseconds and is used to enforce the push interval rate-limiting. There is no persistence of client instance metrics state across broker restarts or between brokers.

Public Interfaces

Kafka Protocol Changes

These new RPCs are only supported on KRaft clusters.

```
GetTelemetrySubscriptionsRequestV0 {
         ClientInstanceId uuid                // UUID4 unique for this client instance.
                                                                                                  // Must be set to Null on the
first request, and to the returned ClientInstanceId
                                          // from the first response for all subsequent requests to any broker.
}

GetTelemetrySubscriptionsResponseV0 {
         ThrottleTimeMs int32                     // The duration in milliseconds for which the request was
throttled due to a quota violation,
                                          // or zero if the request did not violate any quota.
        ErrorCode int16                                                 // The error code, or 0 if there was no
error.
    ClientInstanceId uuid                 // Assigned client instance id if ClientInstanceId was Null in the
request, else Null.
    SubscriptionId int32                  // Unique identifier for the current subscription set for this client
instance.
    AcceptedCompressionTypes Array[int8] // The compression types the broker accepts for PushTelemetryRequest.
CompressionType
                                          // as listed in MessageHeaderV2.Attributes.CompressionType. The array
will be sorted in
                                          // preference order from higher to lower. The CompressionType of NONE
will not be
                                          // present in the response from the broker, though the broker does
support uncompressed
                                          // client telemetry if none of the accepted compression codecs are
supported by the client.
    PushIntervalMs int32                  // Configured push interval, which is the lowest configured interval
in the current subscription set.
    TelemetryMaxBytes int32               // The maximum bytes of binary data the broker accepts in
PushTelemetryRequest.
    DeltaTemporality bool                 // If True; monotonic/counter metrics are to be emitted as deltas to
the previous sample.
                                          // If False; monotonic/counter metrics are to be emitted as cumulative
absolute values.
        RequestedMetrics Array[string]                // Requested telemetry metrics prefix string match.
                                                                                                  // Empty array: No metrics
subscribed.
                                                                                                  // Array[0] "*": All metrics
subscribed.
                                                                                                  // Array[..]: prefix string
match.
}

PushTelemetryRequestV0 {
        ClientInstanceId uuid                // UUID4 unique for this client instance, as retrieved in the
first GetTelemetrySubscriptionsRequest.
    SubscriptionId int32                  // SubscriptionId from the GetTelemetrySubscriptionsResponse for the
collected metrics.
        Terminating bool                     // Client is terminating.
    CompressionType int8                  // Compression codec used for .Metrics (ZSTD, LZ4, Snappy, GZIP, None).
                                          // Same values as that of the current MessageHeaderV2.Attributes.
        Metrics binary                       // Metrics encoded in OpenTelemetry MetricsData v1 protobuf format.
}

PushTelemetryResponseV0 {
          ThrottleTimeMs int32                      // The duration in milliseconds for which the request was
throttled due to a quota violation,
                                          // or zero if the request did not violate any quota.
    ErrorCode int16                       // The error code, or 0 if there was no error.
}
```

Metrics serialization format

The metrics format for PushTelemetryRequestV0 is OpenTelemetry Protobuf protocol definitions version 0.19.

Future versions of `PushTelemetryRequest` and `GetTelemetrySubscriptionsRequest` may include a content-type field to allow for updated OTLP format versions (or additional formats), but this field is currently not included since only one format is specified by this proposal.

SubscriptionId

The SubscriptionId is a unique identifier for a client instance's subscription set, the id is generated by calculating a CRC32C of the configured metrics subscriptions matching the given client including the PushIntervalMs, XORed with the ClientInstanceId. This SubscriptionId is returned to the client in the GetTelemetrySubscriptionsResponse and the client passes that SubscriptionId in each subsequent PushTelemetryRequest for those received metrics. If the configured subscriptions are updated and resulting in a change for a client instance, the SubscriptionId is recalculated. Upon the next PushTelemetryRequest using the previous SubscriptionId, the broker will find that the received and expected SubscriptionIds differ and it will return `UNKNOWN_SUBSCRIPTION_ID` back to the client. When a client receives this error code it will immediately send a GetTelemetrySubscriptionsRequest to retrieve the new subscription set along with a new SubscriptionId.

This mechanism provides a way for the broker to propagate an updated subscription set to the client and is similar to the use of epochs in other parts of the protocol, but simplified in that no state needs to be persisted; the configured subscription set + client instance id is the identifier itself.

Client telemetry receiver

The broker will expose a plugin interface for client telemetry to be forwarded to or collected by external systems. In particular, since we already collect data in OpenTelemetry format, one goal is to make it possible for a plugin to forward metrics directly to an OpenTelemetry collector.

The existing MetricsReporter plugin interface was built for plugins to pull metrics from the broker, but does not lend itself to the model where the broker pushes metric samples to plugins.

In order to minimize the complexity of additional configuration mechanisms, we are proposing to reuse the existing metrics reporter configuration mechanisms, but allow plugins to implement a new interface (trait) to indicate if they support receiving client telemetry.

This also allows a plugin to reuse the existing labels passed through the MetricsContext (e.g. broker, cluster id, and configured labels) and add them to the OpenTelemetry resource labels as needed.

The broker only reports support for GetTelemetrySubscriptions and PushTelemetry requests in its ApiVersionResponse if it has a `MetricsReporter` that implements the `ClientTelemetry` interface. This means that clients will not attempt to push metrics to brokers that are not capable of receiving them.

```
/**
 * A {@link MetricsReporter} may implement this interface to indicate support for collecting client telemetry
on the server side
 */
@InterfaceStability.Evolving
public interface ClientTelemetry {

    /**
     * Called by the broker to create a ClientTelemetryReceiver instance.
     * This instance may be cached by the broker.
     *
     * This method will always be called after the initial call to
     * {@link MetricsReporter#contextChange(MetricsContext)}
     * on the MetricsReporter implementing this interface.
     *
     * @return
     */
    ClientTelemetryReceiver clientReceiver();
}

@InterfaceStability.Evolving
public interface ClientTelemetryReceiver {
    /**
     * Called by the broker when a client reports telemetry. The associated request context can be used
     * by the plugin to retrieve additional client information such as client ids or endpoints.
     *
     * This method should avoid blocking.
     *
     * @param context the client request context for the corresponding PushTelemetryRequest api call
     * @param payload the encoded telemetry payload as sent by the client
     */
    void exportMetrics(AuthorizableRequestContext context, ClientTelemetryPayload payload);
}

@InterfaceStability.Evolving
public interface ClientTelemetryPayload {

    /**
     * Client's instance id.
     */
    Uuid clientInstanceId();

    /**
     * Indicates whether the client is terminating and sending its last metrics push.
     */
    boolean isTerminating();

    /**
     * Metrics data content-type / serialization format.
     * Currently "application/x-protobuf;type=otlp+metrics0.19"
     */
    String contentType();

    /**
     * Serialized uncompressed metrics data.
     */
    ByteBuffer data();
}
```

Admin API

Add a new value `CLIENT_METRICS` to the enum `org.apache.kafka.common.config.ConfigResource.Type` . This enables client metrics subscription configuration to be administered using the Kafka admin client as well as the `kafka-configs.sh` tool.

Client API

Retrieve broker-generated client instance id, may be used by application to assist in mapping the client instance id to the application instance through log messages or other means.

The client instance ids returned correspond to the client_instance_id labels added by the broker to the metrics pushed from the clients. This should be suffi cient information to enable correlation between the metrics available in the client, and the metrics pushed to the broker.

The following method is added to the Producer, Consumer, and Admin client interfaces:

```
/**
 * @return the client's assigned instance id used for metrics collection.
 * @throws InterruptException       If the thread is interrupted while blocked.
 * @throws TimeoutException         Indicates that a request timed out
 * @throws KafkaException           If an unexpected error occurs while trying to determine the client
 *                                  instance ID, though this error does not necessarily imply the
 *                                  consumer client is otherwise unusable.
 * @throws IllegalArgumentException If the {@code timeout} is negative.
 * @throws IllegalStateException    If telemetry is not enabled ie, config `{@code enable.metrics.push}`
 *                                  is set to `{@code false}`.
 */
public Uuid clientInstanceId(Duration timeout);
```

If the client has not yet requested a client instance id from the broker, this call may block up to the duration of the timeout. In the event that the client instance id cannot be obtained within the timeout, the method throws `org.apache.kafka.common.errors.TimeoutException` .

In addition, the following method is added to the KafkaStreams interface to give access to the client instance ids of the producers, consumers and admin clients used by Kafka Streams:

```
/**
 * @return The internal clients' assigned instance ids used for metrics collection.
 *
 * @throws IllegalArgumentException If {@code timeout} is negative.
 * @throws IllegalStateException If {@code KafkaStreams} is not running.
 * @throws TimeoutException Indicates that a request timed out.
 * @throws StreamsException For any other error that might occur.
 */
public ClientInstanceIds clientInstanceIds(Duration timeout);
```

This method is only permitted when Kafka Streams is in state RUNNING or REBALANCING. In the event that Kafka Streams is not in state RUNNING or REBALANCING, the method throws an `IllegalStateException`.

In the event that any of the client instance ids cannot be obtained within the timeout, the method throws `org.apache.kafka.common.errors.TimeoutException` .

The new interface `org.apache.kafka.streams.ClientInstanceIds` is defined as follows:

```
/**
 * Encapsulates the client instance ids used for metrics collection by
 * producers, consumers and admin clients used by Kafka Streams.
 */
public interface ClientInstanceIds {
  /**
   * Get the client instance id of the admin client
   *
   * @return the client instance id
   *
   * @throws IllegalStateException If telemetry is disabled on the admin client.
   */
  Uuid adminInstanceId();

  /**
   * Get the client instance ids of the consumers
   *
   * @return a map from thread key to client instance id
   */
  Map<String, Uuid> consumerInstanceIds();

  /**
   * Get the client instance ids of the producers
   *
   * @return a map from thread key to client instance id
   */
  Map<String, Uuid> producerInstanceIds();
}
```

Finally, a new method is added to `org.apache.kafka.clients.CommonClientConfigs` to return the `ClientTelemetryReporter` if configured. This mirrors the similar `metricsReporters()` methods in that class.

```
public static Optional<ClientTelemetryReporter> telemetryReporter(String clientId, AbstractConfig config);
```

Broker configuration

Configuration	Description	Values
`telemetry.max.bytes`	The maximum size (after compression if compression is used) of telemetry pushed from a client to the broker.	int, default: 1048576, valid values: [1,...]

Client configuration

This applies to producers, consumers, admin client, and of course embedded uses of these clients in frameworks such as Kafka Connect.

Configuration	Description	Values
`enable.metrics.push`	Whether to enable pushing of client metrics to the cluster, if the cluster has a client metrics subscription which matches this client.	`true` (default) - The client will push metrics if there are any matching subscriptions. `false` - The client will not push metrics.

Following the usual convention, the string constant `ENABLE_METRICS_PUSH_CONFIG ("enable.metrics.push")` will be added to CommonClientConfigs, ProducerConfig, ConsumerConfig, AdminClientConfig and StreamsConfig.

Client metrics configuration

These are the configurations for client metrics resources. A client metrics subscription is defined by the configurations for a resource of type `CLIENT_METRICS` .

Configuration	Description	Values

`metrics`	A list of telemetry metric name prefixes which specify the metrics of interest.	An empty list means no metrics subscribed. A list containing just `"*"` means all metrics subscribed. Otherwise, the list entries are prefix-matched against the metric names.
`interval.ms`	The client metrics push interval in milliseconds.	Default: 300000 (5 minutes), minimum: 100, maximum: 3600000 (1 hour)
`match`	The match criteria for selecting which clients the subscription matches. If a client matches all of these criteria, the client matches the subscription.	A list of key-value pairs. The valid keys are: • `client_instance_id` - CLIENT_INSTANCE_ID UUID string representation. • `client_id` - client's reported client.id in the GetTelemetrySubscriptionsRequest. • `client_software_name` - client software implementation name. • `client_software_version` - client software implementation version. • `client_source_address` - client connection's source address from the broker's point of view. • `client_source_port` - client connection's source port from the broker's point of view. The values are anchored regular expressions.

New error codes

`TELEMETRY_TOO_LARGE` - Client sent a PushTelemetryRequest with a payload that was too large.

`UNKNOWN_SUBSCRIPTION_ID` - Client sent a PushTelemetryRequest with an invalid or outdated SubscriptionId. The configured subscriptions have changed.

Metrics

The following new broker metrics should be added:

Metric Name	Type	Group	Tags	Description
ClientMetricsInstanceCount	Gauge	ClientMetrics		Current number of client metric instances being managed by the broker. E.g., the number of unique CLIENT_INSTANCE_IDs with an empty or non-empty subscription set.
ClientMetricsUnknownSubscriptionRequestCount ClientMetricsUnknownSubscriptionRequestRate	Meter	ClientMetrics		Total number/rate of metrics requests for PushTelemetryRequests with unknown subscription id.
ClientMetricsThrottleCount ClientMetricsThrottleRate	Meter	ClientMetrics	client_instance_id	Total number/rate of throttled telemetry requests due to a higher requests rate than the allowed PushIntervalMs.
ClientMetricsPluginExportCount ClientMetricsPluginExportRate	Meter	ClientMetrics	client_instance_id	The total number/rate of metrics requests being pushed to metrics plugins, e.g., the number/rate of exportMetrics() calls.
ClientMetricsPluginErrorCount ClientMetricsPluginErrorRate	Meter	ClientMetrics	client_instance_id	The total number/rate of exceptions raised from plugin's exportMetrics().
ClientMetricsPluginExportTimeAvg ClientMetricsPluginExportTimeMax	Avg and Max	ClientMetrics	client_instance_id	Amount of time broker spends in invoking plugin exportMetrics call

Security

Since client metric subscriptions are primarily aimed at the infrastructure operator that is managing the Kafka cluster it should be sufficient to limit the config control operations to the CLUSTER resource.

There will be no permission checks on the PushTelemetryRequest itself.

API Request	Resource	ACL operation
DescribeConfigs	CLUSTER	DESCRIBE_CONFIGS
AlterConfigs	CLUSTER	ALTER_CONFIGS
GetTelemetrySubscriptions	N/A	N/A
PushTelemetry	N/A	N/A

Tools

kafka-configs.sh

The `kafka-configs.sh` tool can be used to administer client metrics configuration. The `kafka-client-metrics.sh` tool is preferred because it is more useable.

A new `entity-type` of `client-metrics` is added.

For this entity type, the allowed configs are: `interval.ms` , `metrics` and `match` .

Some examples of use of `kafka-configs.sh` are shown in the next section for comparison with `kafka-client-metrics.sh` .

kafka-client-metrics.sh

A new `kafka-client-metrics.sh` tool is added which provides a simpler interface for administering client metrics configuration resources than using k `afka-configs.sh` directly.

Here's the command-line syntax summary.

```
This tool helps to manipulate and describe client metrics configurations.
Option                                  Description
------                                  -----------
--alter                                 Alter the configuration of the client
                                          metrics resource.
--bootstrap-server <String: server to  REQUIRED: The Kafka server to connect to.
  connect to>
--command-config <String: command      Property file containing configs to be
  config property file>                   passed to Admin Client.
--delete                                Delete the configuration of the client
                                          metrics resource.
--describe                              List configurations for client metrics resources.
--generate-name                         Generate a UUID to use as the name.
--help                                  Print usage information.
--interval                              The metrics push interval in milliseconds.
--match                                 Matching selector ‘k1=v1,k2=v2’. The following
                                          is a list of valid selector names:
                                          client_instance_id
                                          client_id
                                          client_software_name
                                          client_software_version
                                          client_source_address
                                          client_source_port
--metrics                               Telemetry metric name prefixes ‘m1,m2’.
--name <String>                         Name of client metrics configuration resource.
--version                               Display Kafka version.
```

Here are some examples.

List all client metrics configuration resources

```
$ kafka-client-metrics.sh --bootstrap-server $BROKERS --describe

$ kafka-configs.sh --bootstrap-server $BROKERS --describe --entity-type client-metrics
```

Describe a client metrics configuration resources

```
$ kafka-client-metrics.sh --bootstrap-server $BROKERS --describe --name MYMETRICS

$ kafka-configs.sh --bootstrap-server $BROKERS --describe --entity-type client-metrics --entity-name MYMETRICS
```

Create a client metrics configuration resource, generating a unique name

In this example, `--generate-name` causes the tool to create a type-4 UUID to use as the client metrics configuration resource name. There is no equivalent in `kafka-configs.sh` .

```
$ kafka-client-metrics.sh --bootstrap-server $BROKERS --alter --generate-name \
  --metrics org.apache.kafka.producer.node.request.latency.,org.apache.kafka.consumer.node.request.latency. \
  --interval 60000
```

Create a client metrics configuration resource for all Python v1.2.* clients

```
$ kafka-client-metrics.sh --bootstrap-server $BROKERS --alter --name MYMETRIC \
  --metrics org.apache.kafka.consumer. \
  --interval 60000 \
  --match "client_software_name=kafka_python,client_software_version=1\.2\..*"

$ kafka-configs.sh --bootstrap-server $BROKERS --alter --entity-type client-metrics --entity-name MYMETRICS \
  --add-config "metrics=org.apache.kafka.consumer.,interval.ms=60000,match=[client_software_name=kafka.python,
client_software_version=1\.2\..*]"
```

Delete a client metrics configuration resource

Deletion of a client metrics configuration resource with `kafka-configs.sh` requires listing the names of the configs to delete.

```
$ kafka-client-metrics.sh --bootstrap-server $BROKERS --delete --name MYMETRICS

$ kafka-configs.sh --bootstrap-server $BROKERS --alter --entity-type client-metrics --entity-name MYMETRICS \
  --delete-config metrics,interval.ms,match
```

Example workflows

Example workflows showcasing how this feature may be used for proactive and reactive purposes.

Proactive monitoring

The Kafka cluster is configured to collect all standard metrics pushed by the client at an interval of 60 seconds, the metrics plugin forwards the collected metrics to an imaginary system that monitors a set of well known metrics and triggers alarms when trends go out of profile.

The monitoring system detects an anomaly for `CLIENT_INSTANCE_ID=4e6fb54c-b8e6-4517-889b-e15b99c09c20` which for more than 180 seconds has exceeded the threshold of 5000 milliseconds.

Before sending the alert to the incident management system, the monitoring system collects a set of labels that are associated with this CLIENT_INSTANCE_ID, such as:

- client.id
- client_source_address and client_source_port on broker id X (1 or more such mappings based on how many connections the client has used to push metrics).
- principal
- tenant
- client_software_name and client_software_version
- In case of consumer: group_id, group_instance_id (if configured) and the latest known group_member_id
- In case of transactional producer: transactional_id

This collection of information, along with the triggered metric, is sent to the incident management system for further investigation or aggregation, and provides enough information to identify which client and where the client is run. Further action might be to contact the organization or team that matches the principal, transactional.id, source address and so on for further investigation.

Reactive monitoring

The Kafka cluster configuration for metrics collection (i.e., metrics subscriptions) is irrelevant to this use-case, given that a metrics plugin is enabled on the brokers. The metrics plugin is configured to write metrics to a topic. A support system with an interactive interface is reading from this metrics topic, and has an Admin client to configure the cluster with desired metrics subscriptions.

The application owner reports a lagging consumer that is not able to keep up with the incoming message rate and asks for the Kafka operator to help troubleshoot. The application owner, who unfortunately does not know the client instance id of the consumer, provides the client.id, userid, and source address.

The Kafka operator adds a metrics subscription for metrics matching prefix `"org.apache.kafka.consumer."` and with the client_source_address and client_id as matching selector. Since this is a live troubleshooting case, the metrics push interval is set to a low 10 seconds.

The metrics subscription propagates through configuration change notifications to all brokers which update their local metrics subscription config cache and regenerates the SubscriptionId.

Upon the next PushTelemetryRequest or GetTelemetrySubscriptionsRequest from the clients matching the given source address and client_id, the receiving broker sees that the SubscriptionId no longer matches its metrics subscription cache, the client retrieves the new metrics subscription and schedules its next metrics push to a random value between PushIntervalMs * 0.5 .. PushIntervalMs * 1.5.

Upon the next PushTelemetryRequest, which now includes metrics for the subscribed metrics, the metrics are written to the output topic and the PushIntervalMs is adjusted to the configured interval of 10 seconds. This repeats until the metrics subscription configuration is changed.

Multiple consumers from the same source address and using the same client_id may now be pushing metrics to the cluster. The support system starts receiving the metrics and displays the matching client metrics to the operator. If the operator is able to further narrow down the client instance through other information in the metrics it may alter the metrics subscription to match on that client's client_instance_id. But in either case the metrics matching the given client.id are displayed to the operator.

The operator identifies an increasing trend in client.consumer.processing.time which indicates slow per-message processing in the application and reports this back to the application owner, ruling out the client and Kafka cluster from the problem space.

The operator removes the metrics subscription which causes the next PushTelemetryResponse to return an error indicating that the metrics subscription has changed, causing the client to get its new subscription which is now back to empty.

Future work

Tracing and logging are outside the scope of this proposal but the use of OpenTelemetry allows for those additional two methods within the same semantics and nomenclature outlined by this proposal.

When we add new client metrics to Kafka, we can choose whether to make them standard or required metrics in order to enhance the ability to analyze and troubleshoot client performance.

We could also add proper support for histogram metrics in the client, and this would nicely fit with histograms in OpenTelemetry metrics.

In network environments where there are network proxies (such as Kubernetes ingress) on the path between the client and broker, it may be problematic obtaining the originating client's IP address. One way to address this in the future would be to support the PROXY protocol in Kafka.

Configuration properties or extensions to the Metrics plugin interface on the broker to change the temporality is outside the scope of this KIP and may be addressed at a later time as the need arises. For example, if a metrics back-end has a preference for a particular temporality, it may be helpful to let it indicate that using the Metrics plugin interface so that the broker can use this temporality when requesting metrics from the clients.

Compatibility, Deprecation, and Migration Plan

What impact (if any) will there be on existing users?

Clients

Apart from the configuration property (`"enable.metrics.push"`) to disable telemetry, there are no user-facing changes to client users.

Depending on metrics subscription intervals there might be increased CPU and network load, but for modestly set subscription intervals this should be negligible.

Broker

The Kafka admin needs to explicitly configure a Metrics reporter plugin that supports the new push-based telemetry interface. If no such plugin is configured, the telemetry collection is disabled and there will be no user-facing changes. The default configuration for the Kafka broker does not configure a plugin.

Depending on the metrics subscription patterns and intervals, there is likely to be increased CPU and network load as brokers will receive, decompress and forward pushed client telemetry to the metrics plugin.

If we are changing behavior how will we phase out the older behavior?

No behavioral changes.

If we need special migration tools, describe them here.

None.

When will we remove the existing behavior?

Not relevant.

Rejected Alternatives

Create a new set of client telemetry metrics

An earlier version of this KIP introduced a new set of client telemetry metrics, completely independent of the existing Kafka client metrics. This was rejected because it was felt that the lack of correspondence between the metrics available on the client and those sent to the broker to support this KIP was confusing. For example, there were two independent ways of measuring latency which did not match.

Send metrics out-of-band directly to collector or to separate metric cluster

There are plenty of existing solutions that allows a client implementation to send metrics directly to a collector, but it falls short to meet the enabled-by-default requirements of this KIP:

- will require additional client-side configuration: endpoints, authentication, etc.
- may require additional network filtering and our routing configuration to allow the client host to reach the collector endpoints. By using the Kafka protocol we already have a usable connection.
- adds another network protocol the client needs to handle and its runtime dependencies (libraries..).
- makes correlation between client instances and connections on the broker harder - which makes correlating client side and broker side metrics harder.
- more points of failure: what if the collector is down but the cluster is up?
- **zero conf is an absolute must for KIP-714 to provide value**: It is already possible today to send metrics out-of-band, but people don't and they still won't if any extra configuration is needed.

An alternative approach is to send metrics to a different Kafka cluster, the idea is that having metrics go through the same cluster may mean client metrics may be unavailable if the original cluster is unavailable, and this suggestion would solve that by sending metrics to a different cluster that is not affected by original cluster problems.

This however has the same problems as described above, and requires an additional client instance to connect to the metrics cluster for each client instance connecting to the original cluster, which adds more complexity. Also, it is not clear how the metrics client itself is monitored.

Produce metrics data directly to a topic

Instead of using a dedicated PushTelemetryRequest, the suggestion is to use the existing ProduceRequest to send telemetry to a predefined topic on the cluster. While this has benefits in that it provides an existing method for sending compressed data to the cluster, there are more issues:

- While an existing producer instance could also produce to this topic, a consumer would need to instantiate a new producer instance for sending these metrics. In an application with many consumers, which is not uncommon, this would double the number of connections to the cluster, and increase the resource usage (memory, cpu, threads) of the application.
- A separate producer instance makes the mapping between original connection and client instance id more complex.
- What observes/collect metrics for the metrics producer instance?
- Lack of abstraction. We don't do this for any other parts of the protocol (i.e., OffsetCommitRequest could be a ProduceRequest, so could the transactional requests. FindCoordinator could be done through local hashing and metadata, etc, etc,). Makes future improvements, changes, a lot more problematic: How do we add functionality that is not covered by the produce API?
- More points of failure by introducing an additional client instance and additional connections.
- Requires metric to (at least temporarily) to be stored in a topic. Operators may want to push metrics upstream directly.

Dedicated Metrics coordinator based on client instance id

An earlier version of this proposal had the notion of metrics coordinator which was selected based on the client instance id. This metrics coordinator would be responsible for rate-limiting pushed metrics per client instance. To avoid each client to have a connection to its metric coordinator, the proposal suggested using the impersonation/request-forwarding functionality suggested in a contemporary design by Jason G.

Since this made the broker-side implementation more complex and did not provide foolproof control of misbehaving clients, (a rogue client could simply generate new instance ids on each request) it was decided that the value of this functionality was not on par with the implementation cost and thus removed from the proposal.

JMX

Scales poorly, Java-specific, commonly mistaken for a trick bike.

Combine with Dynamic client configuration

The Push interface is well suited for propagating client configuration to clients, but to keep scope down we leave that out of this proposal. It should be straightforward to extend and augment the interface described in this proposal to also support dynamic client configuration, log/event retrieval and so on at a later time.

Use native Kafka protocol framing for metrics

The Kafka protocol has flexible fields that would allow us to construct the metrics serialization with pure Kafka protocol fields, but this would duplicate the efforts of the OpenTelemetry specification where Kafka would have to maintain its own specification. On the broker side, the native Kafka protocol metrics would need to be converted to OpenTelemetry (or other) format anyway.

The OpenTelemetry serialization format is all Protobufs and should thus be generally available across all client implementation languages.