Migrating to OpenEdge Messaging with Apache Kafka from Sonic and JMS

WHITEPAPER

Introduction

OpenEdge 12.5 includes an exciting new feature, OpenEdge Messaging. This feature includes integration with a native Apache Kafka® client on Windows and Linux and a new object-oriented API that replaces the procedural-based ABL-JMS API. Kafka is a very popular messaging platform that has seen widespread adoption across businesses. Kafka is focused on throughput and scalability. While Kafka is not a direct replacement for JMS, it can support many JMS use cases, and supports a wide variety of platforms.

Kafka support in OpenEdge is currently limited to the Windows and Linux platforms. The Kafka producer is supported on every AVM client and server, allowing the production of Kafka records. The Kafka consumer is not supported on Progress Application Server (PAS) for OpenEdge or in a GUI-based event-driven client. The Kafka consumer requires messages to be polled and processed within a time frame and is well-suited for an ABL batch client. For PAS for OpenEdge, Confluent provides an HTTP Sink connector that allows you to consume Kafka messages on PAS for OpenEdge with a custom WebHandler. If you would like GUI client support or a native PAS for OpenEdge connector, please let OpenEdge Product Management know.

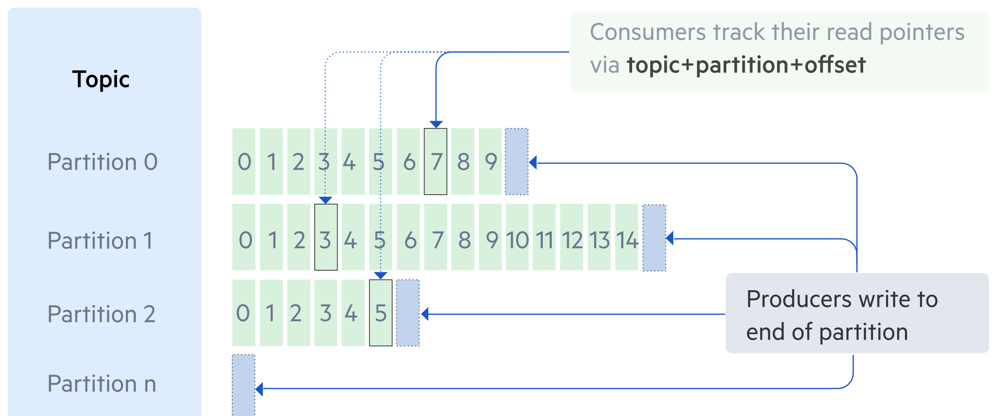
Kafka overview

Apache Kafka is a community-distributed event-streaming platform capable of handling trillions of events a day. Initially conceived as a messaging queue, Kafka is based on an abstraction of a distributed commit log. Since being created and open sourced by LinkedIn in 2011, Kafka has quickly evolved from a messaging queue to a full-fledged event-streaming platform.

Kafka provides fault tolerance and scalability through the use of Kafka clusters. It is recommended that the minimum Kafka cluster be comprised of three brokers. The number of brokers is determined by your organization's needs.

© 2022 Progress. All Rights Reserved.

Kafka persists records across the cluster, with each system consisting of a partition assigned to a topic. Each Kafka record can be identified through topic, partition, and offset. Here is a visual representation:



Records are persisted in a partition subject to policy-based time or storage settings. The default retention period is seven days and the default storage setting is 1GB. As a consumer reads and accepts records, those offsets are committed to the cluster, allowing consumers to pick up where they left off.

OpenEdge Messaging requires that each consumer is part of a named consumer group, even if the group consists of a single consumer. This allows multiple clients to process messages in a cooperative process, allowing easy scalability and recovery by just adding another client.

OpenEdge Messaging overview

OpenEdge Messaging consists of an object-oriented API based on the familiar builder pattern. The native Kafka client, librdkafka, is a shared library that does not ship with OpenEdge. To use OpenEdge Messaging, you are required to install librdkafka 1.7 or later on each client machine. Details can be found in the documentation.

Records in Kafka are treated as an array of bytes with no formal schema. Unlike JMS, Kafka does not define specific message types. OpenEdge Messaging provides serializers and deserializers for some basic types, and an interface you can use to create your own serializers and deserializers to support any record format you want.

© 2022 Progress. All Rights Reserved.

Kafka producer

Producing records is straightforward in Kafka. There are no domains, and fault tolerance is built into the Kafka client. Kafka does support the notion of batching records in a transaction, but that is currently not supported by the OpenEdge producer. Transacted records produced by other Kafka clients can be consumed by the OpenEdge consumer and is covered in this document.

Here is an example of creating a Kafka producer:

```
using OpenEdge.Messaging.ProducerBuilder.
using OpenEdge.Messaging.RecordBuilder.
using OpenEdge.Messaging.IProducer.
using OpenEdge.Messaging.IProducerRecord.
using OpenEdge.Messaging.ISendResponse.
using OpenEdge.Messaging.ProducerConfig.
using OpenEdge.Messaging.Kafka.KafkaProducerConfig.
/* *************************** Definitions ************************** */
block-level on error undo, throw.

var char theTopic = “JMSTopic”.
var char theMsg = “This is message “.
var int numRecs = 50.
var int cnt = 0.

var ProducerBuilder pb.
var RecordBuilder rb.

var IProducer theProducer.
var IProducerRecord theRecord.
var ISendResponse theResponse.

/* ******************** Preprocessor Definitions ******************** */

/* *************************** Main Block *************************** */
// This builder will create our producer
pb = ProducerBuilder:Create(“progress-kafka”).

// Kafka reqires at least one bootstrap server host and port.
pb:SetProducerOption(KafkaProducerConfig:BootstrapServers,
    “host:port,host:port”).
```


© 2022 Progress. All Rights Reserved.

```
// Logging - Very little from kafka below debug level
pb:SetProducerOption("debug", "topic, msg").
pb:SetProducerOption("log_level", "7")

// A value serializer is required.
pb:SetProducerOption(ProducerConfig:ValueSerializer,
    "OpenEdge.Messaging.StringSerializer").

// Create the producer
theProducer = pb:Build().

// Create a record to send
rb = theProducer:RecordBuilder.

// The topic name is always required
rb:SetTopicName(theTopic).

repeat while cnt < numRecs on error undo, throw:

    cnt = cnt + 1.
    // The body of the record is always required
    rb:SetBody(theMsg + string(cnt) + "-" + string(now)).
    theRecord = rb:Build().

    // send the record (message)
    theResponse = theProducer:Send(theRecord).
end.

// We threw away all SendResponse objects except the last one
// When flush returns true, SendResponse object will be updated repeat while
theProducer:Flush(100) = false.
    // allow AVM to have control instead of setting a big timeout in flush
end.

message "Success: " theResponse:Success "Partition: " theResponse:PartitionId
    "Offset: " theResponse:Offset view-as alert-box.

catch e AS Progress.Lang.MessagingError:
    message "Caught error:" e:MessagingErrorNum skip
        e:GetMessage(1).
end.
```

© 2022 Progress. All Rights Reserved.

ABL support for OpenEdge Messaging is contained in `OpenEdge.Messaging.pl` and must be added to $PROPATH along with `OpenEdge.Net.pl` and `OpenEdge.Core.pl.` When you create a `ProducerBuilder`, the Kafka native client shared library is loaded.

A set of bootstrap servers is required (replace `host:port` with actual values in the above example).

During development, you may want to enable client logging in the Kafka client. See *Use OpenEdge Messaging with Apache® Kafka®* for details.

You specify a serializer that turns your message into bytes (memptr) for Kafka. The example uses the included `StringSerializer`. An interface is provided to allow you to create your own serializers and deserializers. Once all the properties and options required are set, you create your producer.

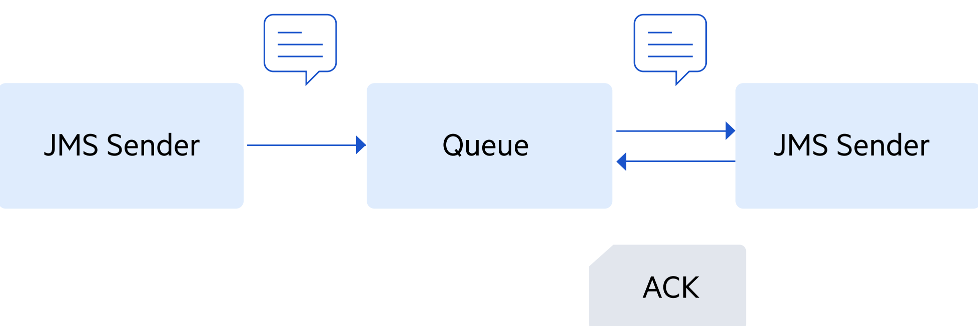
The producer provides you with a `RecordBuilder` that you use to create your record. The `RecordBuilder` requires a topic name. You can configure your Kafka cluster for automatic creation of topics, which is useful in development. However, a production Kafka cluster most likely does not allow this, instead requiring configuration by an administrator.

The above example creates 50 `String` records and sends them to the specified topic. Messages are sent asynchronously and a `SendResponse` object is returned for each record sent. In this example, we only care about the last `SendResponse` object, and produce messages as fast as we can. After generating 50 messages, we sit in a loop calling `Flush` () with a small timeout. `Flush` () returns false if it times out, so we loop until `Flush` () returns true. We can then look at the `SendResponse` object to get additional information on the last message, such as `PartitionId` and `Offset.` If you need to ensure every message was sent, saving `SendResponse` objects in a `Progress.Collections.Listis` one option. After running this code, there are now 50 records sitting in the Kafka cluster that are persisted as long as the policy permits. Unlike a JMS queue, these records are not deleted as they are read.

JMS semantics are strictly a function of the consumer.

© 2022 Progress. All Rights Reserved.

Modeling queue-based messaging



Modeling queue-based messaging

Above is a representation of how JMS queue-based messaging works. A sender creates a JMS message of a certain type and sends it to a JMS broker where it is persisted. A JMS consumer then reads the message of the queue. If it is processed, the consumer sends back an acknowledgment and that message is no longer persisted. If the consumer does not acknowledge, it remains on the broker and can be consumed at a later time. Also, once a message is acknowledged, any previous messages consumed but not acknowledged are automatically acknowledged.

An OpenEdge Messaging consumer is always part of a named consumer group, even if it is the only consumer. Consumer groups allow multiple clients to consume and process records, allowing you to scale up with ease. Consumer groups also allow you to model JMS-like behavior through the group id and the `auto.offset.reset` property.

When a consumer connects to a Kafka cluster and subscribes to a topic, Kafka starts at the last committed offset. If this is the first time a consumer group connects to a topic, Kafka needs to know where it should set that offset by the value of the `auto.offset.reset` property. The property accepts values of "earliest", `"smallest"`, `"latest"`, `"largest"`, and `"error"` which basically says to set the offset pointer to the first record available, or after the last record available. If you specify "error", an error is triggered if this consumer group has no offsets stored. For a queue-like operation, you set the property to `"earliest"` or `"smallest"`.

Another requirement for a queue-like operation is to use a well-known group id for the consumer group. Offsets are stored for consumer groups per topic. To keep it simple, setting the `group.id` property to the topic name works. Multiple consumers can then process messages cooperatively. Below is an example of queue-like functionality with OpenEdge Messaging:

© 2022 Progress. All Rights Reserved.

```
using OpenEdge.Messaging.ConsumerBuilder.
using OpenEdge.Messaging.IConsumer.
using OpenEdge.Messaging.IConsumerRecord.
using OpenEdge.Messaging.ConsumerConfig.
using OpenEdge.Messaging.Kafka.KafkaConsumerConfig.

/* *************************** Definitions ************************** */

block-level on error undo, throw.

// Set group id to topic name to only allow a record to be consumed by a single
// consumer.
var char theTopic = "JMSTopic".
var char theGroupId = theTopic.

// If no offsets commited, start at first available record
var char autoOffReset = "earliest".

var ConsumerBuilder cb.
var IConsumer theConsumer.
var IConsumerRecord theRecord.

var char clientId.
var integer msgCnt.

/* ******************** Preprocessor Definitions ******************** */

/* *************************** Main Block *************************** */ cb =
ConsumerBuilder:Create("progress-kafka").
cb:SetConsumerOption(KafkaConsumerConfig:BootstrapServers,
   "host:port,host:port").

/* *************************** Logging Settings ********************* */
cb:SetConsumerOption("debug", "consumer").
cb:SetConsumerOption("log_level", "7").

/* *************************** Auto/Manual Offset Commit ************ */
// Will automatically commit received messages
cb:SetConsumerOption("enable.auto.commit", "true").
```

© 2022 Progress. All Rights Reserved.

```
// Want all unacknowledged messages
// Only applies when no offset is stored for the consumer group
cb:SetConsumerOption("auto.offset.reset", autoOffReset).

/* **************************** Consumer Group ********************* */
// For queue type operation, set group.id to topic name
// Records will be reread if group.id is different from what first
// read the records
cb:SetConsumerOption("group.id", theGroupId).

// Kafka will randomly generate a client ID if you don't set one
clientId = "QueueConsumer".
cb:SetConsumerOption("client.id", clientId).

// A deserializer is required
cb:SetConsumerOption(ConsumerConfig:ValueDeserializer,
    "OpenEdge.Messaging.StringDeserializer").

// Consumers can subscribe to multiple topics
cb:AddSubscription(theTopic).

theConsumer = cb:Build().

msgCnt = 0.

// loop until we receive 10 records or timeouts.
repeat while msgCnt < 10:
    // request a record, waiting up to 1 second for some records to be available
    theRecord = theConsumer:Poll(1000).

    if valid-object(theRecord) then do:
        message theRecord:Body view-as alert-box.
    end.
    msgCnt = msgCnt + 1.
end.
```

© 2022 Progress. All Rights Reserved.

In this example, we subscribe to a topic and retrieve 10 messages, or timeouts, before exiting. `Enable.auto.commit` is set to true, meaning offsets are committed as messages are received. The important properties to set are group.id (set to the topic name), and `auto.offset.reset` (set to `"earliest"`). Since we have 50 records already stored in our Kafka cluster, running this code the first time sets the offset to the first record and retrieves 10 records before exiting. The second time the code is run, Kafka sees this consumer group has committed offsets, so the `auto.offset.reset` property does not come into play. This is true even if you were to change its value, as it is only applicable when a consumer group has no committed offsets for a topic.

To retrieve messages, call `Poll` () giving it a timeout. `Poll` () returns either a record, or times out and return an unknown value. Since `enable.auto.commit` is true, there is nothing else we need to do.

Modeling PubSub-based messaging

With PubSub-based messaging, a client connects to a topic on a JMS broker, and receives messages generated after connection. There is no storage of these messages on the JMS broker, so once they are sent, they are gone. To model this with OpenEdge Messaging and Kafka, you need only slight modifications to the queue-based consumer.

```
// Set group id to a unique value so all consumers of the topic track their
// offsets independently.
var char theTopic = "JMSTopic".
var char theGroupId = GUID.

// If no offsets commited, start at the end of the partition to await new records

var char autoOffReset = "latest".
```

We need to ensure each client has a unique group id so that they are in their own consumer group. The second change is setting `auto.offset.reset` to `"latest".` This ensures the client only receives new messages.

© 2022 Progress. All Rights Reserved.

One more change is required, which is unique to the OpenEdge implementation. Since we always use consumer groups, and Kafka stores committed offsets for each group, this could lead to useless data being stored in the Kafka cluster. Since the group id we are using is guaranteed to be unique, there is never a time where this stored information is required. Auto.delete.groupis a custom property added to our consumer:

```
cb:SetConsumerOption("auto.delete.group", "true").
```

When `auto.delete.group` is set to true, the consumer automatically deletes the consumer group from the Kafka cluster when it terminates. This property is specific to OpenEdge and is not part of the default Kafka behavior.

JMS also supports a durable subscription. With this model, messages are stored while the consumer is not active. This is the default behavior in Kafka. To support a durable subscription, all you need to do is take the queue-based example and set `group.id` to the subscriber name. Each subscriber has its own topics and offsets stored in the Kafka cluster.

Using JMS message types

Records in Kafka are just a bag of bytes. OpenEdge Messaging provides you with interfaces to create serializers and deserializers allowing you to support any message format you require. OpenEdge Messaging includes support for `Memptr` (JMS Bytes message), `String` (JMS Text message), and Json serializers.

Sonic's implementation extended the Text message to create an XML message. OpenEdge further extended Sonic's XML message to create `TempTable` and `DataSet` messages. As far as JMS was concerned, they were all text messages. You can create `TempTable` records in Kafka by writing your own serializers and deserializers. This is the interface for a serializer:

```
interface OpenEdge.Messaging.ISerializer:
    method public memptr Serialize(data as Object,
        serializationContext as ISerializationContext).
end.
```


© 2022 Progress. All Rights Reserved.

You are required to implement one method that takes an object and returns a memptr. A context object is included that allows you to get record headers as well as provide you with other information that may be useful. In order to support sending a `TempTable`, you wrap the table handle in a holder object. This is just a simple object with a `TableHandle` property. This is what the serializer looks like:

```
method public memptr Serialize(data as Object, ctx as ISerializationContext):
    var handle ttHdl.
    var memptr testData.
    var logical sts.

    ttHdl = cast(data, TTHolder):TableHandle.
    if not valid-handle(ttHdl) then do:
        undo, throw new MessagingError("Invalid TEMP-TABLE handle.", 0).
    end.
    sts = ttHdl:Write-XML("MEMPTR", /* target-type */
         testData,        /* target for XML */
         false,            /* formatted XML */
         ?,               /* encoding */
         ?,               /* schema-location of external schema */
         true,            /* write-schema in XML along with data */
         false) NO-ERROR.  * min-schema writes minimum schema info */

    if not sts and error-status:error then do:
        undo, throw new MessagingError("Unable to serialize TEMP-TABLE handle.", 0).
    end.

    return testData.
end.
```

A serializer needs a corresponding deserializer that implements this interface:

```
interface OpenEdge.Messaging.IDeserializer:
    method public Object Deserialize(data as memptr, ctx as IDeserializationContext).
end.
```


© 2022 Progress. All Rights Reserved.

The **`Deserialize ()`** method takes a memptr and returns an object. The object is our **`TempTable`** Holder. Here is what the deserializer looks like.

```
method public Object Deserialize(data as memptr, ctx as IDeserializationContext):
    var handle ttHdl.
    var logical sts.

    create temp-table ttHdl.
    sts = ttHdl:Read-XML("MEMPTR",     /* source-type of XML source */
         data,              /* source of XML */
         "EMPTY",          /* read-mode for temp-table */
         ?,                /* schema-location of external schema (URL) */
         false,            /* override-default-mapping of 4GL datatypes */
         ?,                /* field-type-mapping for XML to 4GL datatypes */
         =?) no-error.     /* verify-schema-mode of XML schema */

   if not sts and error-status:error then do:
       undo, throw new MessagingError("Unable to deserialize TEMP-TABLE handle.", 0).
end.

   return new TTHolder(ttHdl).
end.
```

Just like the serializer, the deserializer includes a context object that contains useful information on the record that was received.

Supporting transactions

OpenEdge Messaging currently does not support creating a transaction around a group of records. However, the consumer does allow support for transactions from non-OpenEdge producers. When you wrap a group of records in a transaction, consumers do not receive them until that transaction is committed.

© 2022 Progress. All Rights Reserved.

Conclusion

Kafka is a highly-scalable, well-supported, and widely-used platform for streaming records. While it may not support every feature found in Sonic and the OpenEdge Adapter for JMS, it does provide many other features JMS does not have. You may also find the implementation of a native client in ABL much easier to maintain due to not needing an adapter.

For a copy of the code discussed, visit the Progress Community

About Progress

Dedicated to propelling business forward in a technology-driven world, Progress (NASDAQ: PRGS) helps businesses drive faster cycles of innovation, fuel momentum and accelerate their path to success. As the trusted provider of the best products to develop, deploy and manage high-impact applications, Progress enables customers to build the applications and experiences they need, deploy where and how they want and manage it all safely and securely. Hundreds of thousands of enterprises, including 1,700 software companies and 3.5 million developers, depend on Progress to achieve their goals—with confidence. Learn more at www.progress.com

2022 Progress Software Corporation and/or its subsidiaries or affiliates. All rights reserved.
Rev 2022/04 RITM0155454

Worldwide Headquarters

Progress, 14 Oak Park,
Bedford, MA 01730 USA
Tel: +1-800-477-6473

facebook.com/progresssw
twitter.com/progresssw
youtube.com/progresssw
linkedin.com/company/progress-software
progress_sw_