KIP-382: MirrorMaker 2.0

Status

Current state: *Accepted*

Discussion thread: *here*

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

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

Further Improvements and Proposals on Mirror Maker 2

Accepted

- KIP-545: support automated consumer offset sync across clusters in MM 2.0
- KIP-597: MirrorMaker2 internal topics Formatters
- KIP-690: Add additional configuration to control MirrorMaker 2 internal topics naming convention
- KIP-710: Full support for distributed mode in dedicated MirrorMaker 2.0 clusters
- KIP-716: Allow configuring the location of the offset-syncs topic with MirrorMak
- KIP-720: Deprecate MirrorMaker v1er2

Under Discussion

- KIP-781: Improve MirrorMaker2's client configuration

Abandoned

- KIP-656: MirrorMaker2 Exactly-once Semantics (superseded by KIP-618: Exactly-Once Support for Source Connectors and KIP-710: Full support for distributed mode in dedicated MirrorMaker 2.0 clusters)
- KIP-712: Shallow Mirroring

Motivation

MirrorMaker has been used for years in large-scale production environments, but not without several problems:

- Topics are created with default configuration. Often they'll need to be repartitioned manually.
- ACL and configuration changes are not synced across mirrored clusters, making it difficult to manage multiple clusters.
- Records are repartitioned with DefaultPartitioner. Semantic partitioning may be lost.
- Any configuration change means the cluster must be bounced. This includes adding new topics to the whitelist, which may be a frequent operation.
- There's no mechanism to migrate producers or consumers between mirrored clusters.
- There's no support for exactly-once delivery. Records may be duplicated during replication.
- Clusters can't be made mirrors of each other, i.e. no support for active/active pairs.
- Rebalancing causes latency spikes, which may trigger further rebalances.

For these reasons, MirrorMaker is insufficient for many use cases, including backup, disaster recovery, and fail-over scenarios. Several other Kafka replication tools have been created to address some of these limitations, but Apache Kafka has no adequate replication strategy to date. Moreover, the lack of a native solution makes it difficult to build generic tooling for multi-cluster environments.

I propose to replace MirrorMaker with a new multi-cluster, cross-data-center replication engine based on the Connect framework, MirrorMaker 2.0 (MM2). The new engine will be fundamentally different from the legacy MirrorMaker in many ways, but will provide a drop-in replacement for existing deployments.

Highlights of the design include:

- Leverages the Kafka Connect framework and ecosystem.
- Includes both source and sink connectors.
- Includes a high-level driver that manages connectors in a dedicated cluster.
- Detects new topics, partitions.
- Automatically syncs topic configuration between clusters.
- Manages downstream topic ACL.
- Supports "active/active" cluster pairs, as well as any number of active clusters.
- Supports cross-datacenter replication, aggregation, and other complex topologies.
- Provides new metrics including end-to-end replication latency across multiple data centers/clusters.
- Emits offsets required to migrate consumers between clusters.
- Tooling for offset translation.
- MirrorMaker-compatible legacy mode.
- No rebalancing.

Public Interfaces

This KIP subsumes the small interface change proposed in KIP-416: Notify SourceTask of ACK'd offsets, metadata

New classes and interfaces include:

- MirrorSourceConnector, MirrorSinkConnector, MirrorSourceTask, MirrorSinkTask classes.
- MirrorCheckpointConnector, MirrorCheckpointTask.
- MirrorHeartbeatConnector, MirrorHeartbeatTask.
- MirrorConnectorConfig, MirrorTaskConfig classes.
- ReplicationPolicy interface. DefaultReplicationPolicy and LegacyReplicationPolicy classes.
- Heartbeat, checkpoint, offset sync topics and associated schemas.
- RemoteClusterUtils and MirrorClient classes for querying remote cluster reachability and lag, and for translating consumer offsets between clusters.
- MirrorMaker driver class with main entry point for running MM2 cluster nodes.
- MirrorMakerConfig used by MirrorMaker driver.
- HeartbeatMessageFormatter, CheckpointMessageFormatter
- ./bin/connect-mirror-maker.sh and ./config/mirror-maker.properties sample configuration.

New metrics include:

- replication-latency-ms(-avg/-min/-max): timespan between each record's timestamp and downstream ACK
- record-bytes(-avg/-min/-max): size of each record being replicated
- record-age-ms(-avg/-min/-max): age of each record when consumed
- record-count: number of records replicated
- checkpoint-latency-ms(-avg/-min/-max): timestamp between consumer group commit and downstream checkpoint ACK

The mbean name for these metrics will be: `kafka.mirror.connect:type=MirrorSourceConnect,target=([.\w]+),topic=([.\w]+),partition=([.\d]+)` and `kafka.mirror.connect:type=MirrorCheckpointConnector,target=([.\w]+),source=([.\w]+),group=([.\w]+)`

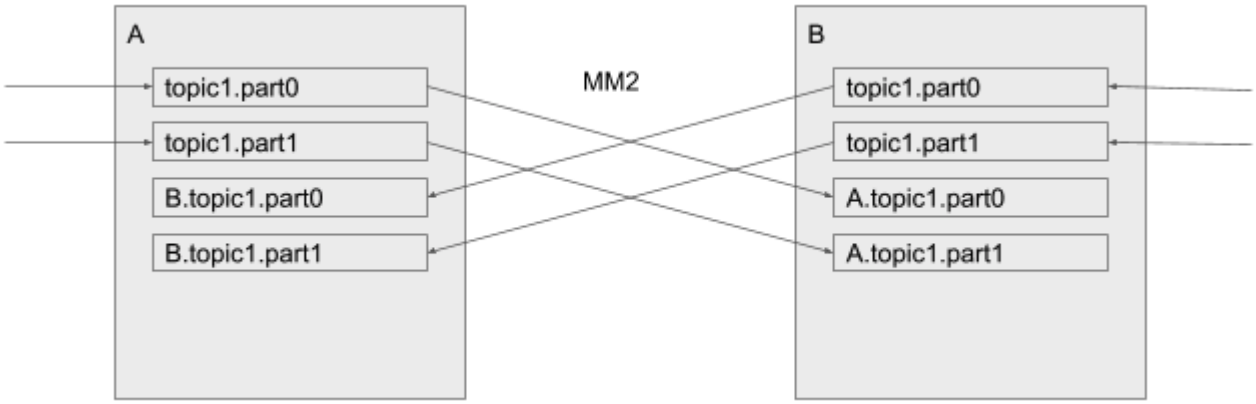
Proposed Changes

Remote Topics, Partitions

This design introduces the concept of **remote topics**, which are replicated topics referencing a source cluster via a naming convention, e.g. "us-west.topic1", where topic1 is a **source topic** and us-west is a **source cluster alias**. Any partitions in a remote topic are **remote partitions** and refer to the same partitions in the source topic. Specifically:

- Partitioning and order of records is preserved between source and remote topics.
- Remote topics must have the same number of partitions as their source topics.
- A remote topic has a single source topic.
- A remote partition has a single source partition.
- Remote partition *i* is a replica of source partition *i*.

MM2 replicates topics from a source cluster to corresponding remote topics in the destination cluster. In this way, replication never results in merged or out-of-order partitions. Moreover, the topic renaming policy enables "active/active" replication by default:



"active/active" replication

This is not supported by legacy MirrorMaker out-of-the-box (without custom handlers) -- records would be replicated back and forth indefinitely, and the topics in either cluster would be merged inconsistently between clusters. The remote naming convention avoids these problems by keeping local and remote records in separate partitions. In either cluster above, the normal topic1 contains only locally produced records. Likewise, B.topic1 contains only records produced in cluster B.

To consume local records only, a consumer subscribes to topic1 as normal. To consume records that have been replicated from a remote cluster, consumers subscribe to a remote topic, e.g. B.topic1.

This convention extends to any number of clusters.

Aggregation

Downstream consumers can aggregate across multiple clusters by subscribing to the local topic and all corresponding remote topics, e.g. topic1, us-west.topic1, us-east.topic1… or by using a regex subscription.

Topics are never merged or repartitioned within the connector. Instead, merging is left to the consumer. This is in contrast to MirrorMaker, which is often used to merge topics from multiple clusters into a single aggregate cluster.

Likewise, a consumer can always elect not to aggregate by simply subscribing to the local topic only, e.g. topic1.

This approach eliminates the need for special purpose aggregation clusters without losing any power or flexibility.

Cycle detection

It is possible to configure two clusters to replicate each other ("active/active"), in which case all records produced to either cluster can be seen by consumers in both clusters. To prevent infinite recursion, topics that already contain "us-west" in the prefix won't be replicated to the us-west cluster.

This rule applies across all topics regardless of topology. A cluster can be replicated to many downstream clusters, which themselves can be replicated, yielding topics like us-west.us-east.topic1 and so on. The same cluster alias will not appear twice in a topic name due to cycle detection.

In this way, any topology of clusters is supported, not just DAGs.

Config, ACL Sync

MM2 monitors source topics and propagates configuration changes to remote topics. Any missing partitions will be automatically created.

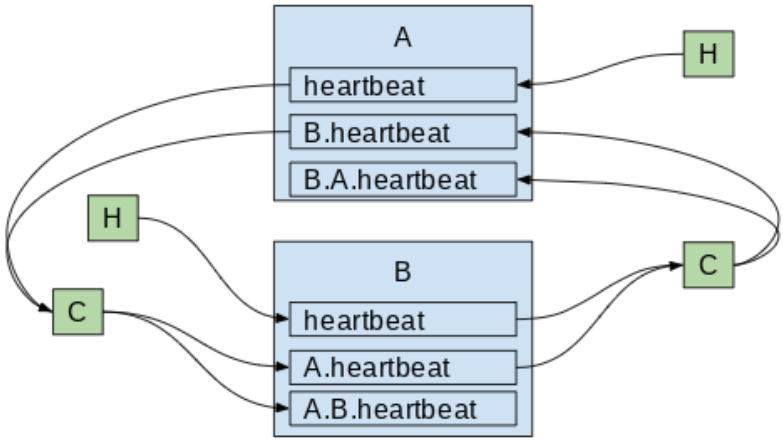
Remote topics should never be written to, except by the source or sink connector. Producers must only write to local topics (e.g. topic1), which may then be replicated elsewhere. To enforce this policy, MM2 propagates ACL policies to downstream topics using the following rules:

- Principals that can READ the source topic can READ the remote topic.

- No one can WRITE to the remote topic except MM2.

Internal Topics

MM2 emits a **heartbeat topic** in each source cluster, which is replicated to demonstrate connectivity through the connectors. Downstream consumers can use this topic to verify that a) the connector is running and b) the corresponding source cluster is available. Heartbeats will get propagated by source and sink connectors s.t. chains like backup.us-west.us-east.heartbeat are possible. Cycle detection prevents infinite recursion.



heartbeats are replicated between clusters

The schema for the heartbeat topic contains:

- target cluster (String): cluster where the heartbeat was emitted
- source cluster (String): source cluster of connector doing the emitting
- timestamp (long): epoch millis when heartbeat was created

Additionally, the connectors periodically emit **checkpoints** in the destination cluster, containing offsets for each consumer group in the source cluster. The connector will periodically query the source cluster for all committed offsets from all consumer groups, filter for those topics being replicated, and emit a message to a topic like us-west.checkpoints.internal in the destination cluster. The message will include the following fields:

- consumer group id (String)
- topic (String) – includes source cluster prefix
- partition (int)
- upstream offset (int): latest committed offset in source cluster
- downstream offset (int): latest committed offset translated to target cluster
- metadata (String)
- timestamp

As with __consumer_offsets, the checkpoint topic is log-compacted to reflect only the latest offsets across consumer groups, based on a topic-partition-group composite key. The topic will be created automatically by the connector if it doesn't exist.

Finally, an **offset sync** topic encodes cluster-to-cluster offset mappings for each topic-partition being replicated.

- topic (String): remote topic name
- partition (int)
- upstream offset (int): an offset in the source cluster
- downstream offset (int): an equivalent offset in the target cluster

The heartbeat, checkpoint, and offset sync records are encoded using Kafka's internal serdes, as with __consumer_offsets. Relevant classes below.

```
public class Checkpoint {
    public static final String TOPIC_KEY = "topic";
    public static final String PARTITION_KEY = "partition";
    public static final String CONSUMER_GROUP_ID_KEY = "group";
    public static final String UPSTREAM_OFFSET_KEY = "upstreamOffset";
    public static final String DOWNSTREAM_OFFSET_KEY = "offset";
    public static final String METADATA_KEY = "metadata";

--->%---

    public Checkpoint(String consumerGroupId, TopicPartition topicPartition, long upstreamOffset,
            long downstreamOffset, String metadata) ...

    public String consumerGroupId() ...

    public TopicPartition topicPartition() ...

    public long upstreamOffset() ...

    public long downstreamOffset() ...

    public String metadata() ...

    public OffsetAndMetadata offsetAndMetadata() ...

--->%---

public class Heartbeat {
    public static final String SOURCE_CLUSTER_ALIAS_KEY = "sourceClusterAlias";
    public static final String TARGET_CLUSTER_ALIAS_KEY = "targetClusterAlias";
    public static final String TIMESTAMP_KEY = "timestamp";

--->%---

    public Heartbeat(String sourceClusterAlias, String targetClusterAlias, long timestamp) ...

    public String sourceClusterAlias() ...

    public String targetClusterAlias() ...

    public long timestamp() ...

--->%---

public class OffsetSync {
    public static final String TOPIC_KEY = "topic";
    public static final String PARTITION_KEY = "partition";
    public static final String UPSTREAM_OFFSET_KEY = "upstreamOffset";
    public static final String DOWNSTREAM_OFFSET_KEY = "offset";

--->%---

    public OffsetSync(TopicPartition topicPartition, long upstreamOffset, long downstreamOffset) ...

    public TopicPartition topicPartition() ...

    public long upstreamOffset() ...

    public long downstreamOffset() ...
```

Remote Cluster Utils

A utility class **RemoteClusterUtils** will leverage the internal topics described above to assist in computing reachability, inter-cluster lag, and offset translation. It won't be possible to directly translate any given offset, since not all offsets will be captured in the checkpoint stream. But for a given consumer group, it will be possible to find high water marks that consumers can seek() to. This is useful for inter-cluster consumer migration, failover, etc.

The interface is as follows:

```
// Calculates the number of hops between a client and an upstream Kafka cluster based on replication heartbeats.
// If the given cluster is not reachable, returns -1.
int replicationHops(Map<String, Object> properties, String upstreamClusterAlias)

// Find all heartbeat topics, e.g. A.B.heartbeat, visible from the client.
Set<String> heartbeatTopics(Map<String, Object> properties)

// Find all checkpoint topics, e.g. A.checkpoint.internal, visible from the client.
Set<String> checkpointTopics(Map<String, Object> properties)

// Find all upstream clusters reachable from the client
Set<String> upstreamClusters(Map<String, Object> properties)

// Find the local offsets corresponding to the latest checkpoint from a specific upstream consumer group.
Map<TopicPartition, OffsetAndMetadata> translateOffsets(Map<String, Object> properties,
            String targetClusterAlias, String consumerGroupId, Duration timeout)
```

The utility class assumes DefaultReplicationPolicy is used for replication.

MirrorClient

The RemoteClusterUtils class wraps a low-level MirrorClient, with the following exposed methods:

```
MirrorClient(Map<String, Object> props) ...

void close() ...

ReplicationPolicy replicationPolicy() ...

int replicationHops(String upstreamClusterAlias) ...

Set<String> heartbeatTopics() ...

Set<String> checkpointTopics() ...

// Finds upstream clusters, which may be multiple hops away, based on incoming heartbeats.
Set<String> upstreamClusters() ...

// Find all remote topics on the cluster
Set<String> remoteTopics() ...

// Find all remote topics from the given source
Set<String> remoteTopics(String source) ...

// Find the local offsets corresponding to the latest checkpoint from a specific upstream consumer group.
Map<TopicPartition, OffsetAndMetadata> remoteConsumerOffsets(String consumerGroupId,
            String remoteClusterAlias, Duration timeout) ...
```

Replication Policies and Filters

A **ReplicationPolicy** defines what a "remote topic" is and how to interpret it. This should generally be consistent across an organization. The interface is as follows:

```
/** Defines which topics are "remote topics", e.g. "us-west.topic1". */
public interface ReplicationPolicy {

    /** How to rename remote topics; generally should be like us-west.topic1. */
    String formatRemoteTopic(String sourceClusterAlias, String topic);

    /** Source cluster alias of given remote topic, e.g. "us-west" for "us-west.topic1".
        Returns null if not a remote topic.
    */
    String topicSource(String topic);

    /** Name of topic on the source cluster, e.g. "topic1" for "us-west.topic1".
        Topics may be replicated multiple hops, so the immediately upstream topic
        may itself be a remote topic.
        Returns null if not a remote topic.
    */
    String upstreamTopic(String topic);

    /** The name of the original source-topic, which may have been replicated multiple hops.
        Returns the topic if it is not a remote topic.
    */
    String originalTopic(String topic);

    /** Internal topics are never replicated. */
    boolean isInternalTopic(String topic);
}
```

The DefaultReplicationPolicy specifies the <source>.<topic> convention described throughout this document. LegacyReplicationPolicy causes MM2 to mimic legacy MirrorMaker behavior.

Additionally, several Filters control the behavior of MM2 at a high level, including which topics to replicate. The DefaultTopicFilter, DefaultGroupFilter, and DefaultConfigPropertyFilter classes honor whitelists and blacklists via regex patterns.

```
/** Defines which topics should be replicated. */
public interface TopicFilter {

    boolean shouldReplicateTopic(String topic);
}

/** Defines which consumer groups should be replicated. */
public interface GroupFilter {

    boolean shouldReplicateGroup(String group);
}

/** Defines which topic configuration properties should be replicated. */
public interface ConfigPropertyFilter {

    boolean shouldReplicateConfigProperty(String prop);
}
```

Connectors

Both source and sink connectors are provided to enable complex flows between multiple Kafka clusters and across data centers via existing Kafka Connect clusters. Generally, each data center has a single Connect cluster and a **primary** Kafka cluster, K.

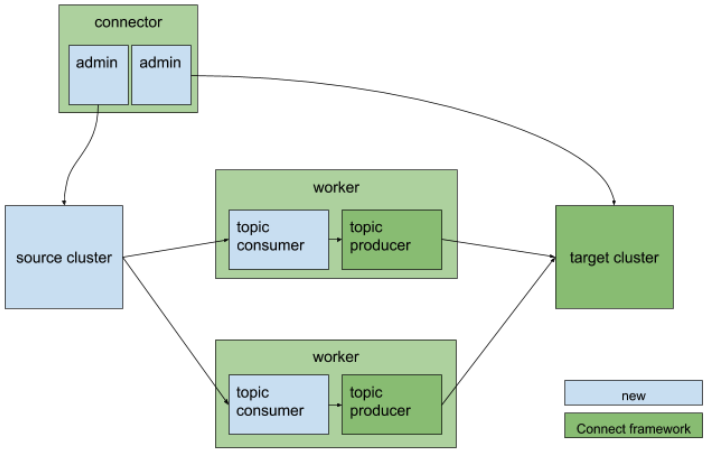
MirrorSourceConnector workers replicate a set of topics from a single source cluster into the primary cluster.

MirrorSinkConnector workers consume from the primary cluster and replicate topics to a single sink cluster.

MirrorCheckpointConnector emits consumer offset checkpoints.

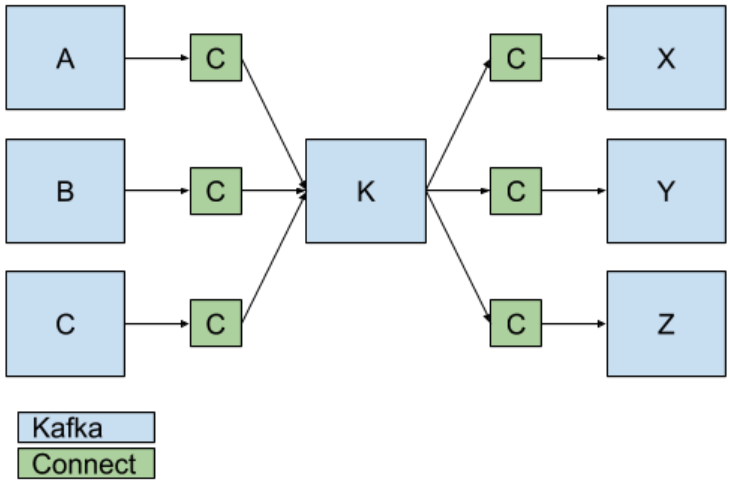
MirrorHeartbeatConnector emits heartbeats.

The source and sink connectors contain a pair of producers and consumers to replicate records, and a pair of AdminClients to propagate configuration /ACL changes.



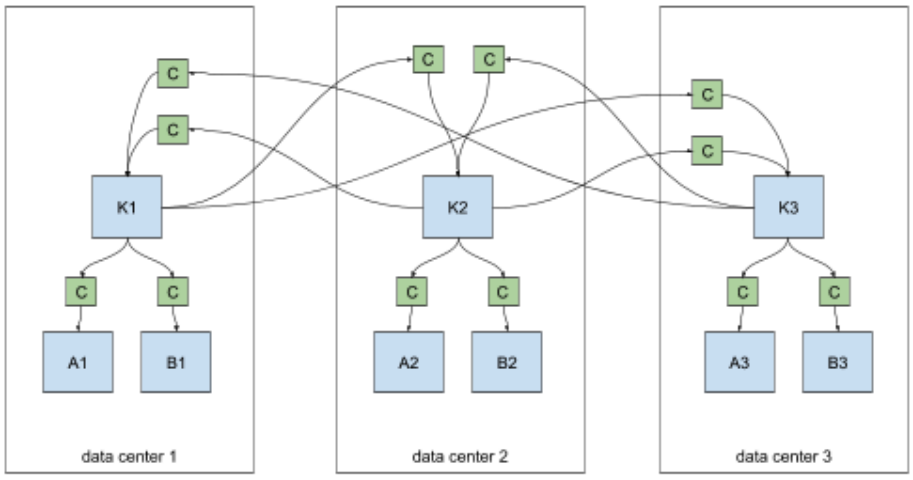
MirrorSourceConnector and workers

By combining SourceConnectors and SinkConnectors, a single Connect cluster can manage replication across multiple Kafka clusters.



Connectors (C) replicate data between Kafka clusters.

For cross-datacenter replication (XDCR), each datacenter should have a single Connect cluster which pulls records from the other data centers via source connectors. Replication may fan-out within each datacenter via sink connectors.



cross-datacenter replication of multiple clusters

Connectors can be configured to replicate specific topics via a whitelist or regex. For example, clusters A1 and B1 above might contain only a subset of topics from K1, K2, or K3.

Connector Configuration Properties

Properties common to the SourceConnector(s) and SinkConnector:

property	default value	description
name	required	name of the connector, e.g. "us-west->us-east"
topics	empty string	regex of topics to replicate, e.g. "topic1|topic2|topic3". Comma-separated lists are also supported.
topics.blacklist	".*\.internal, .*\.replica, __consumer_offsets" or similar	topics to exclude from replication
groups	empty string	regex of groups to replicate, e.g. ".*"
groups.blacklist	empty string	groups to exclude from replication
source.cluster.alias	required	name of the cluster being replicated
target.cluster.alias	required	name of the downstream Kafka cluster
source.cluster.bootstrap.servers	required	upstream cluster to replicate
target.cluster.bootstrap.servers	required	downstream cluster
sync.topic.configs.enabled	true	whether or not to monitor source cluster for configuration changes
sync.topic.acls.enabled	true	whether to monitor source cluster ACLs for changes
emit.heartbeats.enabled	true	connector should periodically emit heartbeats
emit.heartbeats.interval.seconds	5 (seconds)	frequency of heartbeats
emit.checkpoints.enabled	true	connector should periodically emit consumer offset information
emit.checkpoints.interval.seconds	5 (seconds)	frequency of checkpoints
refresh.topics.enabled	true	connector should periodically check for new topics
refresh.topics.interval.seconds	5 (seconds)	frequency to check source cluster for new topics
refresh.groups.enabled	true	connector should periodically check for new consumer groups
refresh.groups.interval.seconds	5 (seconds)	frequency to check source cluster for new consumer groups
readahead.queue.capacity	500 (records)	number of records to let consumer get ahead of producer
replication.policy.class	org.apache.kafka.connect.mirror.DefaultReplicationPolicy	use LegacyReplicationPolicy to mimic legacy MirrorMaker
heartbeats.topic.retention.ms	1 day	used when creating heartbeat topics for the first time
checkpoints.topic.retention.ms	1 day	used when creating checkpoint topics for the first time
offset.syncs.topic.retention.ms	max long	used when creating offset sync topic for the first time
replication.factor	2	used when creating remote topics

In addition, internal client properties can be overridden as follows:

property	description

source.cluster.consumer.*	overrides for the source-cluster consumer
source.cluster.producer.*	overrides for the source-cluster producer
source.cluster.admin.*	overrides for the source-cluster admin
target.cluster.consumer.*	overrides for the target-cluster consumer
target.cluster.producer.*	overrides for the target-cluster producer
target.cluster.admin.*	overrides for the target-cluster admin

These properties are specified in the MirrorConnectorConfig class, which is shared by all MirrorMaker connectors (MirrorSourceConnector, MirrorCheckpointConnector, etc). Some properties are relevant to specific Connectors, e.g. "emit.heartbeats" is only relevant to the MirrorHeartbeatConnector.

Example Configuration

A sample configuration file ./config/connect-mirror-source.properties is provided:

```
name = local-mirror-source
topics = .*
connector.class = org.apache.kafka.connect.mirror.MirrorSourceConnector
tasks.max = 1


# for demo, source and target clusters are the same
source.cluster.alias = upstream
source.cluster.bootstrap.servers = localhost:9092
target.cluster.bootstrap.servers = localhost:9092


# use ByteArrayConverter to ensure that records are not re-encoded
key.converter = org.apache.kafka.connect.converters.ByteArrayConverter
value.converter = org.apache.kafka.connect.converters.ByteArrayConverter
```

This enables running standalone MM2 connectors as follows:

```
$ ./bin/connect-standalone.sh ./config/connect-standalone.properties ./config/connect-mirror-source.properties
```

Legacy MirrorMaker Configuration

By default, MM2 enables several features and best-practices that are not supported by legacy MirrorMaker; however, these can all be turned off. The following configuration demonstrates running MM2 in "legacy mode":

```
--%<----


# emulate legacy mirror-maker
replication.policy.class = org.apache.kafka.mirror.LegacyReplicationPolicy


# disable all new features
refresh.topics.enabled = false
refresh.groups.enabled = false
emit.checkpoints.enables = false
emit.heartbeats.enabled = false
sync.topic.configs.enabled = false
sync.topic.acls.enabled = false


--%<----
```

MirrorMaker Clusters

The MirrorMaker.java driver class and ./bin/connect-mirror-maker.sh script implement a distributed MM2 cluster which does not depend on an existing Connect cluster. Instead, MM2 cluster nodes manage Connect workers internally based on a high-level configuration file. The configuration file is needed to identify each Kafka cluster. A sample MirrorMakerConfig properties file will be provided in ./config/mirror-maker.properties:

```
clusters = primary, backup
cluster.primary.bootstrap.servers = localhost:9091
cluster.backup.bootstrap.servers = localhost:9092
```

This can be run as follows:

```
$ ./bin/connect-mirror-maker.sh ./config/mirror-maker.properties
```

Internally, the MirrorMaker driver sets up MirrorSourceConnectors, MirrorCheckpointConnectors, etc between each pair of clusters, based on the provided configuration file. For example, the above configuration results in two MirrorSourceConnectors: one replicating from primarybackup and one from backupprimary. The configuration for the primarybackup connector is automatically populated as follows:

```
name = MirrorSourceConnector
connector.class = org.apache.kafka.connect.mirror.MirrorSourceConnector
source.cluster.alias = primary
target.cluster.alias = backup
source.cluster.bootstrap.servers = localhost:9091
target.cluster.bootstrap.servers = localhost:9092
key.converter.class = org.apache.kafka.connect.converters.ByteArrayConverter
value.converter.class = org.apache.kafka.connect.converters.ByteArrayConverter
```

The MirrorMaker properties file can specify static configuration properties for each connector:

```
clusters = primary, backup
primary.bootstrap.servers = localhost:9091
backup.bootstrap.servers = localhost:9092
primary->backup.topics = .*
primary->backup.emit.heartbeats.enabled = false
```

In this case, two MirrorSourceConnectors, two MirrorCheckpointConnectors, and two MirrorHeartbeatConnectors will be created by the driver and the primarybackup connectors will start replicating all source topics at launch, with heartbeats disabled. Sub-configurations like primary->backup.x.y.z will be applied to all connectors in that sourcetarget flow (MirrorSourceConnector, MirrorCheckpointConnector), avoiding the need to configure each connector individually.

The MirrorMaker driver class is similar to the existing ConnectDistributed driver, except it runs multiple DistributedHerders, one for each Kafka cluster. The herders coordinate via their respective Kafka clusters – otherwise they are completely independent from each other (i.e. are separate logical Connect clusters). This driver makes it possible to bring up a "MirrorMaker cluster" with replication flows between multiple Kafka clusters without configuring each Connector, Worker, and Herder manually.

MirrorMaker Configuration Properties

The high-level configuration file required by the MirrorMaker driver supports the following properties:

property	default value	description
clusters	required	comma-separated list of Kafka cluster "aliases"
cluster.bootstrap.servers	required	connection information for the specific cluster
cluster.x.y.z	n/a	passed to workers for a specific cluster
source->**target**.x.y.z	n/a	passed to a specific connector

Walkthrough: Running MirrorMaker 2.0

There are four ways to run MM2:

- As a dedicated MirrorMaker cluster.
- As a Connector in a distributed Connect cluster.
- As a standalone Connect worker.
- In legacy mode using existing MirrorMaker scripts.

Running a dedicated MirrorMaker cluster

In this mode, MirrorMaker does not require an existing Connect cluster. Instead, a high-level driver manages a collection of Connect workers.

First, specify Kafka cluster information in a configuration file:

```
# mm2.properties
clusters = us-west, us-east
us-west.bootstrap.servers = host1:9092
us-east.bootstrap.servers = host2:9092
```

Optionally, you can override default MirrorMaker properties:

```
topics = .*
groups = .*
emit.checkpoints.interval.seconds = 10
```

You can also override default properties for specific clusters or connectors:

```
us-west.offset.storage.topic = mm2-offsets
us-west->us-east.emit.heartbeats.enabled = false
```

Second, launch one or more MirrorMaker cluster nodes:

```
$ ./bin/connect-mirror-maker.sh mm2.properties
```

Running a standalone MirrorMaker connector

In this mode, a single Connect worker runs MirrorSourceConnector. This does not support multinode clusters, but is useful for small workloads or for testing.

First, create a "worker" configuration file:

```
# worker.properties
bootstrap.servers = host2:9092
```

An example is provided in ./config/connect-standalone.properties.

Second, create a "connector" configuration file:

```
# connector.properties
name = local-mirror-source
connector.class = org.apache.kafka.connect.mirror.MirrorSourceConnector
tasks.max = 1
topics=.*

source.cluster.alias = upstream
source.cluster.bootstrap.servers = host1:9092
target.cluster.bootstrap.servers = host2:9092

# use ByteArrayConverter to ensure that in and out records are exactly the same
key.converter = org.apache.kafka.connect.converters.ByteArrayConverter
value.converter = org.apache.kafka.connect.converters.ByteArrayConverter
```

Finally, launch a single Connect worker:

```
$ ./bin/connecti-standalone.sh worker.properties connector.properties
```

Running MirrorMaker in a Connect cluster

If you already have a Connect cluster, you can configure it to run MirrorMaker connectors.

There are four such connectors:

- MirrorSourceConnector

- MirrorSinkConnector (coming soon)
- MirrorCheckpointConnector
- MirrorHeartbeatConnector

Configure these using the Connect REST API:

```
PUT /connectors/us-west-source/config HTTP/1.1

{
    "name": "us-west-source",
    "connector.class": "org.apache.kafka.connect.mirror.MirrorSourceConnector",
    "source.cluster.alias": "us-west",
    "target.cluster.alias": "us-east",
    "source.cluster.bootstrap.servers": "us-west-host1:9091",
    "topics": ".*"
}
```

Running MirrorMaker in legacy mode

After legacy MirrorMaker is deprecated, the existing ./bin/kafka-mirror-maker.sh scripts will be updated to run MM2 in legacy mode:

```
$ ./bin/kafka-mirror-maker.sh --consumer consumer.properties --producer producer.properties
```

Compatibility, Deprecation, and Migration Plan

A new version of ./bin/kafka-mirror-maker.sh will be implemented to run MM2 in "legacy mode", i.e. with new features disabled and supporting existing options and configuration properties. Existing deployments will be able to upgrade to MM2 seamlessly.

The existing MirrorMaker source will be removed from Kafka core project. MM2 will be added to the connect project under a new module "mirror" and package "org.apache.kafka.connect.mirror".

Deprecation will occur in three phases:

- Phase 1 (targeting next Apache Kafka release): All MirrorMaker 2.0 Java code is added to ./connect/mirror/.
- Phase 2 (subsequent release): Legacy MirrorMaker Scala code is deprecated, but kept in place. Sample MM2 scripts and configuration files are added to ./bin/ and ./config/.
- Phase 3 (subsequent release): Legacy MirrorMaker Scala code is removed from Apache Kafka. A new ./bin/kafka-mirror-maker.sh script is provided which replaces and emulates the legacy script.

Future Work

- Command-line tools wrapping RemoteClusterUtils. Document cross-cluster migration procedure.
- Broker and KafkaConsumer support for unambiguous remote topic names, e.g. with a reserved separator character.
- CheckpointSinkConnector/Task to support upstream checkpointing when a single Connect cluster is used.
- Additional metrics.
- Support reconfiguration of running Tasks without stop, start.
- Exactly-once semantics in Connect and MM2.
- Prevent duplicate streams from "diamond problem" (fan-out-fan-in).
- Support multiple clusters/workers/herders in ConnectDistributed to obviate MM2-specific driver.

Rejected Alternatives

- We could release this as an independent project, but we feel that cluster replication should be one of Kafka's fundamental features.
- We could deprecate MirrorMaker but keep it around. However, we see this as a natural evolution of MirrorMaker, not an alternative solution.
- We could update MirrorMaker rather than completely rewrite it. However, we'd end up recreating many features provided by the Connect framework, including the REST API, configuration, metrics, worker coordination, etc.
- We could build on Uber's uReplicator, which solves some of the problems with MirrorMaker. However, uReplicator uses Apache Helix to provide features that overlap with Connect, e.g. REST API, live configuration changes, cluster management, coordination etc. A native MirrorMaker should only use native parts of Apache Kafka. That said, uReplicator is a major inspiration for the MM2 design.
- We could provide a high-level REST API for remote-controlling a MirrorMaker cluster. However, this has overlapping concerns with Connect.
- Instead of separate connectors for heartbeats and checkpoints, we could do everything in a single connector. However, this restricts how Converters and Transformations can be used. For example, you might want a replication pipeline that converts to JSON, but that doesn't necessarily mean you also want heartbeats to be JSON-encoded. It's possible to create a separate KafkaProducer within MirrorSourceConnector to emit Heartbeats and Checkpoints without going through the Connect transformation pipeline, but I don't want to introduce the additional configuration properties for a whole new producer.