KIP-291: Separating controller connections and requests from the data plane

Status

Current state: Accepted

Discussion thread: *here*

Vote thread: *here*

JIRA: KAFKA-4453

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

Terminology

- **listeners:** The endpoints that a broker uses to bind server sockets and listens for incoming connections.
- **advertised-listeners:** The endpoints that a broker publishes to Zookeeper, which are used by other brokers or clients to establish connections with the publisher broker. Its value may be the same as the **listeners,** but can also be overwritten to be different.
- **inter-broker-listener-name:** The listener name, e.g. INTERNAL, used for inter broker connections. Its value is derived from either the "inter.broker.listener.name" config or the "security.inter.broker.protocol" config.

Motivation

Today there is no separate between controller requests and regular data plane requests. Specifically (1) a controller in a cluster uses the same advertised endpoints to connect to brokers as what clients and regular brokers use (2) on the broker side, the same network (processor) thread could be multiplexed by handling a controller connection and many other data plane connections (3) after a controller request is read from the socket by a network thread, it is enqueued into the single FIFO requestQueue, which is used for all types of requests (4) request handler threads poll requests from the requestQueue and handles the controller requests with the same priority as regular data requests.

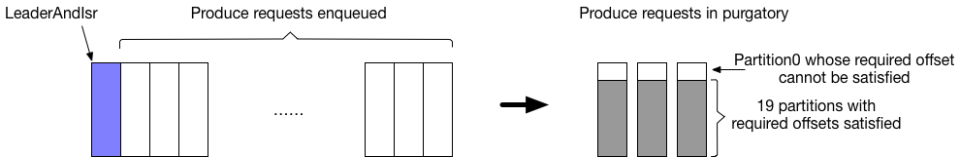
Because of the multiplexing at every stage of request handling, controller requests could be significantly delayed under the following scenarios:

1. The requestQueue is full, and therefore blocks a network (processor) thread that has a controller request ready to be enqueued.
2. A controller request is enqueued into the requestQueue after a backlog of data requests, and experiences a long queuing time in the requestQueue.

Delaying a controller request for a prolonged period can have serious consequences, and we'll examine the impact of the delayed processing for a LeaderAndISR request and a UpdateMetadata request now[1].

1. Delayed processing of a LeaderAndISR request
   a. LeaderAndISR with partitions to be transitioned to *followers***:** Consider the case that a LeaderAndISR request is enqueued after a backlog of Produce requests; the LeaderAndISR request contains a partition that needs to be transitioned from a leader to a follower, say partition0; the Produce requests ahead of it all have records for partition0, and let's assume each of the produce requests has a total of 20 partitions, partition0, partition1,..., partition19. Further we assume that the previous followers fetching from this broker for partition0 have *stopped* fetching.
      i. If the produce requests have required acks = -1 (all), they would be parked in the purgatory after their records are appended to the local log, waiting for followers to meet the required offsets for all of the 20 partitions. Unfortunately because the previous followers have stopped fetching for partition0, the required offset for partition0 can never be satisfied. Even after the remaining 19 partitions satisfy their required offsets, the produce requests will still be pending in the purgatory until the LeaderAndISR

request is processed to mark partition0 as no longer being the leader, or the produce request time out is triggered. The impact for users is increased latency for the produce requests ahead of the LeaderAndISR.



ii. If the produce requests have acks = 0 or acks = 1, their records will be appended to the local log, and a response will be sent to the client immediately. In this case, the produce requests do not have prolonged latency. However since the appended records will not be replicated to other followers, after processing of the LeaderAndISR that makes the broker a follower, those records will be truncated. In contrast, if we can change the behavior and process the LeaderAndISR request immediately, an error code corresponding to NotLeaderForPartition will be returned to the clients, causing the clients to retry and avoid the data loss. Even though losing data for the acks = 0 and acks = 1 produce requests is allowed in Kafka, it'll be better if we can minimize the data loss.

b. LeaderAndISR with partitions to be transitioned to *leaders:* Again let's consider the case that a LeaderAndISR request is delayed because of a backlog of Produce requests ahead of it, and the LeaderAndISR contains a partition that needs to be transitioned from a follower to a leader, say *partition0*. Further let's assume the previous leader for partition0 have resigned its leadership. In this case, before the LeaderAndISR is processed, the partition is effectively unavailable, both for producing and consuming. If the LeaderAndISR request is processed immediately, we can greatly shorten the unavailability interval.

2. Delayed processing of an UpdateMetadata request. Delayed processing of an UpdateMetadata request means clients may receive stale metadata. For example, the stale metadata may have the wrong leadership info for certain partitions, causing the client not being to produce or consume until the correct metadata with up-to-date leadership is received. It will be much better if the UpdateMetadataRequest can be processed immediately after arriving at a broker.

In summary, we'd like to mitigate the effect of stale metadata by shortening the latency between a controller request's arrival and processing on a given broker.

Public Interfaces

- We plan to add the following new metrics :
  1. kafka.network:name=ControlPlaneRequestQueueSize,type=RequestChannel
     kafka.network:name=ControlPlaneResponseQueueSize,type=RequestChannel
     to show the size of the new control plane, request and response queues.
  2. kafka.network:name=ControlPlaneNetworkProcessorAvgIdlePercent,type=SocketServer
     kafka.server:name=ControlPlaneRequestHandlerAvgIdlePercent,type=KafkaRequestHandlerPool
     with the former monitoring the idle percentage of pinned control plane network thread, and the latter monitoring the idle percentage of pinned control plane request handler thread. (Pinned control plane threads are explained in details below.)
  3. kafka.network:name=ControlPlaneExpiredConnectionsKilledCount,type=SocketServer
     to show the number of expired connections that were disconnected on the control plane.
- The meaning of the existing metric
  `kafka.network:name=RequestQueueSize,type=RequestChannel`

  will be changed, and it will be used to show the size of the data request queue only, which does not include controller requests.
- The meaning of the two existing metrics
  kafka.network:name=NetworkProcessorAvgIdlePercent,type=SocketServer
  kafka.server:name=RequestHandlerAvgIdlePercent,type=KafkaRequestHandlerPool
  will be changed slightly in that they now only cover the threads for data plane threads, not including the pinned threads for controller requests.
- We plan to add a new config for brokers to specify a dedicated listener for controller connections: the "control.plane.listener.name" config. When not set explicitly, the config will have a default value of "null" and indicate that the proposed change in this KIP will not take effect. A detailed explanation of the config is shown in the "proposed changes" section.
- A new listener-to-endpoint entry dedicated to controller connections need to be added to the "listeners" config and the "advertised.listeners" config.
  Also a new entry needs to be added to the "listener.security.protocol.map" to specify the security protocol of the new endpoint.

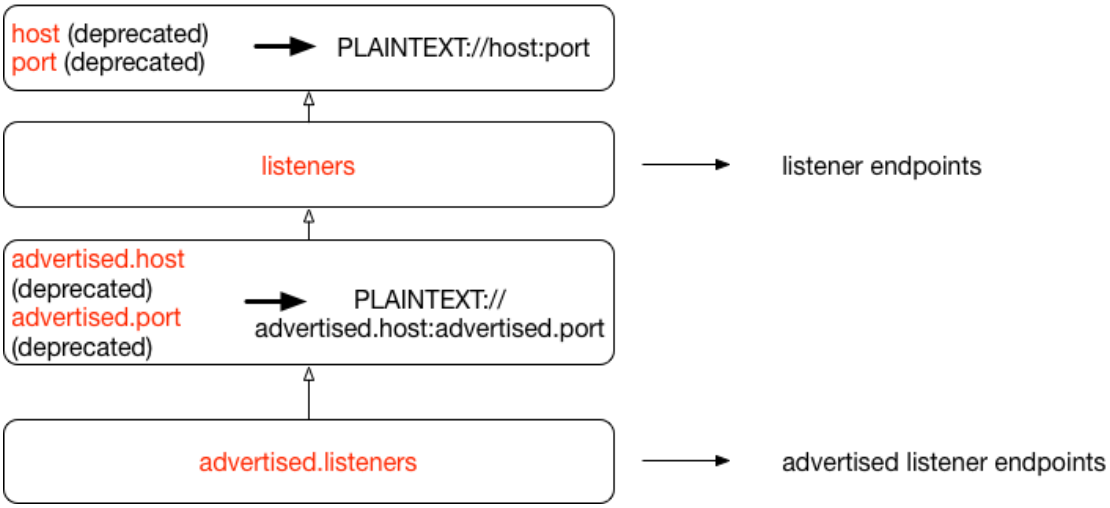
Proposed Changes

In order to eliminate queuing for controller requests, we plan to add a dedicated endpoint on a brokers for controller connections, as well as two dedicated control plane threads for handling controller requests. To explain the proposed change, we first go through how brokers should get the dedicated endpoints through configs, and expose the endpoints to Zookeeper. Then we discuss how a controller can learn about the dedicated endpoints exposed by brokers. Finally we describe how controller requests are handled over the dedicated connections.

How does a broker get the dedicated endpoints through configs, and expose the endpoints to Zookeeper?

How does it work today?

Upon startup, a broker needs to get two list of endpoints: the **listeners** endpoints that are used to bind the server socket and accept incoming connections, as well as an **advertised listeners** endpoints list that are published to Zookeeper for clients or other brokers to establish connections with. More details on the reason of separating these two lists can be found at KAFKA-1092 and KIP-103. In terms of how the values for the two lists are derived, we find it intuitive to understand the relationships of different configs using the following chart:



- Broker configs are marked in red, e.g. listeners, advertised.host.
- To calculate the **listeners** endpoints, the "listeners" config value will be used directly if it's set. Otherwise, the result will be a single endpoint, constructed using the listener name PLAINTEXT, the "host" config, and the "port" config.
- To calculate the **advertised-listeners** endpoints, the "advertised.listeners" config value will be used directly if it's set. Otherwise if the "advertised.listeners" is not set, the logic goes to check if either of the "advertised.host" or "advertised.port" config is set, if so, the result will be a single endpoint, constructed using the listener name PLAINTEXT, the "advertised.host" config and the "advertised.port" config. If neither of "advertised.host" or "advertised.port" is set, the next step is to use the value for the **listeners** endpoints, whose value calculation is described in the previous bullet.

Proposed change

To support dedicated ports for controller connections, we need a way to specify the dedicated endpoints. We propose to support the new dedicated endpoints by adding new a new entry to the "listeners" and "advertised.listeners" config. For instance, if a cluster already has multiple listener names with config

```
listener.security.protocol.map=INTERNAL:PLAINTEXT,EXTERNAL:SSL
advertised.listeners=INTERNAL://broker1.example.com:9092,EXTERNAL://host1.example.com:9093
listeners=INTERNAL://192.1.1.8:9092,EXTERNAL://10.1.1.5:9093
```

in order to support the new endpoint for controller, it can be changed to

```
listener.security.protocol.map=CONTROLLER:PLAINTEXT,INTERNAL:PLAINTEXT,EXTERNAL:SSL
advertised.listeners=CONTROLLER://broker1.example.com:9091,INTERNAL://broker1.example.com:9092,EXTERNAL://host1.
example.com:9093
listeners=CONTROLLER://192.1.1.8:9091,INTERNAL://192.1.1.8:9092,EXTERNAL://10.1.1.5:9093
```

Upon startup, a broker should maintain the existing behavior by publishing all the endpoints in **advertised-listeners** to Zookeeper.

How can a controller learn about the dedicated endpoints exposed by brokers?

How does it work today?

Today each broker publishes a list of endpoints to Zookeeper, in the json format:

Broker Info exposed to Zookeeper

```
{
        "listener_security_protocol_map": {
        "INTERNAL": "PLAINTEXT",
        "EXTERNAL": "SSL"
    },
    "endpoints": [
        "INTERNAL://broker1.example.com:9092",
        "EXTERNAL://host1.example.com:9093"
    ],
    "host": "host1.example.com",
    "port": 9092,
    "jmx_port": -1,
    "timestamp": "1532467569343",
    "version": 4
}
```

Upon detecting a new broker through Zookeeper, the controller will figure out which endpoint it should use to connect to the new broker by first determining the the **inter-broker-listener-name**. The **inter-broker-listener-name** is decided by using either the "inter.broker.listener.name" config or the "security.inter.broker.protocol" config. Then by using the "endpoints" section of the broker info, the controller can determine which endpoint to use for the given **inter-broker-listener-name**. For instance, with the sample json payload listed above, if the controller first determines **inter-broker-listener-name** to be "INTERNAL", then it knows to use the endpoint "INTERNAL://broker1.example.com:9092" and security protocol PLAINTEXT to connect to the given broker.

Proposed change by using the "control.plane.listener.name" config

Instead of using the **inter-broker-listener-name** value, we propose to add a new config "control.plane.listener.name" for determining the control plane endpoints. For instance, if the controller sees that the exposed endpoints by a broker is the following:

Broker Info exposed to Zookeeper

```
{
        "listener_security_protocol_map": {
                "CONTROLLER": "PLAINTEXT"
        "INTERNAL": "PLAINTEXT",
        "EXTERNAL": "SSL"
    },
    "endpoints": [
                "CONTROLLER://broker1.example.com:9091",
        "INTERNAL://broker1.example.com:9092",
        "EXTERNAL://host1.example.com:9093"
    ],
    "host": "host1.example.com",
    "port": 9092,
    "jmx_port": -1,
    "timestamp": "1532467569343",
    "version": 4
}
```

and the "control.plane.listener.name" config is set to value "CONTROLLER", it will use the corresponding endpoint "CONTROLLER://broker1.example.com:9091" and the security protocol "PLAINTEXT" for connections with this broker.

Whenever the "control.plane.listener.name" is set, upon broker startup, we will validate its value and make sure it's different from the **inter-broker-listener-name** value.

If the "control.plane.listener.name" config is not set, the controller will fall back to the current behavior and use **inter-broker-listener-name** value to determine controller-to-broker endpoints.

How are controller requests handled over the dedicated connections?

With the dedicated endpoints for controller connections, upon startup a broker will use the "control.plane.listener.name" to look up the corresponding endpoint in the **listeners** list for binding. For instance, in the example given above, the broker will derive the dedicated endpoint to be "CONTROLLER://192.1.1.8:9091". Then it will have a new dedicated acceptor that binds to this endpoint, and listens for controller connections. When a connection is received, the socket will be given to a dedicated *control plane* processor thread (network thread). The dedicated processor thread reads controller requests from the socket and enqueues them to a new dedicated *control plane* request queue, whose capacity is 20 [2]. On the other side of the controller request queue, a dedicated *control plane* request handler thread will take requests out, and handles them in the same way as being done today. In summary, we are 1) adding a dedicated acceptor, 2) pinning one processor thread, 3) adding a new request queue, and 4) pinning one request handler thread for controller connections and requests. The two new threads are exclusively for requests from the controller and do not handle data plane requests.

The metrics

```
kafka.network:name=ControlPlaneRequestQueueSize,type=RequestChannel
kafka.network:name=ControlPlaneResponseQueueSize,type=RequestChannel
```

will be added to monitor the size of the new *control plane* request and response queues. Another two new metrics

```
kafka.network:name=ControlPlaneNetworkProcessorIdlePercent,type=SocketServer
kafka.server:name=ControlPlaneRequestHandlerIdlePercent,type=KafkaRequestHandlerPool
```

will be added to monitor the idle ratio of the new control plane network thread, and control plane request handler thread respectively.

Finally as a special case, if the "control.plane.listener.name" config is not set, then there is no way to tell the dedicated endpoint for controller. Hence there will be no dedicated acceptor, network processor, or request handler threads. The behavior should be exactly same as the current implementation.

Compatibility, Deprecation, and Migration Plan

- Impacts: Controller requests will not longer be blocked by data requests, which should mitigate the effect of stale metadata listed in the motivation section.
- Migration plan: 2 rounds of rolling upgrades are needed to pick up the proposed changes in this KIP. The goal of the first round is to add the controller endpoint, without adding the "control.plane.listener.name" config. Specifically, an endpoint with the controller listener name should be added to the "listeners" config, e.g. "CONTROLLER://192.1.1.8:9091"; if the "advertised.listeners" config is explicitly configured and is not getting its value from "listeners", the new endpoint for controller should also be added to "advertised.listeners". After the first round is completed, controller to brokers communication should still behave in the same way that uses the **inter-broker-listener-name,** since the "control.plane.listener.name" is not set yet. In the 2nd round, the "control.plane.listener.name" config should be set to the corresponding listener name, e.g. "CONTROLLER". During rolling upgrade of the 2nd round, controller to brokers traffic will start using the "control.plane.listener.name", and go through the proposed changes in this KIP.
- No special migration tools are needed.
- This KIP does not support dynamic update of config.controlPlaneListenerName, to add or remove the control-plane.

Rejected Alternatives

1. A few previous designs do not involve adding the dedicated endpoints, and focus on controller request prioritization after controller requests are read from the socket. However without the dedicated controller endpoints, a controller request can still be blocked in cases where the request queue for data requests is full. This is because today one processor thread can handle multiple connections, say 100 connections represented by connection0, ... connection99, among which connection0-98 are from clients, and connection99 is from the controller. Further let's assume after one selector polling, there are incoming requests on all connections. When the request queue is full, the processor thread will be blocked first when trying to enqueue the data request from connection0, and then possibly blocked again for the data request from connection1, ... etc even though the controller request is ready to be enqueued.

[1] There is another type of Controller request, which is the StopReplica request. Topic deletion uses the StopReplica request with the field deletePartitions set to true, hence delayed processing of such StopReplica requests can degrade the performance of the Topic deletion process. Whether topic deletion is more important than client requests may vary under different settings, and when topic deletion is more important, it'll be better to prioritize the StopReplica requests over data requests.

[2] The rationale behind the default value is that currently the max number of inflight requests from controller to broker is hard coded to be 1, meaning a broker should have at most one controller request from a given controller. However, during controller failovers, a broker might receive multiple controller requests from different controllers. Yet we expect it to be rare for the number of controller requests to go above 20.