KIP-649: Dynamic Client Configuration

Status

Current state: Under Discussion

Discussion thread: here

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

PR (In-progress): https://github.com/apache/kafka/pull/9101

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

Motivation

Producer and consumer configurations are currently reconfigurable only by restarting the client. Allowing the user to dynamically reconfigure misbehaving clients would eliminate the time consuming process of restarting one or multiple clients. This KIP proposes the mechanisms for dynamic configuration of the following client configs:

Producer Configs

- `acks`

Consumer Configs

- `session.timeout.ms`
- `heartbeat.interval.ms`

Background

Config entities

Client entity names are more dynamic than broker and topic entity names. For example, client quotas can be tied to a user principle that is associated with a session as well as a client-id which is a generic workload identifier. This is not as simple as a broker with a broker id, so dynamic client configs should also have similar expressibility and extensibility to that which was introduced in KIP-546 for client quotas. The ClientConfigs APIs will follow the design pattern of the ClientQuotas APIs with a few differences.

Config values

Quota values are limited to double-precision 64-bit IEEE 754 format in the APIs introduced in KIP-546. However, client config values are strings in a .properties file until the config values are parsed into their respective types based on the client's config definition. Dynamic client configs should also be strings so that dynamic support can be added for any type of config in the future. This also allows dynamic client configs to be parsed and validated in the same way as static client configs.

Hierarchy for resolving dynamic client configs

The hierarchy for resolving client quotas is rather complex. This is because quotas need to be set on every application in the system to achieve consistent results. If a quota is set for client A but not client B, client B can end up hoarding resources because it is not limited. This would potentially cause client A to do work at a lower rate than what the quota specifies. Having a robust hierarchy for quotas allows the user to set quotas on all applications rather easily and then fine tune as needed. However, the hierarchy for client configs does not need to have the same amount of depth. There will just be a dynamic default and a dynamic config so that dynamic client configuration is consistent with static client configuration (e.g. .properties file & client defaults). Since users should not be able to change each others dynamic configs, the entities in the hierarchy will be scoped by at least user principle and optionally by client-id.

Public Interfaces

Network Protocol

DescribeClientConfigs

```
{
  "apiKey": 50,
  "type": "request",
  "name": "DescribeClientConfigsRequest",
  "validVersions": "0",
  "flexibleVersions": "none",
  "fields": [
    { "name": "Components", "type": "[]ComponentData", "versions": "0+",
      "about": "Filter components to apply to config entities.", "fields": [
      { "name": "EntityType", "type": "string", "versions": "0+",
        "about": "The entity type that the filter component applies to." },
      { "name": "MatchType", "type": "int8", "versions": "0+",
        "about": "How to match the entity {0 = exact name, 1 = default name, 2 = any specified name}." },
      { "name": "Match", "type": "string", "versions": "0+", "nullableVersions": "0+",
        "about": "The string to match against, or null if unused for the match type." }
    ]},
    { "name": "SupportedConfigs", "type": "[]string", "versions": "0+", "nullableVersions": "0+",
      "about": "The configuration keys to register, or null if not registering configuration keys." },
    { "name": "ResolveEntity", "type": "bool", "versions": "0+",
      "about": "True if an application is requesting dynamic configs for itself, false otherwise." },
    { "name": "Strict", "type": "bool", "versions": "0+",
      "about": "Whether the match is strict, i.e. should exclude entities with unspecified entity types." }
  ]
}


{
  "apiKey": 50,
  "type": "response",
  "name": "DescribeClientConfigsResponse",
  "validVersions": "0",
  "flexibleVersions": "none",
  "fields": [
    { "name": "ThrottleTimeMs", "type": "int32", "versions": "0+",
      "about": "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." },
    { "name": "ErrorCode", "type": "int16", "versions": "0+",
      "about": "The error code, or `0` if the config description succeeded." },
    { "name": "ErrorMessage", "type": "string", "versions": "0+", "nullableVersions": "0+",
      "about": "The error message, or `null` if the config description succeeded." },
    { "name": "Entries", "type": "[]EntryData", "versions": "0+", "nullableVersions": "0+",
      "about": "A result entry.", "fields": [
      { "name": "Entity", "type": "[]EntityData", "versions": "0+",
        "about": "The config entity description.", "fields": [
        { "name": "EntityType", "type": "string", "versions": "0+",
          "about": "The entity type." },
        { "name": "EntityName", "type": "string", "versions": "0+", "nullableVersions": "0+",
          "about": "The entity name, or null if the default."
        ]},
              { "name": "Values", "type": "[]ValueData", "versions": "0+",
                  "about": "The config values for the entity.", "fields": [
          { "name": "Key", "type": "string", "versions": "0+",
            "about": "The configuration key." },
          { "name": "Value", "type": "string", "versions": "0+",
            "about": "The configuration value." }
        ]}
      ]}

    ]}
  ]
}
```

AlterClientConfigs

```
{
  "apiKey": 51,
  "type": "request",
  "name": "AlterClientConfigsRequest",
  "validVersions": "0",
  "flexibleVersions": "none",
  "fields": [
    { "name": "Entries", "type": "[]EntryData", "versions": "0+",
      "about": "The configuration entries to alter.", "fields": [
      { "name": "Entity", "type": "[]EntityData", "versions": "0+",
        "about": "The config entity to alter.", "fields": [
        { "name": "EntityType", "type": "string", "versions": "0+",
          "about": "The entity type." },
        { "name": "EntityName", "type": "string", "versions": "0+", "nullableVersions": "0+",
          "about": "The name of the entity, or null if the default." }
      ]},
      { "name": "Ops", "type": "[]OpData", "versions": "0+",
        "about": "An individual configuration entry to alter.", "fields": [
        { "name": "Key", "type": "string", "versions": "0+",
          "about": "The configuration key." },
        { "name": "Value", "type": "string", "versions": "0+",
          "about": "The value to set, otherwise ignored if the value is to be removed." },
        { "name": "Remove", "type": "bool", "versions": "0+",
          "about": "Whether the configuration value should be removed, otherwise set." }
      ]}
    ]},
    { "name": "ValidateOnly", "type": "bool", "versions": "0+",
      "about": "Whether the alteration should be validated, but not performed." }
  ]
}

{
  "apiKey": 51,
  "type": "response",
  "name": "AlterClientConfigsResponse",
  "validVersions": "0",
  "flexibleVersions": "none",
  "fields": [
    { "name": "ThrottleTimeMs", "type": "int32", "versions": "0+",
      "about": "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." },
    { "name": "Entries", "type": "[]EntryData", "versions": "0+",
      "about": "The configuration entries to alter.", "fields": [
      { "name": "ErrorCode", "type": "int16", "versions": "0+",
        "about": "The error code, or `0` if the config alteration succeeded." },
      { "name": "ErrorMessage", "type": "string", "versions": "0+", "nullableVersions": "0+",
        "about": "The error message, or `null` if the config alteration succeeded." },
      { "name": "Entity", "type": "[]EntityData", "versions": "0+",
        "about": "The config entity to alter.", "fields": [
        { "name": "EntityType", "type": "string", "versions": "0+",
          "about": "The entity type." },
        { "name": "EntityName", "type": "string", "versions": "0+", "nullableVersions": "0+",
          "about": "The name of the entity, or null if the default." }
      ]}
    ]}
  ]
}
```

Proposed Changes

Admin Client Changes

Admin client calls will be added to support {Describe, Alter}ClientConfigs.

Broker Changes

If a `{Describe, Alter}ClientConfigsRequest` is made without a user component, an InvalidRequest error code will be returned to the client. Apart from this, when the EntityRequest field is not set to true, the mechanics of the <user, client-id> or user config entity descriptions are very similar to the mechanics outlined in KIP-546. This is because the bulk of the code in the brokers that handles fetching client quota entity configs from zookeeper can be reused for dynamic client configs.

Default dynamic client configs will be stored in the children of the znode /config/users, while client-id specific dynamic client configs will be stored in the children of /config/users/<user>/clients.

The user config is updated when the client-id component is missing from an `AlterClientConfigsRequest`. The <user, client-id> config is updated otherwise.

When the broker handles a `DescribeClientConfigsRequest` that a client is making for it's own dynamic configs (e.g. ResolveEntity field set to true), the user config and the <user, client-id> config will be returned as one entity whose configs are resolved with the following order of precedence from most precedent to least precedent:

`/config/users/<user>/clients/<client-id>`

`/config/users/<user>`

For example, any config key value pairs found in `/config/users/<user>/clients/<client-id>` will override any config key value pairs found in `/config/users/<user>`. The final resolved map of configs will then be sent back to the client and will overwrite statically provided client configs.

Client quotas are stored in these znodes as well. However, all configs that are not quota configs are filtered out when constructing a `DescribeClientQuotasResponse`. Similar to this, all configs that are not dynamic client configs will be filtered out when constructing a `DescribeClientConfigsResponse`. The value for each key will also be validated against the allowed values for that key. For example, if the user tries to set acks=2, an InvalidRequest error code will be sent back. The client will also have to validate dynamic configs against user-provided configs, so the broker is only doing partial validation here. This is worth doing since partially validated configs may only be invalid for a subset of clients, whereas acks=2 would be invalid for all clients.

The same authorization that is necessary for `{Describe,Alter}ClientQuotas`, `CLUSTER` authorization, will be used when handling `{Describe, Alter}ClientConfigsRequest`.

Producer Changes

The Java producer will have a `DynamicProducerConfig` that will periodically fetch dynamic configs from the producer's IO thread asynchronously. The interval on which dynamic configs are fetched will be the same amount of time as the interval for MetadataRequest, `metadata.max.age.ms`. It will use `DescribeClientConfigsRequest` as the RPC, validate the dynamic configs returned in `DescribeConfigsResponse` against the user provided configs, and log any configurations that are accepted. The client will reconfigure its acks value by using a method in `DynamicProducerConfig` that gets the current value of `acks`. The dynamic `acks` config will take precedence over user provided `acks` config unless the user provided configs require `acks` to be a certain value, such as `enable.idempotence=true`. In this case the dynamic update will be ignored.

Consumer Changes

The `GroupCoordinator` in the broker receives a group member's session timeout upon the `JoinGroupRequest` and stores this with the rest of the group member's metadata. This means that to dynamically configure a consumer's session timeout, the consumer must send a `JoinGroupRequest`. Currently, this could trigger an expensive rebalance operation when members are stable. JoinGroup behavior will be changed so that the session timeout can be updated using JoinGroup without triggering a rebalance in stable group members. The Java consumer's initial `DescribeClientConfigsRequest` will still be done synchronously before the first JoinGroupRequest to avoid sending an unnecessary JoinGroupRequest.

The Java consumer will have a `DynamicConsumerConfig` that will periodically fetch dynamic configs. The interval on which dynamic configs are fetched will be the same amount of time as the interval for `MetadataRequest, metadata.max.age.ms`. It will use `DescribeConfigsRequest` as the RPC, validate the dynamic configs that are returned in `DescribeConfigsResponse` against the user provided configs and log any configurations that are accepted. The client will either reconfigure itself by changing the session timeout and heartbeat interval in the `GroupRebalanceConfig,` or discard the configs if the heartbeat interval is greater than or equal to the session timeout. The dynamic configs will take precedence over user provided client configs as long as the heartbeat interval is strictly less than the session timeout.

Command Line Changes

`kafka-configs.sh` will be extended to support the client configurations listed at the beginning of this KIP. The same entity types that are used for client quotas, users and `clients`, will be used for dynamic client configuration.

For example, the user can add the new configs supported with this KIP along with the quota configs that are supported for the admin client in KIP-546 to their default dynamic config. In this example the user mixes some dynamic client configs that this KIP introduces with the quota config `producer_byte_rate`:

```
bin/kafka-configs.sh --bootstrap-server localhost:9092 \
  --alter \
  --entity-type users \
  --entity-name alice \
  --add-config acks=-1,session.timeout.ms=11000,producer_byte_rate=50000
  Completed updating config for user alice.
```

The user can also add configs *specific to a client-id* that will override the user's *default dynamic configs*:

```
bin/kafka-configs.sh --bootstrap-server localhost:9092 \
  --alter \
  --entity-type users \
  --entity-name alice \
  --entity-type clients \
  --entity-name clientid-override \
  --add-config acks=0,heartbeat.interval.ms=2000,producer_byte_rate=60000
  Completed updating config for user alice.
```

The user can describe these configs the same way that client quotas are described with the users and `clients` entity types. To make this possible, *kafka-configs.sh* will be sending a `DescribeClientConfigsRequest` as well as the DescribeClientQuotasRequest. Dynamic client configs must at least be scoped by a user:

```
bin/kafka-configs.sh --bootstrap-server localhost:9092 \
  --describe \
  --entity-type users \
  --entity-name alice
Quota configs for user-principal 'alice' are producer_byte_rate=50000.0
Dynamic configs for user-principal 'alice' are session.timeout.ms=11000, acks=-1
```

They may optionally be scoped by a client-id:

```
bin/kafka-configs.sh --bootstrap-server localhost:9092 \
  --describe \
  --entity-type users \
  --entity-name alice \
  --entity-type clients \
  --entity-name clientid-override
Quota configs for user-principal 'alice', client-id 'clientid-override' are producer_byte_rate=60000.0
Dynamic configs for user-principal 'alice', client-id 'clientid-override' are heartbeat.interval.ms=2000, acks=0
```

If a client-id is not specified when describing, all of the <user, client-id> entity configs will be returned:

```
bin/kafka-configs.sh --bootstrap-server localhost:9092 \
  --describe \
  --entity-type users \
  --entity-name alice \
  --entity-type clients
Quota configs for user-principal 'alice', client-id 'clientid-override' are producer_byte_rate=60000.0
Dynamic configs for user-principal 'alice', client-id 'clientid-override' are heartbeat.interval.ms=2000, acks=0
Dynamic configs for user-principal 'alice', client-id '""' are acks=-1
```

The *default dynamic* config will be used in the case that the *client-id dynamic* config does not contain a key that the default does contain, but only if the client is requesting configs with the ResolveEntity flag set to true.

Any number of the configs that this KIP provides dynamic support for can be added or deleted with `--add-config` and `--delete-config`. They may optionally be mixed with quotas in the same command.

Compatibility, Deprecation, and Migration Plan

1. If a *new* client with this feature attempts to send a `DescribeClientConfigsRequest` to an *old* broker, the broker will send back an `InvalidRequest` error code and the client will disable this feature.

2. In the case that an *old* client is talking to a *new* broker, nothing will change since the old client will never send a DescribeClientConfigsRequest.
3. In the case that both the broker and client are up to date with this change, the client will take advantage of the feature.
4. The Java producers and consumers will register a list of configs that they support. This will be stored as the value of the dynamic config 'supported.configs' and can be returned to the user. If a new client registers with an entity the old value of this config will be overwritten.

Rejected Alternatives

- Introducing new entity types for `kafka-configs.sh` that producers and consumers can associate themselves with. This would make the tool more cumbersome to use and it is most intuitive that client configurations be dynamically altered with the `clients and users` entity types.
- Use the {Describe, IncrementalAlter}Configs APIs. Client config entities are more dynamic than entities with a singular resource name and type which makes it hard to fit them into generic APIs that expect a distinct entity name and type.
- Use the <user/client-id> hierarchy implemented for client quotas in KIP-55 and extended for the admin client in KIP-546. Quotas are inherently hierarchical but client configs are not, so it seems reasonable to use a hierarchy of shallow depth for dynamic client configs.
- **Making client config compatibility information available to the user**

  The user should be able to see what dynamic configs are supported for each application. However, clients that are using the same <user, client-id> entity may not necessarily support the same dynamic configs, storing a list of supported configs alongs side quotas and configs is a flawed solution.

  A better solution is to store config registrations in an internal topic. The Java producer and consumer clients can register the configs that they support with a DescribeClientConfigsRequest. The broker can write a key-value pair to an internal topic upon receiving the request where the key is the <user, client-id> entity and the value is ClientVersion along with the list of supported configs.

  All versions of clients that registered with a <user, client-id> entity along with the supported configs for each version of client could be aggregated when a DescribeClientConfigsRequest from an admin client is received. This information would then be returned to the admin client in the `DescribeClientConfigsResponse`. For example, supported dynamic configs for user-principal 'alice', client-id 'clientid-override' are "{ 'ClientInformation(softwareName=apache-kafka-java, softwareVersion=x.y.a-SNAPSHOT)': 'acks', 'ClientInformation(softwareName=apache-kafka-java, softwareVersion=x.y.b-SNAPSHOT)': 'acks, enable.idempotence' }".
- Interesting hierarchies for config overrides could be constructed if the Java producer and consumer resolved the dynamic configs instead of the broker. For example, from most precedent to least precedent:
  - /config/users/<user>/clients/<client-id>
  - .properties file configs
  - /config/users/<user>
  - Static default configs defined in `ProducerConfig` and `ConsumerConfig`.

- Adding a config enable.dynamic.config to producers and consumers to enable the feature. This defaulted to true anyway so it was removed.
- Making certain client configurations topic level configurations on the broker.

  - The semantic for the ProduceRequest API would be undefined since the producer would not receive a response with an offset for the ProduceRequests with `acks=0`.
  - If this were implemented for `acks` there would also be quite a bit of overhead associated with extra round trips since the RecordAccumulator sends batches that may contain records from multiple topics. If these topics have different `acks` configurations the records would need to be sent in different batches based on the `acks` value.
  - For example, if a producer is consistently producing to 2 different topics and one is configured as `acks=0` while the other is `acks=-1`. This would require twice the amount of round trips to produce the same number of messages.