KIP-48 Delegation token support for Kafka

Status

Current state: *Accepted*

Discussion thread: *here*

JIRA: *KAFKA-1696*

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

Motivation

We introduced support for security in kafka version 0.9.0. using kerberos as authentication layer. Kafka is designed to work with a lot of producers and consumers so in a secure environment all these clients will need access to a keytab or a TGT to ensure they can communicate with a secure kafka broker. This has few disadvantages:

- Performance/load on KDC as each client has to go to KDC to get the ticket.
- Renewal needs to go through KDC and this renewed TGT's need to be redistributed to all the clients.
- Blast Radius is large if the TGT is compromised as TGT may grant access to more than just kafka service
- Only compatible with kerberos authentication scheme.
- Administration cost as for any new client to work it must have access to keytab or some way to get a TGT from some other node.

Please read http://carfield.com.hk:8080/document/distributed/hadoop-security-design.pdf HDFS section for more detailed explanation of all the disadvantages above. To address the problems listed above we propose to add support for delegation tokens to secure Kafka. Delegation tokens are shared secret between kafka brokers and clients so authentication can be done without having to go through KDC.

Delegation tokens will help processing frameworks to distribute the workload to available workers in a secure environment without the added cost of distributing keytabs or TGT. i.e. In case of Storm, Storm's master (nimbus) is the only node that needs a keytab. Using this keytab Nimbus will authenticate with kafka broker and acquire a delegation token. Nimbus can then distribute this delegation token to all of its worker hosts and all workers will be able to authenticate to kafka using tokens and will have all the access that nimbus keytab principal has.

Public Interfaces

APIs and request/response classes

getDelegationToken(request: CreateDelegationTokenRequest): CreateDelegationTokenResponse

class CreateDelegationTokenRequest(renewer: Set[KafkaPrincipal] = Set.empty, maxLifeTime: long = -1)

class CreateDelegationTokenResponse(issueTimeMillis: long, expiryTimeMillis: long, maxLifeTime: long, tokenId: String, hmac: byte[])

renewDelegationToken(request: RenewDelegationTokenRequest): RenewDelegationTokenResponse

class RenewDelegationTokenRequest(hmac: byte[], expiryTimeMillis: long)

expireToken(request: ExpireDelegationTokenRequest): ExpireDelegationTokenResponse

class ExpireDelegationTokenRequest(hmac: byte[], expireAt: long = Systemtime.currentTimeMillis)

describeToken(request: DescribeDelegationTokenRequest): DescribeDelegationTokenResponse

class DescribeDelegationTokenRequest(owner: Set[KafkaPrincipal])

Protocol changes

CreateDelegationTokenRequest

```
CreateDelegationTokenRequest => [Renewer] MaxDateMs
  Renewer => string
  MaxDateMs => INT64
```

Field	Description
`Renewer`	Renewer is an Kafka PrincipalType+name string, who is allowed to renew this token before the max lifetime expires. If Renewer list is empty, then Renewer will default to the owner (Principal which requested this token).
`MaxDateMs`	Max lifetime for the token in milliseconds. If the value is -1, then MaxLifeTime will default to a server side config value (delegation.token.max.lifetime.ms).

CreateDelegationTokenResponse

```
CreateDelegationTokenResponse => ErrorCode TokenDetails
  ErrorCode => INT16
  TokenDetails =>  IssueDateMs ExpiryDateMs MaxDateMs TokenId HMAC
    IssueDateMs  => INT64
    ExpiryDateMs => INT64
    MaxDateMs => INT64
    TokenId => String
    HMAC => bytes
```

Field	Description
`IssueDateMs`	timestamp (in msec) when this token was generated. Unit is milliseconds since the beginning of the epoch (midnight Jan 1, 1970 (UTC)).
`MaxDateMs`	timestamp (in msec) at which this token expires. Unit is milliseconds since the beginning of the epoch (midnight Jan 1, 1970 (UTC)).
`ExpiryDateMs`	max life timestamp (in msec) of this token. Unit is milliseconds since the beginning of the epoch (midnight Jan 1, 1970 (UTC)).
`TokenId`	Sequence number to ensure uniqueness
`HMAC`	Keyed-hash message authentication code

Possible Error Codes

`* DelegationTokenDisabledException`

RenewDelegationTokenRequest

```
RenewDelegationTokenRequest => HMAC RenewPeriodMs
  HMAC => bytes
  RenewPeriodMs => INT64
```

Field	Description
HMAC	HMAC of the delegation token to be renewed
RenewPeriodMs	Renew Time period in milliseconds. If the value is -1, then Renew Time period will default to a server side config value (delegation.token.expiry.time.ms).

RenewDelegationTokenResponse

```
RenewDelegationTokenResponse => ErrorCode
   ErrorCode => INT32
   ExpiryDateMs => INT64
```

Field	Description
ErrorCode	
ExpiryDateMs	timestamp (in msec) at which this token expires. Unit is milliseconds since the beginning of the epoch (midnight Jan 1, 1970 (UTC))

Possible Error Codes

* DelegationTokenDisabledException

* TokenRenewerMismatchException

* TokenNotFoundException

ExpireDelegationTokenRequest

```
ExpireDelegationTokenRequest => HMAC expiryDateMs
  HMAC => bytes
  ExpiryDateMs => INT64
```

Field	Description
HMAC	HMAC of the delegation token to be renewed
ExpiryDateMs	Expiry time period in milliseconds. If the value is -1, then the token will get invalidated immediately.

ExpireDelegationTokenResponse

```
ExpireDelegationTokenResponse => ErrorCode
  ErroCode => INT32
  ExpiryDateMs => INT64
```

Field	Description
ErrorCode	
ExpiryDateMs	timestamp (in msec) at which this token expires. Unit is milliseconds since the beginning of the epoch (midnight Jan 1, 1970 (UTC)). -1 value will invalidate the token immediately

Possible Error Codes

* DelegationTokenDisabledException

* TokenRenewerMismatchException

* TokenNotFoundException

DescribeDelegationTokenRequest

```
DescribeDelegationTokenRequest => [Owner]
    Owner => String
```

Field	Description
ErrorCode	
Owner	Kakfa Principal which requested the delegation token. If the Owner list is null (i.e., length is -1), the response contains all the allowed tokens from all owners. If Owner list is empty, the response is empty list.

DescribeDelegationTokenResponse

```
DescribeDelegationTokenResponse => ErrorCode [TokenDetails]
  ErrorCode => INT16
  TokenDetails => Owner IssueDateMs ExpiryDateMs TokenId HMAC [Renewer]
    Owner => String
    IssueDateMs  => INT64
    ExpiryDateMs => INT64
    MaxDateMs => INT64
    TokenId => String
    HMAC => bytes
    Renewer => String
```

Field	Description
Owner	Kakfa Principal which requested the delegation token
IssueDateMs	timestamp (in msec) when this token was generated. Unit is milliseconds since the beginning of the epoch (midnight Jan 1, 1970 (UTC)).
MaxDateMs	max life timestamp (in msec) of this token. Unit is milliseconds since the beginning of the epoch (midnight Jan 1, 1970 (UTC)).
ExpiryDateMs	timestamp (in msec) at which this token expires. Unit is milliseconds since the beginning of the epoch (midnight Jan 1, 1970 (UTC)).
TokenId	Sequence number to ensure uniqueness
HMAC	Keyed-hash message authentication code
Renewer	Renewers list

Possible Error Codes

* DelegationTokenDisabledException

Configuration options

The following options will be added to `KafkaConfig.java` and can be configured as properties for Kafka server:

1. delegation.token.max.lifetime.ms : The token has a maximum lifetime beyond which it cannot be renewed anymore. Default value 7 days.
2. delegation.token.expiry.time.ms : The token validity time in seconds before the token needs to be renewed. Default value 1 day.

3. delegation.token.master.key : masterKey/secret to generate and verify delegation tokens. This masterKey/secret needs to be configured with all the brokers. Same secret key must be configured across all the brokers. If the masterKey/secret is not set or set to empty string, brokers will disable the delegation token support.

Proposed Changes

Token

The Kafka authentication token is modeled after the Hadoop user delegation token. The token will consist of:

TokenDetails:

- Owner ID -- Username that this token will authenticate as
- Renewers ID -- designated renewers list
- Issue date -- timestamp (in msec) when this token was generated
- Expiry date -- timestamp (in msec) at which this token expires
- Max Date - max life timestamp (in msec) of this token.
- TokenID – UUID to ensure uniqueness

TokenAuthenticator(HMAC) := HMAC_SHA1(master key, TokenID)

Authentication Token := (**TokenDetails**, TokenAuthenticator(HMAC))

Master Secret Key

The MasterKey/secret is used to generate and verify delegation tokens. This is supplied using config option. Same secret key must be configured across all the brokers. If the secret is not set or set to empty string, brokers will disable the delegation token support. The current proposal does not support rotation of secret.

Procedure to manually rotate the secret:

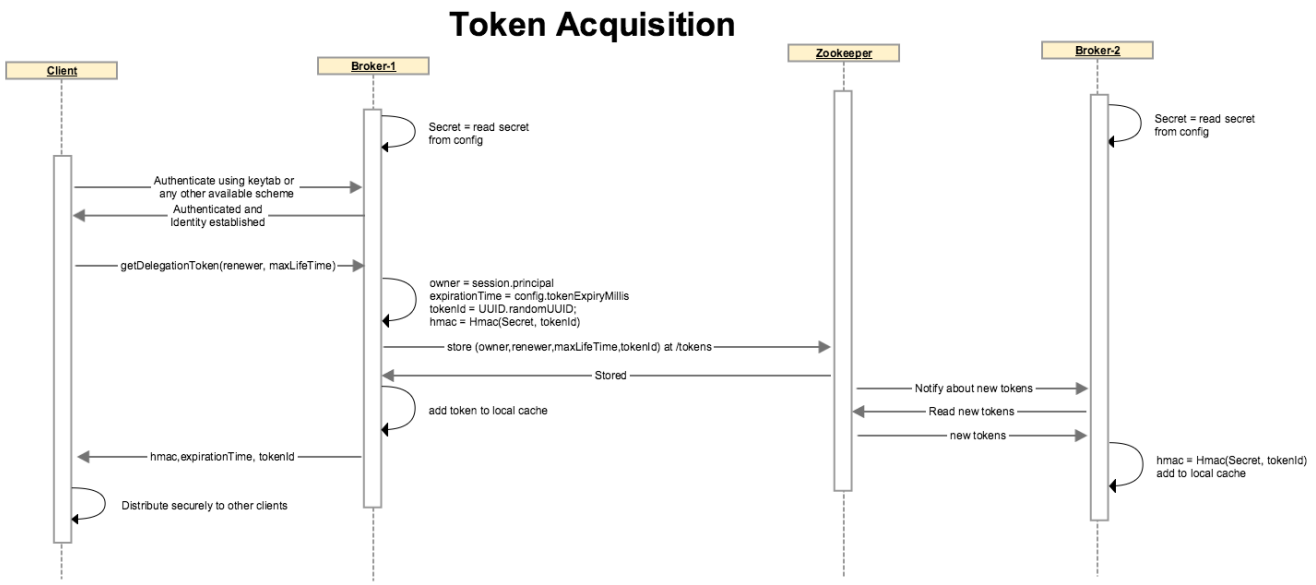
We require a re-deployment when the secret needs to be rotated. During this process, already connected clients will continue to work. But any new connection requests and renew/expire requests with old tokens can fail.

- expire all existing tokens
- rotate the secret by rolling upgrade, and
- generate new tokens

Token acquisition

Following steps describe how tokens can be acquired:

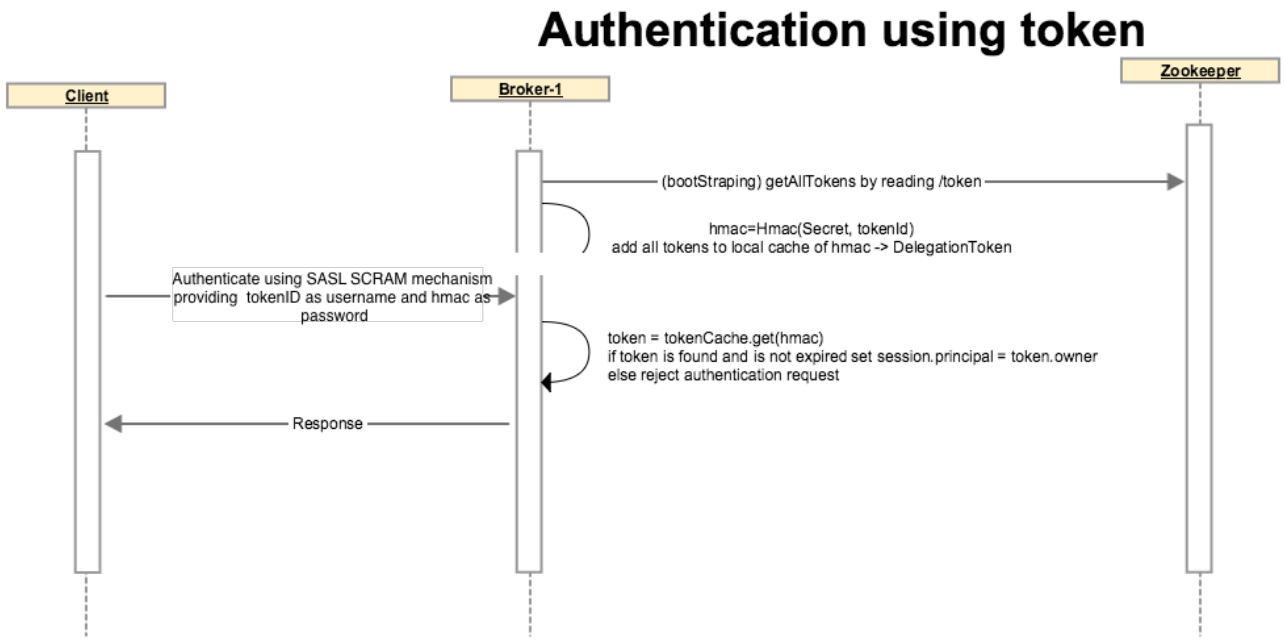
- A (Admin/DelegationToken) client connects with one of the kafka broker. Client must be authenticated using any of the available secure channels so it must have a way to authenticate, i.e. Kerberos keytab or TGT.
- Once a client is authenticated, it will make a broker side call to issue a delegation token. The request for delegation token will have to contain an optional renewer identity and max lifetime for token. The renewer is the user that is allowed to renew this token before the max lifetime expires. Renewer will default to the owner if not provided and Max life time will default to a server side config value (default days) Brokers will allow a token to be renewed until maxLifeTime but a token will still expire if not renewed by the expiry time. The expiry time will be a broker side configuration and will default to min (24 hours, maxlifeTime) . A Delegation Token request can be represented as class DelegationTokenRequest (renewer: Set[KafkaPrincipal], maxLifeTime: long). The owner is implicit in the request connection as the user who requested the delegation token.
- The broker generates a shared secret based on HMAC-SASM(a Password/Secret shared between all brokers, randomly generated tokenId). We can represent a token as scala case class DelegationToken(owner: KafkaPrincipal, renewer: Set[KafkaPrincipal], maxLifeTime: long, id: String, hmac: String, expirationTime: long)
- Broker stores this token in its in memory cache. Broker also stores the DelegationToken without the hmac in the zookeeper. As all brokers share the Password/Secret to generate the HMAC-SASM, they can read the request info from zookeeper , generate the hmac and store the delegation token in local cache.
- All brokers will have a cache backed by zookeeper so they will all get notified whenever a new token is generated and they will update their local cache whenever token state changes.
- Broker returns the token to Client. Client is expected to only make delegation token request over an encrypted channel so the token in encrypted over the wire.
- Client is free to distribute this token to other Kafka clients (Producer/Consumers). It is the client's responsibility to distribute the token securely.



Authentication using Delegation Token

We will reuse the current SASL channel for delegation token based authentication.

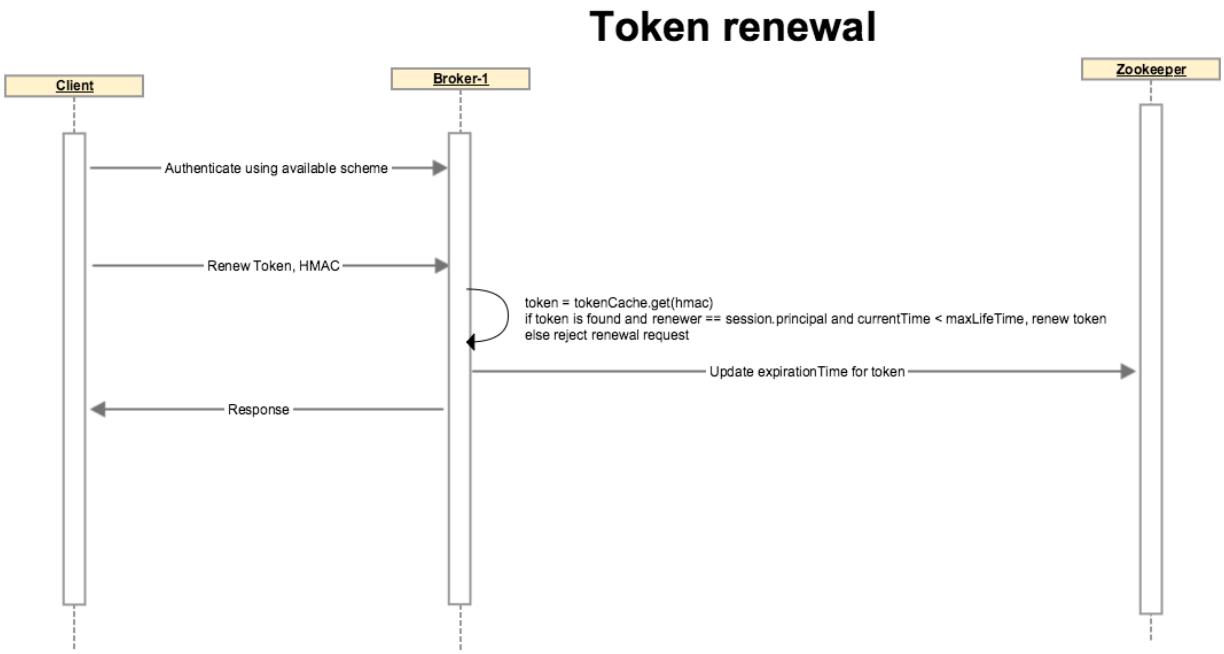
- SCRAM is a suitable mechanism for authentication using delegation tokens. KIP-84 proposes to support SASL SCRAM mechanisms. Kafka clients can authenticate using SCRAM-SHA-256, providing the delegation token HMAC as password.
- Server will look up the token from its token cache, if it finds a match and token is not expired it will authenticate the client and the identity will be established as the owner of the delegation token.
- If the token is not matched or token is expired, broker throws appropriate exception back and does not allow the client to continue.



Token renewal

- The (Admin/Delegation Token) client authenticates using Kerberos or any other available authentication scheme. A token can not be renewed if the initial authentication is done through delegation token, client must use a different auth scheme.
- Client sends a request to renew a token with an optional renew life time which must be < max life time of token.
- Broker looks up the token, if token is expired or if the renewer's identity does not match with the token's renewers, or if token renewal is beyond the Max life time of token, broker disallows the operation by throwing an appropriate exception.

- If none of the above conditions are matched, broker updates token's expiry. Note that the HMAC-SASM is unchanged so the token on client side is unchanged. Broker updates the expiration in its local cache and on zookeeper so other brokers also get notified and their cache statuses are updated as well.



Token expiration and cancellation

If a token is not renewed by the token's expiration time or if token is beyond the max life time, it will be deleted from all broker caches as well as from zookeeper. Periodic token expiry check thread will be run as part of Controller Broker. Alternatively, an owner or renewer can issue a expiration /cancellation by following a similar process as renewal.

Token Details in Zookeeper

Token is stored in Zookeeper as properties in the path /delegation_token/tokens/<tokenUID>. During server startup and token creation, scram credentials are generated and stored in memory (TokenCache).

Delegation Token Details

```
//Delegation Token Details for tokenID token123: Zookeeper persistence path /tokenauth/tokens/token123
{
   "version":1,
   "owner" : "owner",
   "renewer" : "renewer",
   "issueTimestamp" : "issueTimestamp",
   "maxTimestamp" : "maxTimestamp",
   "expiryTimestamp" : "expiryTimestamp",
   "tokenID" : "UUID",
};
```

SCRAM Extensions

SCRAM messages have an optional extensions field which is a comma-separated list of key=value pairs. After KIP-84 implementation , an extension will be added to the first client SCRAM message to indicate that authentication is being requested for a delegation token. This will enable Kafka broker to obtain credentials and principal using a different code path for delegation tokens.

JAAS configuration

Username/password specified in `jass config are` tokenID and token hmac. tokenId is used to retrieve the principal and token hmac on server side.

JAAS configuration for Clients

```
KafkaClient {
        org.apache.kafka.common.security.scram.ScramLoginModule required
        username="test123"
        password="ab24267ac3e827e00e57cdf98465baccecbbeced512e90a719026177e04e547e";
    tokenauth=true
};
```

DelegationToken Client

We will be providing a DelegationToken Client using which users can generate, renew and expire the tokens. This may part of AdminClient implementation (KIP-4).

DelegationTokenClient

```
public class DelegationTokenClient  {

    public TokenDetails generateToken(List<String> renewers, long maxLifeTime);

    public boolean renewToken(bytes[] hmac,  long renewPeriod);

    public boolean expireToken(bytes[] hmac, long expireTimeStamp);

    public boolean invalidateToken(bytes[] hmac);

    public List<TokenDetails> describeTokens();

    public void close();

}
```

Command line tool

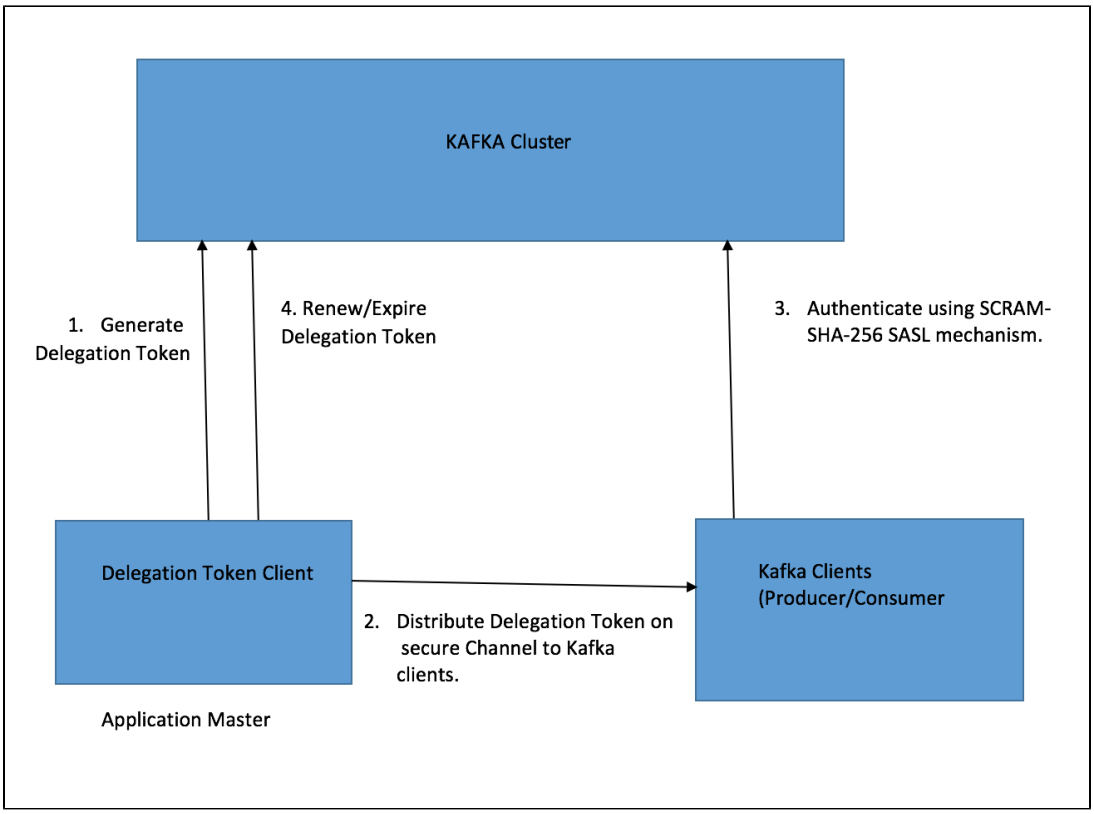
We will provide a commandline script to acquire delegation tokens, renew tokens, invalidate/expire and to describe tokens.

bin/kafka-delegation-token.sh --bootstrap-server broker1:9092 --create --renewer renewer1,renewer2 --max-life-time 1486750745585
bin/kafka-delegation-token.sh --bootstrap-server broker1:9092 --renew --hmac hmacString --renew-time-period 50745585
bin/kafka-delegation-token.sh --bootstrap-server broker1:9092 --expire --hmac hmacString --expiry-time-period 50745585
bin/kafka-delegation-token.sh --bootstrap-server broker1:9092 --describe --owner owner1,owner2

Changes to Java Clients (producer/consumer)

KIP-85 allows dynamic JAAS configuration for Kafka clients. After this we can easily configure the delegation token for SCRAM-SHA-256 authentication.

Below diagram shows the steps required to use the delegation tokens.



ACL

Currently, we only allow a user to create delegation token for that user only. Renew and expire requests should come from owner or renewers of the token. So we don't need ACLs for create/renew/expire requests.

Owners and the renewers can always describe their own tokens. To describe others tokens, we can add DESCRIBE operation on Token Resource. In future, when we extend the support to allow a user to acquire delegation tokens for other users, then we can enable CREATE/DELETE operations.

Operation	Resource	API
`DESCRIBE`	Token	describeTokens
`CREATE`	Token	createToken (Will be introduced in a future release)
`DELETE`	Token	deleteToken (Will be introduced in a future release)

Q/A

Q1. Is there any dependency on Hadoop APIs/Libraries?

A. No.

Future Work

1. Support for master key rotation. Some of the available alternatives are given in below section.
    - delegation.token.master.key could be a list, which would allow users to support both a new and old key at the same time while clients are upgrading tokens.
    - Use the controller to generate and rotate secret and distribute it to all brokers. Brokers will generate hmac based on *current* secret. The advantage is secret rotation can be more frequent and automated. The disadvantage is added complexity to push/pull tokens from the controller to brokers and brokers needs to keep a list of valid secrets till max(max life time of all tokens).

    - Let each broker generate a Random secret on each acquisition request and use this secret to generate the hmac. Broker will store the hmac and secret in zookeeper. However as zkClient does not support SSL the hmac will be on wire unencrypted which is not safe.
2. Support impersonation use cases: Allow users to acquire delegation tokens on behalf of other users