RpcConnectionRegistry and Security

Background

The ConnectionRegistry interface is a big improvement on reducing the dependency on ZooKeeper. First we implement a MasterRegistry, to fetch the bootstrap information from masters. And then we introduce a RpcConnectionRegistry, where we could also fetch the information from region servers, to reduce the load of master.
It generally works well and has already been released in the 2.5.x release line, but we found out that it can not work well when security is enabled. We have already found two problems:

HBASE-25051

Rpc based connection registry(both MasterRegistry and RpcConnectionRegistry) can not work together with token(digest) based auth.
This is because we need to use cluster id to determine which token to use, but we need to use the connection registry API for fetching cluster id, which introduces a cyclic dependency. So now we can only use null cluster id when creating the rpc client for rpc based connection registry implementation, which means we can not know which token is valid, thus we can not use token based auth for rpc based connection registry.

HBASE-28321

RpcConnectionRegistry is broken when security is enabled and we want to use different server principal patterns for master and region server.
This is specially for RpcConnectionRegistry. As after we introduce RpcConnectionRegistry, master and region server both implement the ClientMetaService, which could serve as bootstrap nodes for fetching bootstrap information.
But in our current rpc negotiation protocol, server will not tell client its server principal, so we use a SecurityInfo, to registry the rpc interface and the related server principal name(or pattern, as usually there is a FQDN in the server principal name) you need to use when you want to connect to a rpc server which implements the rpc interface.
So for ClientMetaService, as long as you do not know whether the remote server is master or region server, you can not determine which server principal pattern to use, even if you change the server principal field in SecurityInfo to an array or list.

Basic Idea

For HBASE-25051, since it is a cluster wide information, which is not very sensitive, I prefer we introduce some light-weighted ways to let clients get this information, without authentication,

even if authentication is enabled for the cluster. We could use something like a port unification service in front of our rpc server to serve the request for fetching cluster id.
For HBASE-28321, it should be part of our rpc negotiation, where the server should return its server principal to the client, to let the client acquire the necessary ticket for authentication. For me, I do not think this will increase the security risk. Of course, at client side, we should check whether the returned server principal at least matches one of our candidates recorded in SecurityInfo, if not, we should fail the connection setup as it may be a malicious server.

Design

Rpc Protocol Change

The current preamble header is constructed with:

4 bytes magic + 1 byte rpc version + 1 byte authentication mode

The magic header is ‘Hbas’.
So a possible way to add new logic here is to use different magic headers.

HBASE-25051

For implementing HBASE-25051, where we just want to get a cluster id, and then use it to create a Connection/AsyncConnection, we can send a 6 byte magic header ‘Regist’ as preamble header, and when server receives this header, it just returns the cluster id to client and then close the connection.

HBASE-28321

For implementing HBASE-28321, where we want to continue the negotiation after getting the server principal, we can send a 6 byte magic header ‘Securi’ as preamble header, and when server receives this header, it returns the server principal to client, and after client receives the server principal, it could start the normal rpc negotiation, send the normal preamble header, and then start the sasl negotiation.

Client Implementation

HBASE-25051

For HBASE-25051, we need to return the cluster id to the upper layer, so we’d better introduce a fake rpc interface, like ‘ConnectionRegistryService’, for getting cluster id. Inside the implementation of RpcClient, if we find out that this is a call from the ‘ConnectionRegistryService’ interface, we will use the special logic introduced in the above section for finishing the call.

And notice that, when creating the rpc client for getting cluster id, we should disable security. And why we still need a RpcClient, instead of just implementing a new network client is because we should not bypass the SSL negotiation, as it is done before sending any real data. Using the existing RpcClient can reuse the code for handling SSL negotiation.

HBASE-28321

First, we should extend the SecurityInfo, to make the server principal field a List. And then, when connecting, if SASL is enabled and we have more than one server principal candidate, we first send the ‘Securi’ preamble header to get the server principal, and then after getting the server principal, we use the server principal to start the normal rpc connection setup.
And to increase performance, after receiving the server principal, we could store it in the rpc connection, so next time when we want to setup the rpc connection again, we do not need to send the special preamble header again.

Server & Client Compatibility

New Server vs. Old Client

No problem at all since the old client will never send the special preamble handler, so everything just works as the old time.

Old Server vs. New Client

In HBase, in general we do not guarantee the compatibility between new client and old server, but it is still better to keep the compatibility. And since the RpcConnectionRegistry has already been released in the 2.5.x release line, what we want to do here could be considered as ‘bug fix’, which means we should also apply these changes to 2.5.x and then release them in a patch release, we have to keep compatible between new client and old server.
In general, the old server can not recognize the new preamble header, it will send back us a FatalConnectionException response, the message will be something like ‘Expected HEADER=...’. At the client side, we can decode this error message, and find out that it is an old server, and then fallback to using the old way to communicate with the server.
This means that we need to use the same way of sending bad preamble header responses to send the cluster id or server principal back to the client, so at client side, we can always decode the message, no matter if the server is old or new.

HBASE-25051

For HBASE-25051, if we find out the server is old, we just return the error message to upper layer since we are doing a rpc call, the upper layer could fallback to use null cluster id to set up the final RpcClient, which is the old behavior.

HBASE-28321

For HBASE-28321, ideally if we can not get the server principal back, we just randomly select a principal pattern to start the SASL negotiation. But the difficulty here is that the rpc calls are tied with the connection, so once the server closes the connection with an unexpected header, we will mark all the pending rpc calls as failure and there is no chance for us to retry.
There are basically several ways to deal with this problem:

1. Let it go. Even if we have the fallback logic, it could still fail if we choose the wrong server principal at client side, and the feature is completely broken between old client and old server under this scenario, at least we have fixed for new client and new server. And in our compatibility guide, we do not guarantee the compatibility between new client and old server.
2. Set a flag in the RpcConnection instance, when the upper layer issues a retry, we will skip the security preamble call, just randomly select a server principal to use.
3. Based on #2's effort, issue a special exception for this failure, and in AbstractRpcClient, do not finish the stub call with this exception, instead, just issue a new call to hide the retry logic to the upper layer.
4. Retry at rpc connection level(but we do not know how to do this, actually…)

In the first implementation, we just go with #1 since it is the easiest way, where we just do not implement any fallback logic.