Feed API Implementation Overview and Demo Note

Gefei Zhou

The Feed API is implemented in a web app project(feedreader.war) which would be deployed in tomcat container. It was developed by Java 1.7(JDK 7) and also needs ActiveMQ as well as Redis working as necessary components.

Data Model

There are three entities in the feed reader system: User, Feed and Article. Their relations are: User(many) to Feed(many), Feed(many) to Article(many), and User(many) to Article(many). Like inbox mail of Facebook, one article could be sent multiple users, so the Article and Feed relation is designed as many to many.
As there are many solution for object persistence, the APIs uses unique ids to represent these entities. Both relational database and non-relational database could be employed to store the full information of the entities using the unique id as their identifiers in Oracle or MongoDB.
The App will use articleId as the topic when sending and subscribing to certain topics in message queue.

Prerequisites and Manual Test

The application depends on ActiveMQ and Redis.
ActiveMQ - Install ActiveMQ(http://activemq.apache.org/download.html). ActiveMQ 5.6.0 Release is recommended. If ActiveMQ is unzipped on a windows machine, then navigate to its folder and launch it locally by running “activemq.bat” or “./activemq start” under the bin directory if it is going to run on Linux machine. The default port number of ActiveMQ is 61616, make sure it not being used by other software.

Redis- Install Redis(http://redis.io/download). Redis-2.4.6 is recommended. After Redis is installed, it can be launched by executing redis-server.exe for windows machine and redis-server for Linux machine. The default port number is 6379, make sure it not being used by other software.

Tomcat - Install Tomcat(https://tomcat.apache.org/tomcat-7.0-doc/appdev/installation.html) and configure environment variables. Tomcat 7.0.57 is recommended. The default port is 8080. The war file of this app should be put under the the “webapps” directory of tomcat home folder. And then user the startup.bat under “bin” to start the server.

Manual Test - After deploying the war file under tomcat and launched properly, then use browser to visit http://localhost:8080/feedreader/ to open the demo page. The demo page should show up a few input boxes and buttons with explanations. Each box is for the demo of one API. After type in some value in the input box and hit the button, it will render the results of the API at the bottom of each box

Test Case(s):
Use the “Create Feed”box to create feed, like create “f1” and “f2”.

Then use “Subscribe a User to Feed” box to subscribe a user to a feed, like “u1” ->”f1”, “u2” ->”f1”, “u2” ->”f2”.
Use “Publish Article” box to publish article to feed, like “f1” -> “a1”, “f2” -> “a2”.
Then use other boxes to call APIs to return each user’s articles and feeds. In this case, “u1” should have “a1” , and “u2” should have “a1” and “a2”. The server side debug messages should be displaying on the command line console like “Publisher: XXX sent Article YYY” or “ZZZ Read Article WWW” .

Feature

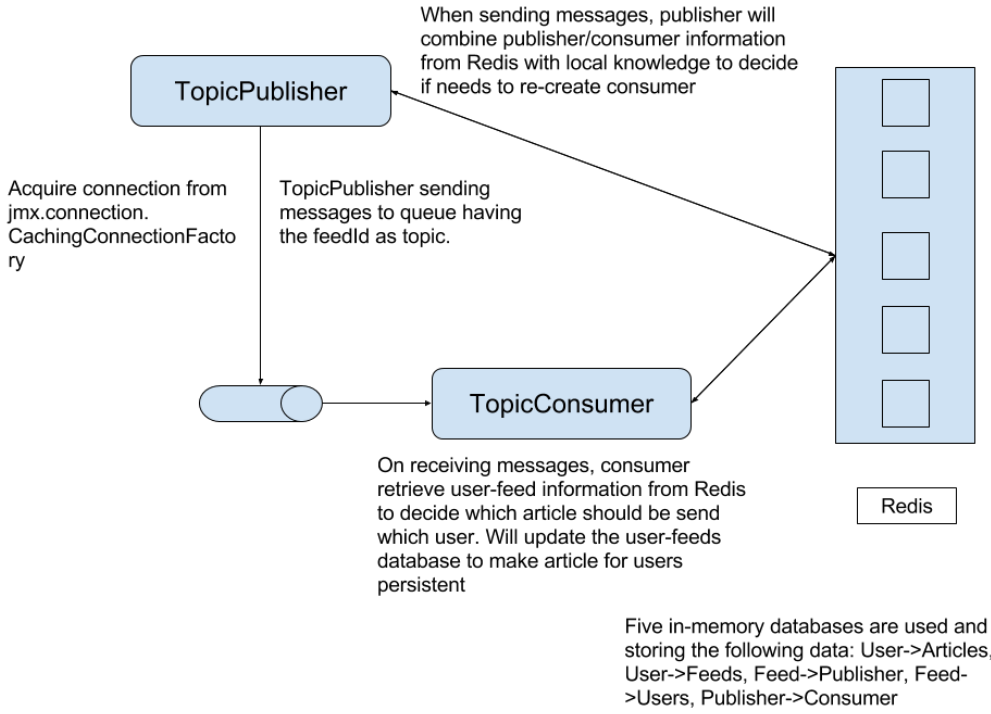
Support concurrency - This app chooses message queue and uses it in Publisher-Subscriber way to implement the functionalities of sending and receiving articles to/from feed. ActiveMQ’s publisher-subscriber mode fits the goal of one-to-many publishing very well. This app creates topic producer and consumer in multithreading way which allow publishers and consumers send/receive messages concurrently.
When the app needs to talk to Redis, the app uses a managed Redis client thread pool to create synchronized Redis client object to read and write Redis data store. As Redis in a in-memory data store and read/write data very fast, so combined with Redis this app is supposed to handle concurrency well.

Persistent - The app creates durable topic subscriber in ActiveMQ, so if a subscriber is down, it will not miss the messages sent to the topic it subscribed to. After restart, the subscriber is supposed to get all the messages sent while it is away.
As Redis stores the subscription and the relation data between entities, so if the app or Redis goes down, when they are up and running, the subscription and the relation data is still persistent and not missing.

Scalability - Working with message queue in a multithreading way, the app is able to process fairly large number of requests.
If the number of users increases quickly, it would scale up well by easily creating additional topic consumers to handle large number of users reading requests. If the number of incoming articles or publisher grows quickly, adding more queues on one machine or across multiple machines should handle this situation well.
And Redis is an in-memory datastore, if the data grows too fast, adding more physical memory could scale up with the increase of data. Redis also support sharding and can be deployed in cluster mode, which enables the app to deal with potential data increase in future.

Structural Design

The APIs were implemented in a WAR project and have http endpoints for manual testing.
Except the WAR, ActiveMQ is used in Publish-Subscribe mode to support reliable and efficient article publishing and receiving.
Redis is used as a in memory datastore to persist the relation information between these entities.
When a feed is create, an ActiveMQ topic consumer will be created and subscribe to this feedId.
When a user subscribes to a feed, their user-feed relation information is stored in Redis.
The interaction between these components is:



Once an article is published on a feed by sending a message binded to the feedId as topic, then the topic consumer will get this message through ActiveMQ. The consumer will retrieve user-feed relation information from Redis and then store the user-article pair in another DB in Redis to represent that the user has received the article.

Use Cases and Workflow

The key use cases are:

- Use Case 1 - Create Feed
  Client call(http.GET) the rest API *http://localhost:8080/feedreader/create/feed/{feed}* to create a feed.
  And a key value pair (feedId-publisherId) is inserted in Redis. The publisherId here represents an abstraction of the topic publisher in ActiveMQ.
  The method “com.feedreader.prototype.FeedWebService.createFeed” will insert a feed-publisher pair in Redis as well as a publisher-consumer pair. All these information will help recover the whole app from a restart.
  Meanwhile a topic consumer is create and subscribed to the feedId as topic.
- Use Case 2 - Subscribe a user to a feed
  Client call(http.GET) the rest API
  http://localhost:8080/feedreader/subscribe/{name}/feed/{feed} to create a feed. {name} is the userId and feed is the feedId.
  The method “com.feedreader.prototype.FeedWebService.subscribeOnFeed” will insert the feed into the feed list of this user in Redis as well as added the user into the user list of this feed. These two dbs in Redis are storing the user->feeds and feed->users mapping.
- Use Case 3 - Publish an article to a feed
  Client call(http.GET) the rest API
  http://localhost:8080/feedreader/feed/{feedId}/article/{article} to publish an article to the

feed. The method “com.feedreader.prototype.FeedWebService.publish” will first retrieve Redis to get the relation information about the feed and topic publisher and combine them with its local knowledge to decide whether it needs to recover some topic consumer. If a user has subscribed to a feed and the App shutdown, then after start, the App can use the information from Redis to recover the user-feed subscription. And due to Redis is disk-backed in-memory datastore, the user-articles information also remains after restart. After it made the decision about if it needs to recover topic consumer, it will create a publisher thread and then publish the article to the feed by sending message associated with the feedId to ActiveMQ. The jms.connection.CachingConnectionFactory will manage the total number of cached queue session size.

- Use Case 4 - Unsubscribe a user from a feed
  Client call(http.GET) the rest API *http://localhost:8080/feedreader/unsubscribe/{name}/feed/{feed}* to unsubscribe the user from the given feed. This API will remove the user from the users list of the feed and also remove the feed from the feeds list of the given user by updating the data in Redis.
- Use Case 5 - Get all the feeds that a user subscribing to
  Client call(http.GET) the rest API *http://localhost:8080/feedreader/feeds/subscriber/{name}* to get all the feeds that the given user is subscribing to.
- Use Case 6 - Get all the article that a user has following
  Client call(http.GET) the rest API *http://localhost:8080/feedreader/articles/subscriber/{name}* to get all the articles that the user is following. If a user U has followed Feed1 and Feed2, and Feed1 has publisher A1, Feed2 has published A2, then this API will return {A1, A2}.

Next Step

Security - Currently no authority/right check is performed at API level. So one next plan is to add security features to this feed read system to only give access to those permitted API user or IP range.

Database support - For simplicity and demo purpose, this application uses entities unique ids to represent those entities. But in order to make it more practical for real use, one essential step is to add domain object support for these entities, then it could incorporate relational or nonrelational database to store the entities for more complex operations.

Auto tests - Units and integration tests would be needed.