package com.twitter.search.ingester.pipeline.util;

import java.util.HashSet;

import java.util.Map;

import java.util.Set;

import java.util.concurrent.ConcurrentHashMap;

import com.google.common.collect.Sets;

import com.twitter.util.Future;

import com.twitter.util.Promise;

/\*\*

\* Batches single requests of type RQ -> Future<RP> to an underlying client that supports batch

\* calls with multiple values of type RQ. Threadsafe.

\*/

public class BatchingClient<RQ, RP> {

@FunctionalInterface

public interface BatchClient<RQ, RP> {

/\*\*

\* Issue a request to the underlying store which supports batches of requests.

\*/

Future<Map<RQ, RP>> batchGet(Set<RQ> requests);

}

/\*\*

\* unsentRequests is not threadsafe, and so it must be externally synchronized.

\*/

private final HashSet<RQ> unsentRequests = new HashSet<>();

private final ConcurrentHashMap<RQ, Promise<RP>> promises = new ConcurrentHashMap<>();

private final BatchClient<RQ, RP> batchClient;

private final int batchSize;

public BatchingClient(

BatchClient<RQ, RP> batchClient,

int batchSize

) {

this.batchClient = batchClient;

this.batchSize = batchSize;

}

/\*\*

\* Send a request and receive a Future<RP>. The future will not be resolved until at there at

\* least batchSize requests ready to send.

\*/

public Future<RP> call(RQ request) {

Promise<RP> promise = promises.computeIfAbsent(request, r -> new Promise<>());

maybeBatchCall(request);

return promise;

}

private void maybeBatchCall(RQ request) {

Set<RQ> frozenRequests;

synchronized (unsentRequests) {

unsentRequests.add(request);

if (unsentRequests.size() < batchSize) {

return;

}

// Make a copy of requests so we can modify it inside executeBatchCall without additional

// synchronization.

frozenRequests = new HashSet<>(unsentRequests);

unsentRequests.clear();

}

executeBatchCall(frozenRequests);

}

private void executeBatchCall(Set<RQ> requests) {

batchClient.batchGet(requests)

.onSuccess(responseMap -> {

for (Map.Entry<RQ, RP> entry : responseMap.entrySet()) {

Promise<RP> promise = promises.remove(entry.getKey());

if (promise != null) {

promise.become(Future.value(entry.getValue()));

}

}

Set<RQ> outstandingRequests = Sets.difference(requests, responseMap.keySet());

for (RQ request : outstandingRequests) {

Promise<RP> promise = promises.remove(request);

if (promise != null) {

promise.become(Future.exception(new ResponseNotReturnedException(request)));

}

}

return null;

})

.onFailure(exception -> {

for (RQ request : requests) {

Promise<RP> promise = promises.remove(request);

if (promise != null) {

promise.become(Future.exception(exception));

}

}

return null;

});

}

}