package com.twitter.search.earlybird\_root.filters;

import java.util.Optional;

import java.util.concurrent.ConcurrentHashMap;

import java.util.concurrent.ConcurrentMap;

import javax.inject.Inject;

import com.google.common.annotations.VisibleForTesting;

import com.google.common.cache.CacheBuilder;

import com.google.common.cache.CacheLoader;

import com.google.common.cache.LoadingCache;

import com.google.common.util.concurrent.RateLimiterProxy;

import com.google.common.util.concurrent.TwitterRateLimiterProxyFactory;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import com.twitter.finagle.Service;

import com.twitter.finagle.SimpleFilter;

import com.twitter.search.common.metrics.SearchCustomGauge;

import com.twitter.search.common.metrics.SearchLongGauge;

import com.twitter.search.common.metrics.SearchRateCounter;

import com.twitter.search.common.util.FinagleUtil;

import com.twitter.search.earlybird.common.ClientIdUtil;

import com.twitter.search.earlybird.thrift.EarlybirdRequest;

import com.twitter.search.earlybird.thrift.EarlybirdResponse;

import com.twitter.search.earlybird.thrift.EarlybirdResponseCode;

import com.twitter.search.earlybird.thrift.ThriftSearchResults;

import com.twitter.search.earlybird\_root.quota.ClientIdQuotaManager;

import com.twitter.search.earlybird\_root.quota.QuotaInfo;

import com.twitter.util.Future;

/\*\*

\* A filter that tracks and limits the per-client request rate. The ID of the client is determined

\* by looking at the Finagle client ID and the EarlybirdRequest.clientId field.

\*

\* The configuration currently has one config based implementation: see ConfigRepoBasedQuotaManager.

\*

\* If a client has a quota set, this filter will rate limit the requests from that client based on

\* that quota. Otherwise, the client is assumed to use a "common request pool", which has its own

\* quota. A quota for the common pool must always exist (even if it's set to 0).

\*

\* All rate limiters used in this class are tolerant to bursts. See TwitterRateLimiterFactory for

\* more details.

\*

\* If a client sends us more requests than its allowed quota, we keep track of the excess traffic

\* and export that number in a counter. However, we rate limit the requests from that client only if

\* the QuotaInfo returned from ClientIdQuotaManager has the shouldEnforceQuota property set to true.

\*

\* If a request is rate limited, the filter will return an EarlybirdResponse with a

\* QUOTA\_EXCEEDED\_ERROR response code.

\*/

public class ClientIdQuotaFilter extends SimpleFilter<EarlybirdRequest, EarlybirdResponse> {

private static final class ClientQuota {

private final QuotaInfo quotaInfo;

private final boolean shouldAllowRequest;

private final ClientIdRequestCounters requestCounters;

private ClientQuota(

QuotaInfo quotaInfo,

boolean shouldAllowRequest,

ClientIdRequestCounters requestCounters) {

this.quotaInfo = quotaInfo;

this.shouldAllowRequest = shouldAllowRequest;

this.requestCounters = requestCounters;

}

}

private static final class ClientIdRequestCounters {

private static final String REQUESTS\_RECEIVED\_COUNTER\_NAME\_PATTERN =

"quota\_requests\_received\_for\_client\_id\_%s";

private static final String THROTTLED\_REQUESTS\_COUNTER\_NAME\_PATTERN =

"quota\_requests\_throttled\_for\_client\_id\_%s";

private static final String REQUESTS\_ABOVE\_QUOTA\_COUNTER\_NAME\_PATTERN =

"quota\_requests\_above\_quota\_for\_client\_id\_%s";

private static final String REQUESTS\_WITHIN\_QUOTA\_COUNTER\_NAME\_PATTERN =

"quota\_requests\_within\_quota\_for\_client\_id\_%s";

private static final String PER\_CLIENT\_QUOTA\_GAUGE\_NAME\_PATTERN =

"quota\_for\_client\_id\_%s";

private final SearchRateCounter throttledRequestsCounter;

private final SearchRateCounter requestsReceivedCounter;

private final SearchRateCounter requestsAboveQuotaCounter;

private final SearchRateCounter requestsWithinQuotaCounter;

private final SearchLongGauge quotaClientGauge;

private ClientIdRequestCounters(String clientId) {

this.throttledRequestsCounter = SearchRateCounter.export(

String.format(THROTTLED\_REQUESTS\_COUNTER\_NAME\_PATTERN, clientId));

this.requestsReceivedCounter = SearchRateCounter.export(

String.format(REQUESTS\_RECEIVED\_COUNTER\_NAME\_PATTERN, clientId), true);

this.quotaClientGauge = SearchLongGauge.export(

String.format(PER\_CLIENT\_QUOTA\_GAUGE\_NAME\_PATTERN, clientId));

this.requestsAboveQuotaCounter = SearchRateCounter.export(

String.format(REQUESTS\_ABOVE\_QUOTA\_COUNTER\_NAME\_PATTERN, clientId));

this.requestsWithinQuotaCounter = SearchRateCounter.export(

String.format(REQUESTS\_WITHIN\_QUOTA\_COUNTER\_NAME\_PATTERN, clientId));

}

}

private static final String REQUESTS\_RECEIVED\_FOR\_EMAIL\_COUNTER\_NAME\_PATTERN =

"quota\_requests\_received\_for\_email\_%s";

// We have this aggregate stat only because doing sumany(...) on the

// per-client statistic is too expensive for an alert.

@VisibleForTesting

static final SearchRateCounter TOTAL\_REQUESTS\_RECEIVED\_COUNTER =

SearchRateCounter.export("total\_quota\_requests\_received", true);

private static final int DEFAULT\_BURST\_FACTOR\_SECONDS = 60;

private static final String QUOTA\_STAT\_CACHE\_SIZE = "quota\_stat\_cache\_size";

private static final String MISSING\_QUOTA\_FOR\_CLIENT\_ID\_COUNTER\_NAME\_PATTERN =

"quota\_requests\_with\_missing\_quota\_for\_client\_id\_%s";

private static final Logger LOG = LoggerFactory.getLogger(ClientIdQuotaFilter.class);

private final ConcurrentMap<String, RateLimiterProxy> rateLimiterProxiesByClientId =

new ConcurrentHashMap<>();

private final ClientIdQuotaManager quotaManager;

private final TwitterRateLimiterProxyFactory rateLimiterProxyFactory;

private final LoadingCache<String, ClientIdRequestCounters> clientRequestCounters;

private final LoadingCache<String, SearchRateCounter> emailRequestCounters;

/\*\* Creates a new ClientIdQuotaFilter instance. \*/

@Inject

public ClientIdQuotaFilter(ClientIdQuotaManager quotaManager,

TwitterRateLimiterProxyFactory rateLimiterProxyFactory) {

this.quotaManager = quotaManager;

this.rateLimiterProxyFactory = rateLimiterProxyFactory;

this.clientRequestCounters = CacheBuilder.newBuilder()

.build(new CacheLoader<String, ClientIdRequestCounters>() {

@Override

public ClientIdRequestCounters load(String clientId) {

return new ClientIdRequestCounters(clientId);

}

});

this.emailRequestCounters = CacheBuilder.newBuilder()

.build(new CacheLoader<String, SearchRateCounter>() {

@Override

public SearchRateCounter load(String email) {

return SearchRateCounter.export(

String.format(REQUESTS\_RECEIVED\_FOR\_EMAIL\_COUNTER\_NAME\_PATTERN, email));

}

});

SearchCustomGauge.export(QUOTA\_STAT\_CACHE\_SIZE, () -> clientRequestCounters.size());

}

@Override

public Future<EarlybirdResponse> apply(EarlybirdRequest request,

Service<EarlybirdRequest, EarlybirdResponse> service) {

String finagleClientId = FinagleUtil.getFinagleClientName();

String requestClientId = ClientIdUtil.getClientIdFromRequest(request);

LOG.debug(String.format("Client id from request or attribution: %s", requestClientId));

// Multiple client ids may be grouped into a single quota client id, all the

// unknown or unset client ids for example.

String quotaClientId = ClientIdUtil.getQuotaClientId(requestClientId);

LOG.debug(String.format("Client id used for checking quota: %s", quotaClientId));

ClientQuota clientQuota = getClientQuota(quotaClientId);

if (!clientQuota.shouldAllowRequest && clientQuota.quotaInfo.shouldEnforceQuota()) {

clientQuota.requestCounters.throttledRequestsCounter.increment();

return Future.value(getQuotaExceededResponse(

finagleClientId,

clientQuota.quotaInfo.getQuotaClientId(),

clientQuota.quotaInfo.getQuota()));

}

return service.apply(request);

}

private ClientQuota getClientQuota(String clientId) {

Optional<QuotaInfo> quotaInfoOptional = quotaManager.getQuotaForClient(clientId);

if (!quotaInfoOptional.isPresent()) {

SearchRateCounter noQuotaFoundForClientCounter = SearchRateCounter.export(

String.format(MISSING\_QUOTA\_FOR\_CLIENT\_ID\_COUNTER\_NAME\_PATTERN, clientId));

noQuotaFoundForClientCounter.increment();

}

// If a quota was set for this client, use it. Otherwise, use the common pool's quota.

// A quota for the common pool must always exist.

QuotaInfo quotaInfo = quotaInfoOptional.orElseGet(quotaManager::getCommonPoolQuota);

ClientIdRequestCounters requestCounters = clientRequestCounters

.getUnchecked(quotaInfo.getQuotaClientId());

emailRequestCounters.getUnchecked(quotaInfo.getQuotaEmail()).increment();

// Increment a stat for each request the filter receives.

requestCounters.requestsReceivedCounter.increment();

// Also increment the total stat

TOTAL\_REQUESTS\_RECEIVED\_COUNTER.increment();

// If shouldEnforceQuota is false, we already know that the request will be allowed.

// However, we still want to update the rate limiter and the stats.

final boolean requestAllowed;

if (quotaInfo.getQuota() == 0) {

// If the quota for this client is set to 0, then the request should not be allowed.

//

// Do not update the rate limiter's rate: RateLimiter only accepts positive rates, and in any

// case, we already know that the request should not be allowed.

requestAllowed = false;

} else {

// The quota is not 0: update the rate limiter with the new quota, and see if the request

// should be allowed.

RateLimiterProxy rateLimiterProxy = getClientRateLimiterProxy(quotaInfo.getQuotaClientId(),

quotaInfo.getQuota());

requestAllowed = rateLimiterProxy.tryAcquire();

}

// Report the current quota for each client

requestCounters.quotaClientGauge.set(quotaInfo.getQuota());

// Update the corresponding counter, if the request should not be allowed.

if (!requestAllowed) {

requestCounters.requestsAboveQuotaCounter.increment();

} else {

requestCounters.requestsWithinQuotaCounter.increment();

}

// Throttle the request only if the quota for this service should be enforced.

return new ClientQuota(quotaInfo, requestAllowed, requestCounters);

}

private RateLimiterProxy getClientRateLimiterProxy(String clientId, int rate) {

// If a RateLimiter for this client doesn't exist, create one,

// unless another thread beat us to it.

RateLimiterProxy clientRateLimiterProxy = rateLimiterProxiesByClientId.get(clientId);

if (clientRateLimiterProxy == null) {

clientRateLimiterProxy =

rateLimiterProxyFactory.createRateLimiterProxy(rate, DEFAULT\_BURST\_FACTOR\_SECONDS);

RateLimiterProxy existingClientRateLimiterProxy =

rateLimiterProxiesByClientId.putIfAbsent(clientId, clientRateLimiterProxy);

if (existingClientRateLimiterProxy != null) {

clientRateLimiterProxy = existingClientRateLimiterProxy;

}

LOG.info("Using rate limiter with rate {} for clientId {}.",

clientRateLimiterProxy.getRate(), clientId);

}

// Update the quota, if needed.

if (clientRateLimiterProxy.getRate() != rate) {

LOG.info("Updating the rate from {} to {} for clientId {}.",

clientRateLimiterProxy.getRate(), rate, clientId);

clientRateLimiterProxy.setRate(rate);

}

return clientRateLimiterProxy;

}

private static EarlybirdResponse getQuotaExceededResponse(

String finagleClientId, String quotaClientId, int quota) {

return new EarlybirdResponse(EarlybirdResponseCode.QUOTA\_EXCEEDED\_ERROR, 0)

.setSearchResults(new ThriftSearchResults())

.setDebugString(String.format(

"Client %s (finagle client ID %s) has exceeded its request quota of %d. "

+ "Please request more quota at go/searchquota.",

quotaClientId, finagleClientId, quota));

}

}