package com.twitter.product\_mixer.core.quality\_factor

import com.twitter.product\_mixer.core.pipeline.pipeline\_failure.ClientFailure

import com.twitter.product\_mixer.core.pipeline.pipeline\_failure.PipelineFailure

import com.twitter.product\_mixer.core.quality\_factor.QualityFactorConfig.defaultIgnorableFailures

import com.twitter.servo.util.CancelledExceptionExtractor

import com.twitter.util.Duration

import com.twitter.conversions.DurationOps.RichDuration

/\*\*

\* Quality factor is an abstract number that enables a feedback loop to control operation costs and ultimately

\* maintain the operation success rate. Abstractly, if operations/calls are too expensive (such as high

\* latencies), the quality factor should go down, which helps future calls to ease their demand/load (such as

\* reducing request width); if ops/calls are fast, the quality factor should go up, so we can incur more load.

\*/

sealed trait QualityFactorConfig {

/\*\*

\* specifies the quality factor min and max bounds and default value.

\*/

def qualityFactorBounds: BoundsWithDefault[Double]

/\*\*

\* initialDelay Specifies how much delay we should have before the quality factor calculation start to kick in. This is

\* mostly to ease the load during the initial warmup/startup.

\*/

def initialDelay: Duration

/\*\*

\* [[Throwable]]s that should be ignored when calculating

\* the [[QualityFactor]] if this is [[PartialFunction.isDefinedAt]]

\*/

def ignorableFailures: PartialFunction[Throwable, Unit] = defaultIgnorableFailures

}

object QualityFactorConfig {

/\*\*

\* Default value for [[QualityFactorConfig.ignorableFailures]] that ignores any

\* Cancelled requests and [[ClientFailure]]

\*/

val defaultIgnorableFailures: PartialFunction[Throwable, Unit] = {

case PipelineFailure(\_: ClientFailure, \_, \_, \_) => ()

case CancelledExceptionExtractor(\_) => ()

}

}

/\*\*

\* This is a linear quality factor implementation, aimed to achieve and maintain a percentile latency target.

\*

\* If we call quality factor q, target latency t and target percentile p,

\* then the q (quality factor) formula should be:

\* q += delta for each request with latency <= t

\* q -= delta \* p / (100 - p) for each request with latency > t ms or a timeout.

\*

\* When percentile p latency stays at target latency t, then based on the formula above, q will

\* stay constant (fluctuates around a constant value).

\*

\* For example, assume t = 100ms, p = p99, and q = 0.5

\* let's say, p99 latency stays at 100ms when q = 0.5. p99 means that out of every 100 latencies,

\* 99 times the latency is below 100ms and 1 time it is above 100ms. So based on the formula above,

\* q will increase by "delta" 99 times and it will decrease by delta \* p / (100 - p) = delta \* 99 once,

\* which results in the same q = 0.5.

\*

\* @param targetLatency This is the latency target, calls with latencies above which will cause quality

\* factor to go down, and vice versa. e.g. 500ms.

\* @param targetLatencyPercentile This the percentile where the target latency is aimed at. e.g. 95.0.

\* @param delta the step for adjusting quality factor. It should be a positive double. If delta is

\* too large, then quality factor will fluctuate more, and if it is too small, the

\* responsiveness will be reduced.

\*/

case class LinearLatencyQualityFactorConfig(

override val qualityFactorBounds: BoundsWithDefault[Double],

override val initialDelay: Duration,

targetLatency: Duration,

targetLatencyPercentile: Double,

delta: Double,

override val ignorableFailures: PartialFunction[Throwable, Unit] =

QualityFactorConfig.defaultIgnorableFailures)

extends QualityFactorConfig {

require(

targetLatencyPercentile >= 50.0 && targetLatencyPercentile < 100.0,

s"Invalid targetLatencyPercentile value: ${targetLatencyPercentile}.\n" +

s"Correct sample values: 95.0, 99.9. Incorrect sample values: 0.95, 0.999."

)

}

/\*\*

\* A quality factor provides component capacity state based on sampling component

\* Queries Per Second (qps) at local host level.

\*

\* If we call quality factor q, max qps R:

\* then the q (quality factor) formula should be:

\* q = Math.min([[qualityFactorBounds.bounds.maxInclusive]], q + delta) for each request that observed qps <= R on local host

\* q -= delta for each request that observed qps > R on local host

\*

\* When qps r stays below R, q will stay as constant (value at [[qualityFactorBounds.bounds.maxInclusive]]).

\* When qps r starts to increase above R, q will decrease by delta per request,

\* with delta being an additive factor that controls how sensitive q is when max qps R is exceeded.

\*

\* @param initialDelay Specifies an initial delay time to allow query rate counter warm up to start reflecting actual traffic load.

\* Qf value would only start to update after this initial delay.

\* @param maxQueriesPerSecond The max qps the underlying component can take. Requests go above this qps threshold will cause quality factor to go down.

\* @param queriesPerSecondSampleWindow The window of underlying query rate counter counting with and calculate an average qps over the window,

\* default to count with 10 seconds time window (i.e. qps = total requests over last 10 secs / 10).

\* Note: underlying query rate counter has a sliding window with 10 fixed slices. Therefore a larger

\* window would lead to a coarser qps calculation. (e.g. with 60 secs time window, it sliding over 6 seconds slice (60 / 10 = 6 secs)).

\* A larger time window also lead to a slower reaction to sudden qps burst, but more robust to flaky qps pattern.

\* @param delta The step for adjusting quality factor. It should be a positive double. If the delta is large, the quality factor

\* will fluctuate more and be more responsive to exceeding max qps, and if it is small, the quality factor will be less responsive.

\*/

case class QueriesPerSecondBasedQualityFactorConfig(

override val qualityFactorBounds: BoundsWithDefault[Double],

override val initialDelay: Duration,

maxQueriesPerSecond: Int,

queriesPerSecondSampleWindow: Duration = 10.seconds,

delta: Double = 0.001,

override val ignorableFailures: PartialFunction[Throwable, Unit] =

QualityFactorConfig.defaultIgnorableFailures)

extends QualityFactorConfig