package com.twitter.servo.util

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.util.{Duration, Local}

/\*\*

\* A strategy for tracking success rate, usually over a window

\*/

trait SuccessRateTracker { self =>

def record(successes: Int, failures: Int): Unit

def successRate: Double

/\*\*

\* A [[Gate]] whose availability is computed from the success rate (SR) reported by the tracker.

\*

\* @param availabilityFromSuccessRate function to calculate availability of gate given SR

\*/

def availabilityGate(availabilityFromSuccessRate: Double => Double): Gate[Unit] =

Gate.fromAvailability(availabilityFromSuccessRate(successRate))

/\*\*

\* A [[Gate]] whose availability is computed from the success rate reported by the tracker

\* with stats attached.

\*/

def observedAvailabilityGate(

availabilityFromSuccessRate: Double => Double,

stats: StatsReceiver

): Gate[Unit] =

new Gate[Unit] {

val underlying = availabilityGate(availabilityFromSuccessRate)

val availabilityGauge =

stats.addGauge("availability") { availabilityFromSuccessRate(successRate).toFloat }

override def apply[U](u: U)(implicit asT: <:<[U, Unit]): Boolean = underlying.apply(u)

}

/\*\*

\* Tracks number of successes and failures as counters, and success\_rate as a gauge

\*/

def observed(stats: StatsReceiver) = {

val successCounter = stats.counter("successes")

val failureCounter = stats.counter("failures")

new SuccessRateTracker {

private[this] val successRateGauge = stats.addGauge("success\_rate")(successRate.toFloat)

override def record(successes: Int, failures: Int) = {

self.record(successes, failures)

successCounter.incr(successes)

failureCounter.incr(failures)

}

override def successRate = self.successRate

}

}

}

object SuccessRateTracker {

/\*\*

\* Track success rate (SR) using [[RecentAverage]]

\*

\* Defaults success rate to 100% which prevents early failures (or periods of 0 data points,

\* e.g. tracking backend SR during failover) from producing dramatic drops in success rate.

\*

\* @param window Window size as duration

\*/

def recentWindowed(window: Duration) =

new AverageSuccessRateTracker(new RecentAverage(window, defaultAverage = 1.0))

/\*\*

\* Track success rate using [[WindowedAverage]]

\*

\* Initializes the windowedAverage to one window's worth of successes. This prevents

\* the problem where early failures produce dramatic drops in the success rate.

\*

\* @param windowSize Window size in number of data points

\*/

def rollingWindow(windowSize: Int) =

new AverageSuccessRateTracker(new WindowedAverage(windowSize, initialValue = Some(1.0)))

}

/\*\*

\* Tracks success rate using an [[Average]]

\*

\* @param average Strategy for recording an average, usually over a window

\*/

class AverageSuccessRateTracker(average: Average) extends SuccessRateTracker {

def record(successes: Int, failures: Int): Unit =

average.record(successes, successes + failures)

def successRate: Double = average.value.getOrElse(1)

}

/\*\*

\* EwmaSuccessRateTracker computes a failure rate with exponential decay over a time bound.

\*

\* @param halfLife determines the rate of decay. Assuming a hypothetical service that is initially

\* 100% successful and then instantly switches to 50% successful, it will take `halfLife` time

\* for this tracker to report a success rate of ~75%.

\*/

class EwmaSuccessRateTracker(halfLife: Duration) extends SuccessRateTracker {

// math.exp(-x) = 0.50 when x == ln(2)

// math.exp(-x / Tau) == math.exp(-x / halfLife \* ln(2)) therefore when x/halfLife == 1, the

// decay output is 0.5

private[this] val Tau: Double = halfLife.inNanoseconds.toDouble / math.log(2.0)

private[this] var stamp: Long = EwmaSuccessRateTracker.nanoTime()

private[this] var decayingFailureRate: Double = 0.0

def record(successes: Int, failures: Int): Unit = {

if (successes < 0 || failures < 0) return

val total = successes + failures

if (total == 0) return

val observation = (failures.toDouble / total) max 0.0 min 1.0

synchronized {

val time = EwmaSuccessRateTracker.nanoTime()

val delta = ((time - stamp) max 0L).toDouble

val weight = math.exp(-delta / Tau)

decayingFailureRate = (decayingFailureRate \* weight) + (observation \* (1.0 - weight))

stamp = time

}

}

/\*\*

\* The current success rate computed as the inverse of the failure rate.

\*/

def successRate: Double = 1.0 - failureRate

def failureRate = synchronized { decayingFailureRate }

}

private[servo] trait NanoTimeControl {

def set(nanoTime: Long): Unit

def advance(delta: Long): Unit

def advance(delta: Duration): Unit = advance(delta.inNanoseconds)

}

object EwmaSuccessRateTracker {

private[EwmaSuccessRateTracker] val localNanoTime = new Local[() => Long]

private[EwmaSuccessRateTracker] def nanoTime(): Long = {

localNanoTime() match {

case None => System.nanoTime()

case Some(f) => f()

}

}

/\*\*

\* Execute body with the time function replaced by `timeFunction`

\* WARNING: This is only meant for testing purposes.

\*/

private[this] def withNanoTimeFunction[A](

timeFunction: => Long

)(

body: NanoTimeControl => A

): A = {

@volatile var tf = () => timeFunction

localNanoTime.let(() => tf()) {

val timeControl = new NanoTimeControl {

def set(nanoTime: Long): Unit = {

tf = () => nanoTime

}

def advance(delta: Long): Unit = {

val newNanoTime = tf() + delta

tf = () => newNanoTime

}

}

body(timeControl)

}

}

private[this] def withNanoTimeAt[A](nanoTime: Long)(body: NanoTimeControl => A): A =

withNanoTimeFunction(nanoTime)(body)

private[servo] def withCurrentNanoTimeFrozen[A](body: NanoTimeControl => A): A =

withNanoTimeAt(System.nanoTime())(body)

}