package com.twitter.tweetypie

package config

import com.twitter.ads.internal.pcl.service.CallbackPromotedContentLogger

import com.twitter.ads.loggingclient.AdsLoggingClient

import com.twitter.adserver.thriftscala.AdCallbackEvent

import com.twitter.conversions.DurationOps.\_

import com.twitter.conversions.PercentOps.\_

import com.twitter.container.{thriftscala => ccs}

import com.twitter.deferredrpc.client.DeferredThriftService

import com.twitter.deferredrpc.thrift.Datacenter

import com.twitter.deferredrpc.thrift.DeferredRPC

import com.twitter.deferredrpc.thrift.Target

import com.twitter.escherbird.thriftscala.TweetEntityAnnotationService$FinagleClient

import com.twitter.escherbird.thriftscala.{

TweetEntityAnnotationService => TweetEntityAnnotationScroogeIface

}

import com.twitter.eventbus.client.EventBusPublisher

import com.twitter.eventbus.client.EventBusPublisherBuilder

import com.twitter.expandodo.thriftscala.CardsService$FinagleClient

import com.twitter.expandodo.thriftscala.{CardsService => CardsScroogeIface}

import com.twitter.finagle.\_

import com.twitter.finagle.builder.ClientBuilder

import com.twitter.finagle.client.Transporter

import com.twitter.finagle.factory.TimeoutFactory

import com.twitter.finagle.liveness.FailureAccrualFactory

import com.twitter.finagle.loadbalancer.Balancers

import com.twitter.finagle.mtls.authentication.EmptyServiceIdentifier

import com.twitter.finagle.mtls.client.MtlsClientBuilder.\_

import com.twitter.finagle.mtls.client.MtlsStackClient.\_

import com.twitter.finagle.partitioning.param

import com.twitter.finagle.service.TimeoutFilter.PropagateDeadlines

import com.twitter.finagle.service.\_

import com.twitter.finagle.ssl.OpportunisticTls

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.finagle.thrift.ThriftClientRequest

import com.twitter.finagle.thriftmux.MethodBuilder

import com.twitter.finagle.tracing.DefaultTracer

import com.twitter.flockdb.client.thriftscala.FlockDB

import com.twitter.flockdb.client.FlockResponse

import com.twitter.flockdb.client.TFlockClient

import com.twitter.flockdb.client.UserTimelineGraph

import com.twitter.geoduck.backend.hydration.thriftscala.{Hydration => GeoduckHydration}

import com.twitter.geoduck.backend.relevance.thriftscala.Relevance

import com.twitter.geoduck.backend.relevance.thriftscala.Relevance$FinagleClient

import com.twitter.geoduck.backend.relevance.thriftscala.RelevanceContext

import com.twitter.geoduck.service.common.clientmodules.GeoduckGeohashLocate

import com.twitter.geoduck.thriftscala.ReverseGeocoder

import com.twitter.geoduck.util.service.GeoduckLocate

import com.twitter.gizmoduck.thriftscala.UserService

import com.twitter.hashing.KeyHasher

import com.twitter.limiter.client.LimiterClientFactory

import com.twitter.mediainfo.server.thriftscala.MediaInfoService$FinagleClient

import com.twitter.mediainfo.server.thriftscala.{MediaInfoService => MediaInfoScroogeIface}

import com.twitter.merlin.thriftscala.UserRolesService

import com.twitter.passbird.thriftscala.PassbirdService

import com.twitter.passbird.thriftscala.PassbirdService$FinagleClient

import com.twitter.service.gen.scarecrow.thriftscala.ScarecrowService$FinagleClient

import com.twitter.service.gen.scarecrow.thriftscala.{ScarecrowService => ScarecrowScroogeIface}

import com.twitter.service.talon.thriftscala.Talon$FinagleClient

import com.twitter.service.talon.thriftscala.{Talon => TalonScroogeIface}

import com.twitter.snowflake.client.SnowflakeClient

import com.twitter.snowflake.thriftscala.Snowflake

import com.twitter.socialgraph.thriftscala.SocialGraphService$FinagleClient

import com.twitter.socialgraph.thriftscala.{SocialGraphService => SocialGraphScroogeIface}

import com.twitter.storage.client.manhattan.kv.Experiments

import com.twitter.storage.client.manhattan.kv.ManhattanKVClient

import com.twitter.storage.client.manhattan.kv.ManhattanKVClientMtlsParams

import com.twitter.storage.client.manhattan.kv.NoMtlsParams

import com.twitter.strato.client.Strato

import com.twitter.strato.client.{Client => StratoClient}

import com.twitter.timelineservice.fanout.thriftscala.FanoutService

import com.twitter.timelineservice.fanout.thriftscala.FanoutService$FinagleClient

import com.twitter.timelineservice.{thriftscala => tls}

import com.twitter.tweetypie.backends.\_

import com.twitter.tweetypie.client\_id.ClientIdHelper

import com.twitter.tweetypie.media.MediaClient

import com.twitter.tweetypie.service.ReplicatingTweetService.GatedReplicationClient

import com.twitter.tweetypie.storage.ManhattanTweetStorageClient

import com.twitter.tweetypie.storage.TweetStorageClient

import com.twitter.tweetypie.store.\_

import com.twitter.tweetypie.thriftscala.DeleteLocationData

import com.twitter.tweetypie.thriftscala.RetweetArchivalEvent

import com.twitter.tweetypie.thriftscala.TweetEvent

import com.twitter.tweetypie.thriftscala.TweetServiceInternal$FinagleClient

import com.twitter.user\_image\_service.thriftscala.UserImageService$FinagleClient

import com.twitter.user\_image\_service.thriftscala.{UserImageService => UserImageScroogeIface}

import com.twitter.util.Throw

import com.twitter.util.Timer

import com.twitter.util.{TimeoutException => UtilTimeoutException}

import scala.util.Random

trait BackendClients {

/\*\* returns all the finagle.Names created while building clients \*/

def referencedNames: Seq[Name]

val asyncRetryTweetService: ThriftTweetService

val asyncTweetDeletionService: ThriftTweetService

val asyncTweetService: ThriftTweetService

val configBus: ConfigBus

val creativesContainerService: CreativesContainerService

val darkTrafficClient: Service[Array[Byte], Array[Byte]]

val deleteLocationDataPublisher: EventBusPublisher[DeleteLocationData]

val escherbird: Escherbird

val expandodo: Expandodo

val fanoutServiceClient: FanoutService.MethodPerEndpoint

val geoHydrationLocate: GeoduckLocate

val geoRelevance: Relevance.MethodPerEndpoint

val geoScrubEventStore: GeoScrubEventStore

val geoduckGeohashLocate: GeoduckGeohashLocate

val gizmoduck: Gizmoduck

val gnipEnricherator: GnipEnricherator

val guano: Guano

val limiterService: LimiterService

val lowQoSReplicationClients: Seq[GatedReplicationClient]

val mediaClient: MediaClient

val mediaInfoService: MediaInfoService

val memcacheClient: memcached.Client

val merlin: UserRolesService.MethodPerEndpoint

val passbirdClient: PassbirdService.MethodPerEndpoint

val replicationClient: ThriftTweetService

val retweetArchivalEventPublisher: EventBusPublisher[RetweetArchivalEvent]

val scarecrow: Scarecrow

val snowflakeClient: SnowflakeClient.SnowflakeClient

val socialGraphService: SocialGraphService

val stratoserverClient: StratoClient

val talon: Talon

val tflockReadClient: TFlockClient

val tflockWriteClient: TFlockClient

val timelineService: TimelineService

val tweetEventsPublisher: EventBusPublisher[TweetEvent]

val tweetStorageClient: TweetStorageClient

val userImageService: UserImageService

val callbackPromotedContentLogger: CallbackPromotedContentLogger

}

/\*\*

\* default implementation of BackendClients that connects to real, remote

\* backend services.

\*/

object BackendClients {

// for most services, tweetypie typically maintains only a single connection to

// each host in the cluster, and that is enough for normal steady-state work.

// to prevent ddos'ing backends during unusual traffic influxes, we set the host

// connection limit to be 2-3x the steady-state daily peak, giving plenty of head

// room but without allowing an excessive number of connections.

private val defaultHostConnectionLimit = 3

// 100ms is greater than most gc pauses; smaller values cause more timeouts

private val defaultConnectTimeout = 100.milliseconds

// tcpConnect timeout is less than half of defaultConnectTimeout, to allow at least

// two tries (except when there is a GC pause)

private val defaultTcpConnectTimeout = 20.milliseconds

private val WriteExceptionsOnly: PartialFunction[Try[Nothing], Boolean] =

RetryPolicy.WriteExceptionsOnly

private val ClosedExceptionsOnly: PartialFunction[Try[Nothing], Boolean] = {

case Throw(\_: ChannelClosedException) => true

}

private val TimeoutExceptionsOnly: PartialFunction[Try[Nothing], Boolean] = {

case Throw(\_: TimeoutException) => true

case Throw(\_: UtilTimeoutException) => true

}

private val NoBackoff = Backoff.const(0.second)

private def retry(writeExceptions: Int = 100, closedExceptions: Int = 2, timeouts: Int = 0) =

RetryPolicy.combine(

RetryPolicy.backoff(NoBackoff.take(writeExceptions))(WriteExceptionsOnly),

RetryPolicy.backoff(NoBackoff.take(closedExceptions))(ClosedExceptionsOnly),

RetryPolicy.backoff(NoBackoff.take(timeouts))(TimeoutExceptionsOnly)

)

implicit val warmup: Warmup[BackendClients] = {

// Use a random string so that the keys are likely to hash to

// different memcache instances. Request multiple keys at a time so

// that we don't consider the backend warm just because we can get a

// bunch of successful responses to one cache.

val cacheGet = (\_: memcached.Client).get(Seq.fill(20)(Random.nextLong.toString))

Warmup

.empty[BackendClients]

.warmField(\_.expandodo)

.warmField(\_.gizmoduck)

.warmField(\_.memcacheClient)(Warmup("memcache")(cacheGet))

.warmField(\_.talon)

.warmField(\_.tweetStorageClient)(Warmup("tweetstorage")(\_.ping()))

.warmField(\_.tflockReadClient)(Warmup("tflock")(\_.contains(UserTimelineGraph, 0, 0)))

.warmField(\_.scarecrow)

.warmField(\_.socialGraphService)

.warmField(\_.timelineService)

.warmField(\_.geoRelevance)(Warmup("geo\_relevance")(\_.placeSearch(RelevanceContext())))

}

def apply(

settings: TweetServiceSettings,

deciderGates: TweetypieDeciderGates,

statsReceiver: StatsReceiver,

hostStatsReceiver: StatsReceiver,

timer: Timer,

clientIdHelper: ClientIdHelper,

): BackendClients = {

val thriftClientId = settings.thriftClientId

val tracer = DefaultTracer

val env = settings.env.toString

val zone = settings.zone

val log = Logger(getClass)

val backendsScope = statsReceiver.scope("backends")

/\*\* a Seq builder of finagle.Names loaded via getName \*/

val referencedNamesBuilder = Seq.newBuilder[Name]

/\*\* the default set of exceptions we believe are safe for Tweetypie to retry \*/

val defaultResponseClassifier: ResponseClassifier =

ResponseClassifier.RetryOnChannelClosed.orElse(ResponseClassifier.RetryOnTimeout)

/\*\*

\* Resolve a string into a Finagle Name and record it

\* in referencedNames.

\*/

def eval(address: String): Name = {

val name = Resolver.eval(address)

referencedNamesBuilder += name

name

}

def backendContext(name: String) =

Backend.Context(timer, backendsScope.scope(name))

// by default, retries on most exceptions (see defaultRetryExceptions). if an rpc is not

// idempotent, it should use a different retry policy.

def clientBuilder(name: String) = {

ClientBuilder()

.name(name)

.reportTo(statsReceiver)

.reportHostStats(hostStatsReceiver)

.tracer(tracer)

.daemon(true)

.tcpConnectTimeout(defaultTcpConnectTimeout)

.connectTimeout(defaultConnectTimeout)

.retryPolicy(retry())

}

def thriftMuxClientBuilder(name: String, address: String, clazz: Class[\_]) = {

clientBuilder(name)

.stack(

ThriftMux.client

.withClientId(thriftClientId)

.withOpportunisticTls(OpportunisticTls.Required)

.withServiceClass(clazz))

.loadBalancer(balancer())

.dest(eval(address))

.mutualTls(settings.serviceIdentifier)

}

// Our base ThriftMux.Client

// Prefer using thriftMuxMethodBuilder below but

// can be used to build custom clients (re: darkTrafficClient)

def thriftMuxClient(name: String, propagateDeadlines: Boolean = true): ThriftMux.Client = {

ThriftMux.client

.withClientId(thriftClientId)

.withLabel(name)

.withStatsReceiver(statsReceiver)

.withTracer(tracer)

.withTransport.connectTimeout(defaultTcpConnectTimeout)

.withSession.acquisitionTimeout(defaultConnectTimeout)

.withMutualTls(settings.serviceIdentifier)

.withOpportunisticTls(OpportunisticTls.Required)

.configured(PropagateDeadlines(enabled = propagateDeadlines))

}

// If an endpoint is non-idempotent you should add .nonidempotent and

// leave off any ResponseClassifiers (it will remove any placed before but not after)

// If it is unequivocally idempotent you should add .idempotent and

// leave off any ResponseClassifiers (it will retry on all Throws). This will also

// enable backup requests

def thriftMuxMethodBuilder(

name: String,

dest: String,

): MethodBuilder = {

thriftMuxClient(name)

.withLoadBalancer(balancer(minAperture = 2))

.methodBuilder(dest)

.withRetryForClassifier(defaultResponseClassifier)

.withTimeoutTotal(2.seconds) // total timeout including 1st attempt and up to 2 retries

}

def balancer(minAperture: Int = 2) = Balancers.aperture(minAperture = minAperture)

val eventBusPublisherBuilder =

EventBusPublisherBuilder()

.dest(eval("/s/eventbus/provisioning"))

.clientId(settings.thriftClientId)

// eventbus stats are further scoped by stream, so put all

// publishers under the same stats namespace

.statsReceiver(backendsScope.scope("event\_bus"))

// This makes the underlying kps-client to be resolved over WilyNs vs DNS

.serviceIdentifier(settings.serviceIdentifier)

new BackendClients {

def referencedNames: Seq[Name] = referencedNamesBuilder.result()

val memcacheClient: memcached.Client =

Memcached.client

.withMutualTls(settings.serviceIdentifier)

.connectionsPerEndpoint(2)

.configured(param.KeyHasher(KeyHasher.FNV1\_32))

.configured(Transporter.ConnectTimeout(100.milliseconds))

.configured(TimeoutFilter.Param(200.milliseconds))

.configured(TimeoutFactory.Param(200.milliseconds))

.configured(param.EjectFailedHost(false))

.configured(FailureAccrualFactory.Param(numFailures = 20, markDeadFor = 30.second))

.configured(

PendingRequestFilter.Param(limit = Some(settings.cacheClientPendingRequestLimit))

)

.filtered(new MemcacheExceptionLoggingFilter)

.newRichClient(dest = eval(settings.twemcacheDest), label = "memcache")

/\* clients \*/

val tweetStorageClient: TweetStorageClient =

Manhattan.fromClient(

new ManhattanTweetStorageClient(

settings.tweetStorageConfig,

statsReceiver = backendsScope.scope("tweet\_storage"),

clientIdHelper = clientIdHelper,

)

)

val socialGraphService: SocialGraphService = {

val finagleClient =

new SocialGraphService$FinagleClient(

thriftMuxClientBuilder(

"socialgraph",

"/s/socialgraph/socialgraph",

classOf[SocialGraphScroogeIface.MethodPerEndpoint]

).loadBalancer(Balancers.aperturePeakEwma(minAperture = 16))

.build()

)

settings.socialGraphSeviceConfig(

SocialGraphService.fromClient(finagleClient),

backendContext("socialgraph")

)

}

val tflockClient =

new FlockDB.FinagledClient(

thriftMuxClientBuilder("tflock", "/s/tflock/tflock", classOf[FlockDB.MethodPerEndpoint])

.loadBalancer(balancer(minAperture = 5))

.responseClassifier(FlockResponse.classifier)

.build(),

serviceName = "tflock",

stats = statsReceiver

)

val tflockReadClient: TFlockClient =

settings.tflockReadConfig(tflockClient, backendContext("tflock"))

val tflockWriteClient: TFlockClient =

settings.tflockWriteConfig(tflockClient, backendContext("tflock"))

val gizmoduck: Gizmoduck = {

val clientBuilder =

thriftMuxClientBuilder(

"gizmoduck",

"/s/gizmoduck/gizmoduck",

classOf[UserService.MethodPerEndpoint])

.loadBalancer(balancer(minAperture = 63))

val mb = MethodBuilder

.from(clientBuilder)

.idempotent(maxExtraLoad = 1.percent)

.servicePerEndpoint[UserService.ServicePerEndpoint]

val gizmoduckClient = ThriftMux.Client.methodPerEndpoint(mb)

settings.gizmoduckConfig(Gizmoduck.fromClient(gizmoduckClient), backendContext("gizmoduck"))

}

val merlin: UserRolesService.MethodPerEndpoint = {

val thriftClient = thriftMuxMethodBuilder("merlin", "/s/merlin/merlin")

.withTimeoutPerRequest(100.milliseconds)

.withTimeoutTotal(400.milliseconds)

.idempotent(0.01)

.servicePerEndpoint[UserRolesService.ServicePerEndpoint]

ThriftMux.Client.methodPerEndpoint(thriftClient)

}

val talon: Talon = {

val talonClient =

new Talon$FinagleClient(

thriftMuxClientBuilder(

"talon",

"/s/talon/backend",

classOf[TalonScroogeIface.MethodPerEndpoint])

.build()

)

settings.talonConfig(Talon.fromClient(talonClient), backendContext("talon"))

}

val guano = Guano()

val mediaInfoService: MediaInfoService = {

val finagleClient =

new MediaInfoService$FinagleClient(

thriftMuxClientBuilder(

"mediainfo",

"/s/photurkey/mediainfo",

classOf[MediaInfoScroogeIface.MethodPerEndpoint])

.loadBalancer(balancer(minAperture = 75))

.build()

)

settings.mediaInfoServiceConfig(

MediaInfoService.fromClient(finagleClient),

backendContext("mediainfo")

)

}

val userImageService: UserImageService = {

val finagleClient =

new UserImageService$FinagleClient(

thriftMuxClientBuilder(

"userImage",

"/s/user-image-service/uis",

classOf[UserImageScroogeIface.MethodPerEndpoint])

.build()

)

settings.userImageServiceConfig(

UserImageService.fromClient(finagleClient),

backendContext("userImage")

)

}

val mediaClient: MediaClient =

MediaClient.fromBackends(

userImageService = userImageService,

mediaInfoService = mediaInfoService

)

val timelineService: TimelineService = {

val timelineServiceClient =

new tls.TimelineService$FinagleClient(

thriftMuxClientBuilder(

"timelineService",

"/s/timelineservice/timelineservice",

classOf[tls.TimelineService.MethodPerEndpoint])

.loadBalancer(balancer(minAperture = 13))

.build()

)

settings.timelineServiceConfig(

TimelineService.fromClient(timelineServiceClient),

backendContext("timelineService")

)

}

val expandodo: Expandodo = {

val cardsServiceClient =

new CardsService$FinagleClient(

thriftMuxClientBuilder(

"expandodo",

"/s/expandodo/server",

classOf[CardsScroogeIface.MethodPerEndpoint])

.loadBalancer(balancer(minAperture = 6))

.build()

)

settings.expandodoConfig(

Expandodo.fromClient(cardsServiceClient),

backendContext("expandodo")

)

}

val creativesContainerService: CreativesContainerService = {

val mb = thriftMuxMethodBuilder(

"creativesContainerService",

"/s/creatives-container/creatives-container",

).withTimeoutTotal(300.milliseconds)

.idempotent(maxExtraLoad = 1.percent)

.servicePerEndpoint[ccs.CreativesContainerService.ServicePerEndpoint]

settings.creativesContainerServiceConfig(

CreativesContainerService.fromClient(ccs.CreativesContainerService.MethodPerEndpoint(mb)),

backendContext("creativesContainerService")

)

}

val scarecrow: Scarecrow = {

val scarecrowClient = new ScarecrowService$FinagleClient(

thriftMuxClientBuilder(

"scarecrow",

"/s/abuse/scarecrow",

classOf[ScarecrowScroogeIface.MethodPerEndpoint])

.loadBalancer(balancer(minAperture = 6))

.build(),

serviceName = "scarecrow",

stats = statsReceiver

)

settings.scarecrowConfig(Scarecrow.fromClient(scarecrowClient), backendContext("scarecrow"))

}

val snowflakeClient: Snowflake.MethodPerEndpoint = {

eval("/s/snowflake/snowflake") // eagerly resolve the serverset

val mb = thriftMuxMethodBuilder(

"snowflake",

"/s/snowflake/snowflake"

).withTimeoutTotal(300.milliseconds)

.withTimeoutPerRequest(100.milliseconds)

.idempotent(maxExtraLoad = 1.percent)

SnowflakeClient.snowflakeClient(mb)

}

val deferredRpcClient =

new DeferredRPC.FinagledClient(

thriftMuxClientBuilder(

"deferredrpc",

"/s/kafka-shared/krpc-server-main",

classOf[DeferredRPC.MethodPerEndpoint])

.requestTimeout(200.milliseconds)

.retryPolicy(retry(timeouts = 3))

.build(),

serviceName = "deferredrpc",

stats = statsReceiver

)

def deferredTweetypie(target: Target): ThriftTweetService = {

// When deferring back to the local datacenter, preserve the finagle

// context and dtabs. This will ensure that developer dtabs are honored

// and that context is preserved in eventbus. (eventbus enqueues only

// happen in async requests within the same datacenter.)

//

// Effectively, this means we consider deferredrpc requests within the

// same datacenter to be part of the same request, but replicated

// requests are not.

val isLocal: Boolean = target.datacenter == Datacenter.Local

val deferredThriftService: Service[ThriftClientRequest, Array[Byte]] =

new DeferredThriftService(

deferredRpcClient,

target,

serializeFinagleContexts = isLocal,

serializeFinagleDtabs = isLocal

)

new TweetServiceInternal$FinagleClient(deferredThriftService)

}

val replicationClient: ThriftTweetService =

deferredTweetypie(Target(Datacenter.AllOthers, "tweetypie-replication"))

// used for read endpoints replication

val lowQoSReplicationClients: Seq[GatedReplicationClient] = {

val rampUpGate = Gate.linearRampUp(Time.now, settings.forkingRampUp)

// Gates to avoid sending replicated reads from a cluster to itself

val inATLA = if (settings.zone == "atla") Gate.True else Gate.False

val inPDXA = if (settings.zone == "pdxa") Gate.True else Gate.False

Seq(

GatedReplicationClient(

client = deferredTweetypie(Target(Datacenter.Atla, "tweetypie-lowqos")),

gate = rampUpGate & deciderGates.replicateReadsToATLA & !inATLA

),

GatedReplicationClient(

client = deferredTweetypie(Target(Datacenter.Pdxa, "tweetypie-lowqos")),

gate = rampUpGate & deciderGates.replicateReadsToPDXA & !inPDXA

)

)

}

// used for async operations in the write path

val asyncTweetService: ThriftTweetService =

deferredTweetypie(Target(Datacenter.Local, "tweetypie"))

// used to trigger asyncEraseUserTweetsRequest

val asyncTweetDeletionService: ThriftTweetService =

deferredTweetypie(Target(Datacenter.Local, "tweetypie-retweet-deletion"))

// used for async retries

val asyncRetryTweetService: ThriftTweetService =

deferredTweetypie(Target(Datacenter.Local, "tweetypie-async-retry"))

val darkTrafficClient: Service[Array[Byte], Array[Byte]] = {

val thriftService =

thriftMuxClient(

"tweetypie.dark",

propagateDeadlines = false

).withRequestTimeout(100.milliseconds)

.newService("/s/tweetypie/proxy")

val transformer =

new Filter[Array[Byte], Array[Byte], ThriftClientRequest, Array[Byte]] {

override def apply(

request: Array[Byte],

service: Service[ThriftClientRequest, Array[Byte]]

): Future[Array[Byte]] =

service(new ThriftClientRequest(request, false))

}

transformer andThen thriftService

}

val geoHydrationClient: GeoduckHydration.MethodPerEndpoint = {

val mb = thriftMuxMethodBuilder("geoduck\_hydration", "/s/geo/hydration")

.withTimeoutPerRequest(100.millis)

.idempotent(maxExtraLoad = 1.percent)

ThriftMux.Client.methodPerEndpoint(

mb.servicePerEndpoint[GeoduckHydration.ServicePerEndpoint])

}

val geoHydrationLocate: GeoduckLocate = geoHydrationClient.locate

val geoReverseGeocoderClient: ReverseGeocoder.MethodPerEndpoint = {

val mb = thriftMuxMethodBuilder("geoduck\_reversegeocoder", "/s/geo/geoduck\_reversegeocoder")

.withTimeoutPerRequest(100.millis)

.idempotent(maxExtraLoad = 1.percent)

ThriftMux.Client.methodPerEndpoint(

mb.servicePerEndpoint[ReverseGeocoder.ServicePerEndpoint])

}

val geoduckGeohashLocate: GeoduckGeohashLocate = {

new GeoduckGeohashLocate(

reverseGeocoderClient = geoReverseGeocoderClient,

hydrationClient = geoHydrationClient,

classScopedStatsReceiver = statsReceiver.scope("geo\_geohash\_locate"))

}

val geoRelevance =

new Relevance$FinagleClient(

thriftMuxClientBuilder(

"geoduck\_relevance",

"/s/geo/relevance",

classOf[Relevance.MethodPerEndpoint])

.requestTimeout(100.milliseconds)

.retryPolicy(retry(timeouts = 1))

.build(),

stats = statsReceiver

)

val fanoutServiceClient =

new FanoutService$FinagleClient(

new DeferredThriftService(deferredRpcClient, Target(Datacenter.Local, "fanoutservice")),

serviceName = "fanoutservice",

stats = statsReceiver

)

val limiterService: LimiterService = {

val limiterClient =

new LimiterClientFactory(

name = "limiter",

clientId = thriftClientId,

tracer = tracer,

statsReceiver = statsReceiver,

serviceIdentifier = settings.serviceIdentifier,

opportunisticTlsLevel = OpportunisticTls.Required,

daemonize = true

)(eval("/s/limiter/limiter"))

val limiterBackend = settings.limiterBackendConfig(

LimiterBackend.fromClient(limiterClient),

backendContext("limiter")

)

LimiterService.fromBackend(

limiterBackend.incrementFeature,

limiterBackend.getFeatureUsage,

getAppId,

backendsScope.scope("limiter")

)

}

val passbirdClient =

new PassbirdService$FinagleClient(

thriftMuxClientBuilder(

"passbird",

"/s/passbird/passbird",

classOf[PassbirdService.MethodPerEndpoint])

.requestTimeout(100.milliseconds)

.retryPolicy(retry(timeouts = 1))

.build(),

serviceName = "passbird",

stats = statsReceiver

)

val escherbird: Escherbird = {

val escherbirdClient =

new TweetEntityAnnotationService$FinagleClient(

thriftMuxClientBuilder(

"escherbird",

"/s/escherbird/annotationservice",

classOf[TweetEntityAnnotationScroogeIface.MethodPerEndpoint])

.build()

)

settings.escherbirdConfig(

Escherbird.fromClient(escherbirdClient),

backendContext("escherbird")

)

}

val geoScrubEventStore: GeoScrubEventStore = {

val mhMtlsParams =

if (settings.serviceIdentifier == EmptyServiceIdentifier) NoMtlsParams

else

ManhattanKVClientMtlsParams(

serviceIdentifier = settings.serviceIdentifier,

opportunisticTls = OpportunisticTls.Required)

val mhClient =

new ManhattanKVClient(

appId = "geoduck\_scrub\_datastore",

dest = "/s/manhattan/omega.native-thrift",

mtlsParams = mhMtlsParams,

label = "mh\_omega",

Seq(Experiments.ApertureLoadBalancer)

)

GeoScrubEventStore(

mhClient,

settings.geoScrubEventStoreConfig,

backendContext("geoScrubEventStore")

)

}

val tweetEventsPublisher: EventBusPublisher[TweetEvent] =

eventBusPublisherBuilder

.streamName("tweet\_events")

.thriftStruct(TweetEvent)

.publishTimeout(500.milliseconds)

.serializeFinagleDtabs(true)

.build()

val deleteLocationDataPublisher: EventBusPublisher[DeleteLocationData] =

eventBusPublisherBuilder

.streamName("tweetypie\_delete\_location\_data\_prod")

.thriftStruct(DeleteLocationData)

// deleteLocationData is relatively rare, and publishing to

// eventbus is all that the endpoint does. This means that it

// is much more likely that we will have to make a connection,

// which has much greater latency, and also makes us more

// tolerant of slow requests, so we choose a long timeout.

.publishTimeout(2.seconds)

.build()

val retweetArchivalEventPublisher: EventBusPublisher[RetweetArchivalEvent] =

eventBusPublisherBuilder

.streamName("retweet\_archival\_events")

.thriftStruct(RetweetArchivalEvent)

.publishTimeout(500.milliseconds)

.build()

val gnipEnricherator: GnipEnricherator = {

val gnipEnricherator =

thriftMuxMethodBuilder(

"enricherator",

"/s/datadelivery-enrichments/enricherator"

)

GnipEnricherator.fromMethod(gnipEnricherator)

}

val stratoserverClient: StratoClient = Strato.client

.withMutualTls(

serviceIdentifier = settings.serviceIdentifier,

opportunisticLevel = OpportunisticTls.Required)

.withLabel("stratoserver")

.withRequestTimeout(100.milliseconds)

.build()

val configBus: ConfigBus =

ConfigBus(backendsScope.scope("config\_bus"), settings.instanceId, settings.instanceCount)

val callbackPromotedContentLogger: CallbackPromotedContentLogger = {

val publisher =

eventBusPublisherBuilder

.streamName(settings.adsLoggingClientTopicName)

.thriftStruct(AdCallbackEvent)

.publishTimeout(500.milliseconds)

.serializeFinagleDtabs(true)

.maxQueuedEvents(1000)

.kafkaDest("/s/kafka/ads-callback:kafka-tls")

.build()

val stats = backendsScope.scope("promoted\_content")

val adsLoggingClient = AdsLoggingClient(publisher, stats, "Tweetypie")

new CallbackPromotedContentLogger(adsLoggingClient, stats)

}

}

}

}