[This document is shared externally with Non-googlers. Please comment accordingly.]

gRPC Java Team

gRPC Java LoadBalancer v2

Summary
The current gRPC Java client-side load-balancing mechanism has issues in connection selection and with wait-for-ready RPCs. We propose a new design that tries to solve those issues.

Owner: zhangkun
Contributors: ejona
Status: Implemented
Created: 2016-09-29
Self Link:
go/grpc-java-lb-v2

Background

In gRPC Java, the **Channel** (ManagedChannelImpl) is a logical connection to a target specified by an URI, which represents a single server or a collection of servers that serve the same gRPC service. Channel manages connections to the servers. Each connection is managed by an object called **Transport** (ClientTransport). For each RPC, a Channel selects a Transport and creates a **Stream**—which is a low-level abstraction of the RPC—on the Transport.

Between Channel and Transport, there is a layer called **TransportSet**. A TransportSet is bound to a group of equivalent addresses (**EquivalentAddressGroup**, EAG) that usually point to the same server. A TransportSet has at most one active Transport, and zero to multiple shutdown Transports. **LoadBalancer** is a plug-in component of Channel that actually does the job of choosing an EAG and provides a Transport for each individual RPC.

The original design document is go/grpc-java-lb, which is a bit outdated in terms of naming, but is still accurate in terms of semantics. go/grpc-java-transport-management is also worth reading if you don't know how Channel works.

Known issues

Early commitment

#1600 is the manifestation of this issue in round-robin, but it actually affects all LoadBalancers.

Early commitment means an RPC is assigned to a TransportSet too soon, when it's still connecting or even in transient failure. As a result, a fail-fast RPC may fail even though there are other TransportSets that are ready, e.g., in the round-robin case. A wait-for-ready RPC may stick in the TransportSet forever if it never becomes ready, even though there are other TransportSets that are ready in the round-robin case, or a new TransportSet has been created for a newly resolved address in the pick-first case.

Early commitment is unavoidable in the current LoadBalancer API, because

1. LoadBalancer doesn't have a way to know which TransportSets are ready and which are not.
2. The assignment of an RPC to a TransportSet is irreversible. The RPC sticks with a TransportSet forever.

Wait-for-ready / fail-fast RPCs

wait-for-ready.md

Wait-for-ready semantics depend on a unified view of the connectivity state of the Channel. LoadBalancer is in the best position to have such a view. Therefore, LoadBalancer needs to implement connectivity state semantics, and Channel will just use it.

The decision on whether to fail or buffer (delay) an RPC at the point of connectivity state transition, must be done by someone that knows the current state of the whole Channel. Currently TransportSet makes the decision on the RPCs buffered in its DelayedClientTransport, but it doesn't know the whole Channel's connectivity state, thus is unable to make the right decision.

Repeated work among LoadBalancers

#2211

In current API, LoadBalancer is bound to returning a Transport for each RPC. If it's not ready yet, e.g., waiting for name resolution, it spills a DelayedClientTransport (called interim transport in the LoadBalancer API), in which the RPCs are buffered. It becomes LoadBalancer's responsibility to transfer the buffered RPCs to "real" Transports. Last but not the least, LoadBalancer API is thread-safe, that means implementation needs to have synchronization.

LoadBalancer implementations are found to repeat in two aspects:

- Implementation-wise, every single LoadBalancer would implement the same buffering and transferring logic, similar error handling and synchronization, which are non-trivial.

- Semantic-wise, this gives LoadBalancer the responsibility to make decisions on failing or buffering RPCs according to their wait-for-ready-ness and the LoadBalancer's connectivity state. Every LoadBalancer needs to make sure they comply with the wait-for-ready semantics, which is repeated work, and has thus the ability to violate the semantics, which is worse.

As a side note, having DelayedClientTransport in both the LoadBalancer and the TransportSet doesn't have any benefit and only adds complexity to the system.

Cleaning up unused connections

#2276

The server addresses may change, either from NameResolver or an external load-balancing service. Channel needs to shut down the TransportSets for the addresses that are no longer used. In the current API, updateRetainedTransports() is supposed to do the job, but it isn't mandatory, and the Channel needs to have some backup clean-up mechanism.

An intuitive solution is that Channel may keep track of the time a TransportSet was last used, and shut it down after a certain grace period. However, this isn't a one-size-fit-all grace period. In some hard-affinity cases where the client may connect to a large variety of servers, but doesn't need to keep the connection for long. In a plain round-robin, it's desirable to keep connections for longer.

The current API tries to make LoadBalancers easier to implement, by keeping the responsibility of Transport management in Channel. Now it seems more reasonable to move such responsibility onto LoadBalancer, because Channel doesn't have the knowledge to do the management efficiently.

Other minor issues

The following issues can be resolved through minor tweaks on the API.

Unable to create multiple connections for the same address which is needed by L4 load-balancing

#7957

Channel creates only one TransportSet for each EAG. This can be fixed by adding an additional field to EAG that makes it possible to have two EAGs with the same addresses but are inequal.

Affinity key is not used when assigning a pending RPC to a Transport.

#2302

This can be fixed by changing the InterimTransport interface to pass the affinity key.

Unable to add headers to RPCs

#2301

GRPCLB needs to attach a LB token to each RPC. Currently LoadBalancer API doesn't provide a way to do that. It can be solved by passing the headers to pickTransport().

Proposal

API

```
@NotThreadSafe
interface LoadBalancer {
  // All methods are run in the LoadBalancer executor, a serializing
  // executor.
  void handleResolvedAddresses(
      List<ResolvedServerInfoGroup> servers, Attributes attributes);
  void handleNameResolutionError(Status error);
  // Supersedes handleTransportReady() and handleTransportShutdown()
  void handleSubchannelState(
      Subchannel subchannel, ConnectivityStateInfo stateInfo);
  void shutdown();

  // Where the LoadBalancer implement the main routing logic
  @ThreadSafe
  interface SubchannelPicker {
    PickResult pickSubchannel(Attributes affinity, Metadata headers);
  }
  pojo PickResult {
    // A READY channel, or null
    Subchannel subchannel;
    // An error to be propagated to the application if subchannel == null
    // Or OK if there is no error.
    // Both subchannel and error being null means RPC needs to wait
    Status status;
  }

  // Implemented by Channel, used by LoadBalancer implementations.
  @ThreadSafe
  interface Helper {
    // Subchannel for making RPCs. Wraps a TransportSet
    Subchannel createSubChannel(EquivalentAddressGroup, Attributes);
    // Out-of-band channel for LoadBalancer's own RPC needs, e.g.,
    // talking to an external load-balancer. Wraps a TransportSet
    ManagedChannel createOobChannel(
        EquivalentAddressGroup eag, String authority);
    // LoadBalancer calls this whenever its connectivity state changes.
```

```
    // Channel will use the new picker against all pending RPCs
    void updatePicker(SubchannelPicker);
    // Schedule a task to be run in the Channel Executor, which serializes
    // the task with the other callback methods on LoadBalancer
    void runSerialized(Runnable task);
    // For GRPCLB which needs to resolve the address for delegation
    NameResolver.Factory getNameResolverFactory();
    String getAuthority();
  }

  // Wraps a TransportSet
  @ThreadSafe
  interface Subchannel {
    void shutdown();
    void requestConnection();
    EquivalentAddressGroup getAddresses();
    // The same Attributes passed to createSubChannel.
    // LoadBalancer can use it to attach additional information here, e.g.,
    // the shard this Subchannel belongs to.
    Attributes getAttributes();
  }

  @ThreadSafe
  interface Factory {
    LoadBalancer newLoadBalancer(Helper helper);
  }
}


pojo ConnectivityStateInfo {
  ConnectivityState state;
  // Non-OK if state == TRANSIENT_FAILURE
  Status error;
}
```

Note: most of the “interface”s will be “abstract class”es in the actual Java API for better extensibility.

Highlights

TransportSet (renamed to InternalSubchannel) gets no DelayedClientTransport

TransportSet no longer has its own DelayedClientTransport. When requested, it either returns a READY Transport, or null if it’s not READY.

There are two race conditions:

1. A TransportSet should not be picked if it's not READY, but there is a race between 1) LoadBalancer being notified about a previously READY TransportSet becoming not READY, and 2) picker picking that TransportSet. If Channel gets a null Transport from the TransportSet returned by the picker, it will keep this RPC buffered (the same effect as getting a PickResult(null, null). Eventually LoadBalancer will be notified about the state change and issue another picker which has the updated information.
2. There is also a race between 1) a Transport is picked and newStream() is called on it, and 2) its Subchannel is shutdown by LoadBalancer (e.g., because of address change). If (2) wins, the app will see a spurious error. This can be worked around by delaying shutdown of TransportSet in Subchannel.shutdown().

One DelayedClientTransport rules all

Channel keeps one DelayedClientTransport to buffer RPCs. For every new RPC, it calls SubchannelPicker.pickSubchannel() and make a decision based on the returned PickResult and the RPC's wait-for-ready-ness:

Subchannel	Error	Decision
Yes	*	Assign to Subchannel
No	Yes	Fail-fast: fail immediately. Wait-for-ready: buffer
No	No	Both: buffer

If LoadBalancer has not called Helper.updatePicker() yet, Channel will buffer RPCs.

DelayedClientTransport will have a new method -- reprocess(SubchannelPicker newPicker). Here is how it works:

- Every time pickerUpdated() is called, Channel calls newPicker() to get the latest picker, then call reprocess(SubchannelPicker newPicker), which calls newPicker.pickSubchannel() for every buffered RPC. If the decision is "buffer" again, the RPC stays in the DelayedClientTransport.
- To reduce the risk of deadlock, reprocess() doesn't hold a lock when calling pickSubchannel(). Because Channel makes the new picker current piro
    - There is a race between an RPC being processed by an older picker only to be buffered, and a newer picker being passed to Channel. The RPC may miss the newer picker, unless DelayedClientTransport assigns a version number to each picker it sees, and associate each buffered RPC with the version of the last picker that made the "buffer" decision. reprocess() will repeat the process until all buffered RPCs have seen the latest picker.

OOB channel

Currently the OOB transport is backed by a TransportSet, which has a DelayedClientTransport. This is nice because LoadBalancer only needs a simple wrapper (SingleTransportChannel) and can make OOB RPCs on it, without needing to worry about the readiness of the TransportSet.

Now that TransportSet will lose its DelayedClientTransport, the OOB channel will still wrap a TransportSet, but have its own DelayedClientTransport.

Listening to Subchannels' states

In order to make informative decisions, LoadBalancer needs to listen to the state changes of all Subchannels.

The application-oriented channel-state API on ManagedChannel falls short because it can miss brief edges to TRANSIENT_FAILURE (#28), where the LoadBalancer should fail buffered fail-fast RPCs. It is also essential for Subchannel to provide LoadBalancer the error Status in case of TRANSIENT_FAILURE. Therefore, we don't use the channel-state API, but instead introduce handleSubchannelState() that will not miss a single edge.

Listening to name resolution events

Besides Subchannel state change, LoadBalancer should also react to every name resolution event. The decision is made with both the Subchannel state and name resolution events, so the handling of them should all be synchronized, which is guaranteed by having them on the same non-thread-safe interface (i.e., LoadBalancer).

shutdown()

shutdown() would typically do final cleanups and eventually close all the Subchannels. Alternatively, Channel can shutdown the Subchannels and OOB channels for LoadBalancer when Channel goes to IDLE mode or is shutting down. However, a LoadBalancer may still want to do something, e.g., send a final message on the OOB channel, before shutting down. It's better to leave the responsibility to LoadBalancer.

This means, Channel no longer shuts down TransportSets. It's all up to LoadBalancer to do this. Channel will still keep track of the TransportSet and wait for them to be terminated before it is terminated.

When Channel is shut down, it doesn't necessarily immediately shut down NameResolver and LoadBalancer. Channel's shutdown process will be:

1. When ManagedChannelImpl.shutdown() called:
    a. Stop accepting new calls
    b. Shutdown delayed transport
    c. All other functionalities continue working.
2. Once the delayed transport is terminated, shutdown NameResolver and LoadBalancer.
3. Once all subchannels and all OOB channels have terminated, ManagedChannelImpl will terminate.

Channel Executor

Basically, the LoadBalancer needs to react to every single state change and the reactions needs to be synchronized. It makes sense for all these internal logic to be serialized, by Channel, so that LoadBalancer doesn't need to worry about synchronization.

We call out **Channel Executor**, which is a serializing executor created by Channel for the tasks that mutate the state of Channel. It's used everywhere in LoadBalancer:

1. It runs all callbacks from StateListener
2. It's OOB channel's default executor, thus its RPC callbacks and connectivity state callbacks are run from it.
3. It is exposed via Helper so that LoadBalancer can use it directly

Note that pickSubchannel() is NOT run from the Channel Executor. It's the only thing that can be run concurrently with the rest of the LoadBalancer. The recommended practice for LoadBalancer is always creating SubchannelPickers either without any mutable state or with mutable states that are only accessible within the picker, then the rest of the LoadBalancer doesn't need any locking.

Because it's TransportSet.Callback that notifies LoadBalancer about connectivity state, we will also run TransportSet.Callback from Channel Executor. As a result, Channel's idleness management is also run from Channel Executor. This will also improve code health by clearly defining TransportSet.Callback's threading model, which is currently missing.

How it solves aforementioned issues

Early commitment: Every Subchannel's state change is emitted to LoadBalancer, which can then skip failing TransportSets. Only Channel buffers RPCs, and it goes over all buffered RPC for each new picker, thus an RPC will never stuck in a particular TransportSet.

Wait-for-ready / fail-fast RPCs: LoadBalancer notifies Channel about the connectivity-state through PickResult, and it's Channel that looks at the wait-for-ready-ness and the state to make decision.

Repeated work among LoadBalancers: buffering, most error handling, and synchronization are all done inside Channel. Making the LoadBalancer API not thread-safe substantially simplifies the implementation work of LoadBalancers. They can now focus on the actual routing logic.

Cleaning up unused connections: it now becomes LoadBalancer's responsibility to shutdown unused Subchannels. It greatly reduced the work by Channel, and doesn't add much work to LoadBalancers because they usually have a better idea to tell which Subchannels are useful and which are not.

L4 load-balancing: now LoadBalancer is free to create multiple Subchannels for the same EAG.

Affinity key is not used when assigning a pending RPC to a Transport: when a pending (buffered) RPC picks a real Transport, it goes through the same picking interface as the first attempt, which has the affinity key available.

Adding headers to RPCs: the headers object is accessible at the initial and subsequent picks. LoadBalancer is free to add additional metadata to it at those points.

Caveats

Channel executor is a serializing executor on top of the application executor. If application uses a direct executor, all the methods on LoadBalancer will be run either a NameResolver thread or a transport thread, thus it may be under a lock. We need to be cautious about of the potential risk of deadlock. That said, this is an existing issue in the current API, because the handle*() methods are already run from NameResolver thread and transport thread.

Implementation steps

This is an overhaul of the LoadBalancer API and the related logic. It's hard to keep the backward-compatibility on the API, and to make incremental changes that keep the current behavior. Fortunately the API is still experimental, and we are free to change it. In order to make it easy to review, and not breaking the HEAD in the meantime, we will develop the new version with alternative class names, without overwriting the current implementation. The development can be done in the following incremental steps:

1. InternalSubchannel (**Done**: #2427)
2. DelayedClientTransport2 (**Done**: #2443)
3. LoadBalancer2 API (**Done**: #2443, #2501)
4. Channel Executor (**Done**: #2493, #2503, #2505)
5. ManagedChannelImpl2 (**Done**: #2530)
6. PickFirstLoadBalancerFactory2 and RoundRobinLoadBalancerFactory2 (**Done**: #2479)
7. GrpclbLoadBalancerFactory2 (**Done**: #2557)
8. Switch AbstractManagedChannelImplBuilder to ManagedChannelImpl2 (**Done**: #2707)
9. Delete original implementations, including obsolete code in DelayedClientTransport; Rename new implementation classes to the original names, by removing the suffix "2". (**Done** #2743)

Example implementations

PickFirstLoadBalancer

```
class PickFirstLoadBalancer extends LoadBalancer {
  final Helper helper;
  Subchannel subchannel;

  void handleResolvedAddresses(List<ResolvedServerInfoGroup> servers, … ) {
    EquivalentAddressGroup newEag = flatten(servers);
    if (!newEag.equals(subchannel.getAddresses()) {
      subchannel.shutdown();
      subchannel = helper.createSubchannel(newEag, Attributes.EMPTY);
      helper.updatePicker(new Picker(sc));
    }
```

```
  }

  void handleNameResolutionError(Status error) {
    helper.updatePicker(new Picker(subchannel, error));
  }

  void handleSubchannelState(final Subchannel sc,
      final ConnectivityStateInfo stateInfo) {
    if (sc != this.subchannel || stateInfo.state == SHUTDOWN) {
      return;
    }
    if (stateInfo.state == TRANSIENT_FAILURE) {
      helper.updatePicker(new Picker(stateInfo.error));
    } else if (stateInfo.state == READY) {
      helper.updatePicker(new Picker(sc));
    } else {
      // IDLE or CONNECTING: buffer the RPC
      if (stateInfo.state == IDLE) {
        sc.requestConnection();
      }
      helper.updatePicker(new Picker());
    }
  }

  void shutdown() {
    subchannel.shutdown();
  }

  static class Picker extends SubchannelPicker {
    final Status error;
    final Subchannel sc;
    final PickResult result = new PickResult(error, sc);
    PickResult pickSubchannel(...) {
      sc.requestConnection();
      return result;
    }
  }
}
```

RoundRobinLoadBalancer

```
class RoundRobinLoadBalancer extends LoadBalancer {
  static final Attributes.Key<Holder<ConnectivityStateInfo>> LAST_STATE;

  final Helper helper;
  HashMap<EAG, Subchannel> subchannels;
  // The full list
  ArrayList<Subchannel> roundRobinList;
```

```
// The full list excluding the TRANSIENT_FAILURE ones
ArrayList<Subchannel> activeList;

void handleResolvedAddresses(List<ResolvedServerInfoGroup> servers, … ) {
  List<EquivalentAddressGroup> latestAddressList = convert(servers);
  Set<EquivalentAddressGroup> addedAddresses = …;
  Set<Subchannel> toBeShutdown = …;
  for (EquivalentAddressGroup eag : addedAddresses) {
    Subchannel subchannel = helper.createSubchannel(eag,
        Attributes.with(LAST_STATE, new Holder());
    subchannel.requestConnection();
    subchannels.put(eag, subchannel);
  }
  for (Subchannel subchannel : toBeShutdown) {
    subchannels.remove(subchannel);
    subchannel.shutdown();
  }
  roundRobinList = updateRoundRobinList(latestAddressList);
  refreshList(null);
}

void handleNameResolutionError(Status error) {
  refreshList(activeList, error);
}

void handleSubchannelState(final Subchannel sc,
    final ConnectivityStateInfo stateInfo) {
  if (!subchannels.contains(sc)) {
    return;
  }
  if (stateInfo.state == IDLE) {
    sc.requestConnection();
  }
  sc.getAttributes().get(LAST_STATE).set(stateInfo);
  refreshList(getAggregatedError());
}

void shutdown() {
  for (Subchannel subchannel : subchannels.values()) {
    subchannel.shutdown();
  }
}

private Status getAggregatedError() {
  // if all subchannels are TRANSIENT_FAILURE, return the Status associated
  // with an arbitrary subchannel, or maybe the concatenated message
  // otherwise, return null
}
```

```
  private void refreshList(@Nullable Status error) {
    // filter() discards the Subchannels with state == TRANSIENT_ERROR
    activeList = filter(roundRobinList);
    helper.updatePicker(new Picker(activeList, error));
  }

  static class Picker extends SubchannelPicker {
    final Status error;
    final ArrayList<Subchannel> list;
    int pos;

    PickResult pickSubchannel(...) {
      if (!list.isEmpty()) {
        synchronized (this) {
          return new PickResult(list.get((pos + 1) % list.size()));
        }
      } else {
        return new PickResult(error);
      }
    }
  }
}
```

Revision history

Date	Author	Description	Reviewed by	Approved by
2016-09-30	zhangkun	First version	ejona	ejona
2016-11-15	zhangkun	Renames LoadBalancer Executor to Channel Executor and expands its usage to TransportSet.Callback		