stateful session persistence

1 Background

Session persistence is an important feature of LB. A request for the same session should be guaranteed to be routed to the same back-end Server.

In Envoy, HTTP session persistence is achieved through hash-based load balancing. **When the state of the back-end servers changes (New servers are added, existing servers are deleted, some servers' state is updated, etc.), the hash ring will be rebuilt, the session persistence is broken.** Also, session persistence cannot be used for load balancing algorithms such as Minimum Connection Count, Random, Round Robin, etc.

2 Why stateful session persistence is needed?

- Ensures that when the state of the back-end hosts changes (hosts are marked as degraded or unhealthy, new nodes are added or removed), requests belonging to existing sessions are still correctly routed to the hosts corresponding to the sessions.
- Enables Load Balancers such as Minimum Connection Count to also have session persistence capability. Session persistence is no longer exclusive to hash-based load balancing.
  Note: session persistence breaks the semantics of the original load balancing algorithm, e.g., when using the minimum connection count algorithm, new requests may be routed to a particular node X due to session persistence, instead of selecting the node with the lowest number of connections).
- ~~Improving the performance of hash-based load balancer. The consistent hashing algorithm finds the appropriate host by means of a bisearch. If stateful session persistence is used, it can effectively reduce the overhead of host search when a large number of hosts are available.~~

3 Possible solution one: Cookie

When sending a response downstream, the router filter sets a new cookie via set-cookie **whose value is the address of the upstream host**. When a new request arrives, the router filter tries to resolve the upstream host address from the cookie.
The Load Balancer gets this address through the LoadBlancerContext (requires extending the LoadBlancerContext API) and prefers this address.

Existing envoy cookie hash policy uses cookies to calculate hash value. This hash value will be used by the hash-based load balancer as the basis for selecting a host.

The biggest difference between this solution and the existing envoy cookie hash policy is that this solution directly stores the relevant information of the upstream host in the cookies, and we can directly find the corresponding host based on this information. Even if the host status changes or the hash ring is rebuilt, the result will not be affected.

😊 Simple and easy to implement. **Session persistence can be achieved across multiple Envoy instances. Envoy itself does not need to maintain additional state.**

😞 **Only based on cookies.**

~~4 Possible solution two: Cache~~

~~A hash key can be computed based on certain feature values in the request, and then the hash key and the corresponding host are inserted into the cache as one record. The corresponding host is then fetched from the cache first.~~

~~😊 More flexible. Source IP, request headers and so on can be used.~~
~~😞 Cannot achieve session persistence across Envoy instances. Envoy needs to manage additional cache.~~

~~**This scheme has its advantages, but it also has many flaws. At present, it needs further thinking, so it can be ignored for the time being.**~~

5 Implementation

The whole work can be divided into two parts.

STEP ONE

The upstream cluster and the load balancer are extended so that the load balancer can conditionally skip the load balancing algorithm and directly select a specified host.

Because upstream cluster load balancer uses LoadBalancerContext to exchange information with listener L4/L7 filter. So we can extend LoadBalancerContext to add an interface. The listener L4/L7 filter (usually router filter) can tell the load balancer what the expected host is through this interface.

```
class LoadBalancerContext {
public:
  // .......

  using ExpectedHostStatus = uint32_t;
  using ExpectedHost = std::pair<std::string, ExpectedHostStatus>;

  /**
   * Returns the primary host the load balancer shouled slected. If the
   * expected host exists and the
   * health status of the host matches the expectation, the load
   * balancer can bypass the load
   * balancing algorithm and return the corresponding host directly.
   */
  virtual absl::optional<ExpectedHost> primaryHostShouldSelected() const
PURE;
};
```

`primaryHostShouldSelected` returns the expected host string of the L4/L7 filter and the expected host status. If the host exists and the host status matches, the load balancer will return the host directly. Otherwise, the load balancer will select a new host based on the load balancing algorithm.

In the upstream cluster load balancer, in order to ensure performance, it is necessary to quickly find the corresponding host through the host string. So We can add a new interface readOnlyHostMap to PrioritySet to get a host map for fast host search.

```
class PrioritySet {
public:
  /**
   * @return const HostMapConstSharedPtr& host map that will be used to
search host by address
   * string. If the PrioritySet does not provide the fast searching of
host, this method returns
   * nullptr.
   */
  virtual const HostMapConstSharedPtr& readOnlyHostMap() const PURE;
};
```

In order to reduce the memory overhead, currently readOnlyHostMap will directly return the host map in the eds cluster. In other words, in the current implementation, stateful session persistence is only valid for the eds cluster.

When using the load balancer to select a host, the host specified by `primaryHostShouldSelected` needs to be selected first.

```
HostConstSharedPtr LoadBalancerBase::chooseHost(LoadBalancerContext*
context) {
  HostConstSharedPtr host;

  host = selectPrimaryHost(priority_set_, context);
  if (host != nullptr && !context->shouldSelectAnotherHost(*host)) {
    return host;
  }

  // ......
}
```

STEP TWO

After completing the above work, we can implement different stateful session persistence as required in the L4/L7 filter.
For HTTP, we only need to add some configuration to the HTTP router filter to control the opening and closing of stateful session persistence, and specify the implementation of stateful session persistence.

```
 message StatefulSessionSticky {
    // The expected health status of the host. Suppose the current
    // session stuck on the host X.
    // However, if the health status of host X does not meet
    // expectations, Envoy will select a new host for the current
    // request according to the load balancing algorithm.
    repeated config.core.v3.HealthStatus expected_statuses = 1
        [(validate.rules).repeated = {min_items: 1}];

    config.core.v3.TypedExtensionConfig session_state = 2
        [(validate.rules).message = {required: true}];
  }
```

(Here we are going to use `TypedExtensionConfig` to get better scalability. But there is no final decision yet.)

Take the cookie in Section 3 as an example. When a stateful session remains open and responds, the router will directly encode the upstream host string and store it in a specified cookie. When the request comes, the router will try to parse the expected upstream host string from the request cookie, and inform the upstream cluster load balancer through the primaryHostShouldSelect interface.

```
message CookieBasedSessionState {
    // The name of the cookie that will be used to obtain the encoded
    // upstream host.
    string name = 1 [(validate.rules).string = {min_len: 1}];

    // If specified, a cookie with the TTL will be generated if the
    // cookie is not present. If the TTL is present and zero, the
    // generated cookie will be a session cookie.
    google.protobuf.Duration ttl = 2;

    // The name of the path for the cookie. If no path is specified
    // here, no path will be set for the cookie.
    string path = 3;
}
```