EXTERNAL VERSION
Cache-Aware Server Push
(Client Side)

last major update: 2017-01-31, first proposed: 2016-08-08
by: zhongyi@chromium.org
status: implementation completes, experiments in Canary
canonical link: go/server-push-cancellation
tracking bug: crbug.com/232040

Update Notes

2017-01-31: A new section **Cache Access** has been added to discuss the HttpCache access from the socket layer as observations from the testing of this new feature showing that Chrome has cache splits, i.e., there are multiple HttpCache instances sharing the same network stack.

2016-09-15: The design has been updated to use delegate pattern vs factory. The latest design is elaborated in Design Ideas. And the old design is moved to appendix for reference.

Objective

The cache-aware server push project on the client side is to send RST_STREAM to cancel server push when client finds the pushed resource is already cached. It is an important step forward to integrate server push and browser cache on the client side and will release the bandwidth in a timely fashion.

As a first step to move forward, this project will use URL of the pushed request to match with cache. We will not do validations for the full request header in the first phase. Metrics will be added for this project to analyze whether it is aggressive to cancel server push using URL only. If it turns out to be true, we will then add extra validations for the full request header after URL matching in the second phase.

The project will also add Netlog tracing to inform developers of the push events.

Background

Server push has been implemented for both HTTP/2 and QUIC on the client side and but not widely deployed on the server side yet as the performance impact is not clear. Flywheel(Data Compression Proxy from Chrome) had conducted early server push experiments whose results suggest one of the directions that server push could move forward to is cache-aware server push to release the bandwidth in a timely fashion and push

more efficiently. The integration mainly has two parts: cache-awareness on the client side, which will send RST_STREAM to cancel server push when the pushed resource is already cached; and cache-awareness on the server side, which will only push resources that is believed not in client's cache using cache digest.

Cache-aware server push on the client side could potentially include two parts: cancel the push if client already has the response cached for the pushed request; and bring the response to cache if it is not cached yet. The second part has already been supported by push with preload, where server attaches a preload link element in response header when it decides to push resource to the client. This practice was proposed by W3C but is still working in progress. Starting from 2016 Q1, blink now supports link for rel=preload and chrome is able to bring pushed resource to cache by preloading. Therefore this design doc mainly addresses the push cancellation.

When the client tries to make a request, it will query the cache layer whether it has the corresponding response cached. If so, it will use the cached response. Otherwise, it will create a network transaction to send the request. With that said, in the current world, we only have the reference from cache layer to network layer. And this project will require the reference from the network layer to cache layer as we need to issue a cache lookup when we are down in the net layer.

Design Ideas

Chromium network stack supports both SPDY and QUIC. Thus there are two places where we handle the push promise. And we need a relatively generic design which works for both SPDY and QUIC to support push cancellation semantics. *Figure 1* gives an overview of the Design.

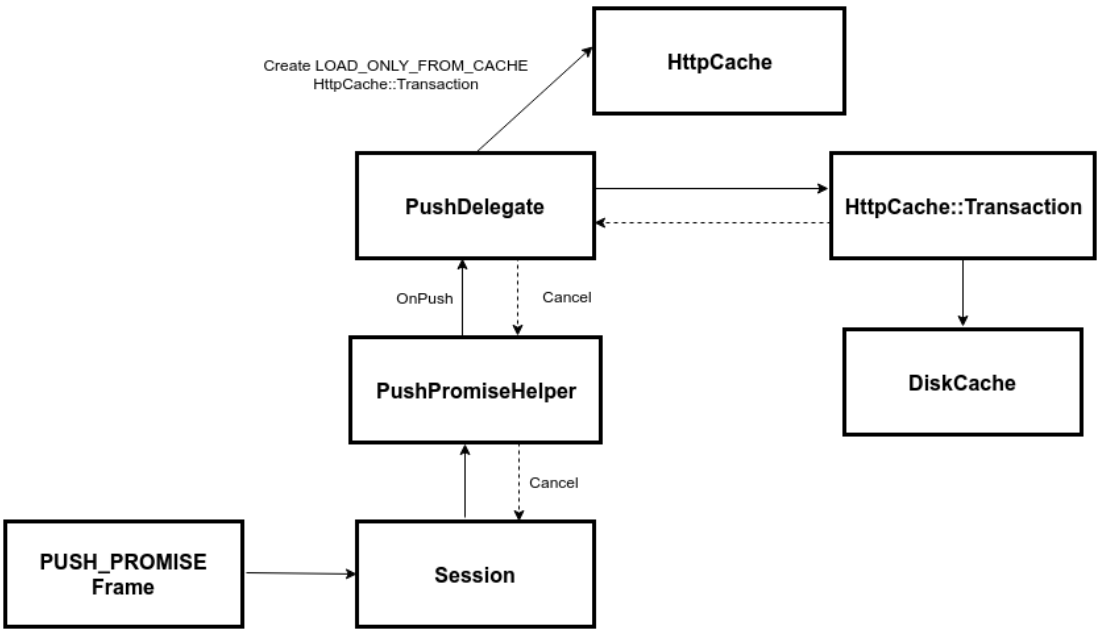

Figure 1. Cache-Aware Server Push Client Design Overview

Overview

When a PUSH_PROMISE frame is received on the client side, it will be parsed into a push promise header and handed to SpdySession/QuicChromiumClientSession. The SpdySession/QuicChromiumClientSession then performs error checkings and sends an RST_STREAM to close the session on error.

With cache-aware server push feature implemented on the client side, if no error is encountered before, SpdySession/QuicChromiumClientSession should create a PushPromiseHelper which reflects information about the pushed request. It will then notify the PushDelegate that a pushed request has been received. The PushDelegate will lookup the browser cache and see if the pushed request is cached. If the PushDelegate finds the pushed request is cached, it will instruct the PushPromiseHelper to cancel the server push which eventually will trigger the session to send an RST_STREAM if the pushed stream is not consumed yet. A closer look is provided in *Figure 2*.

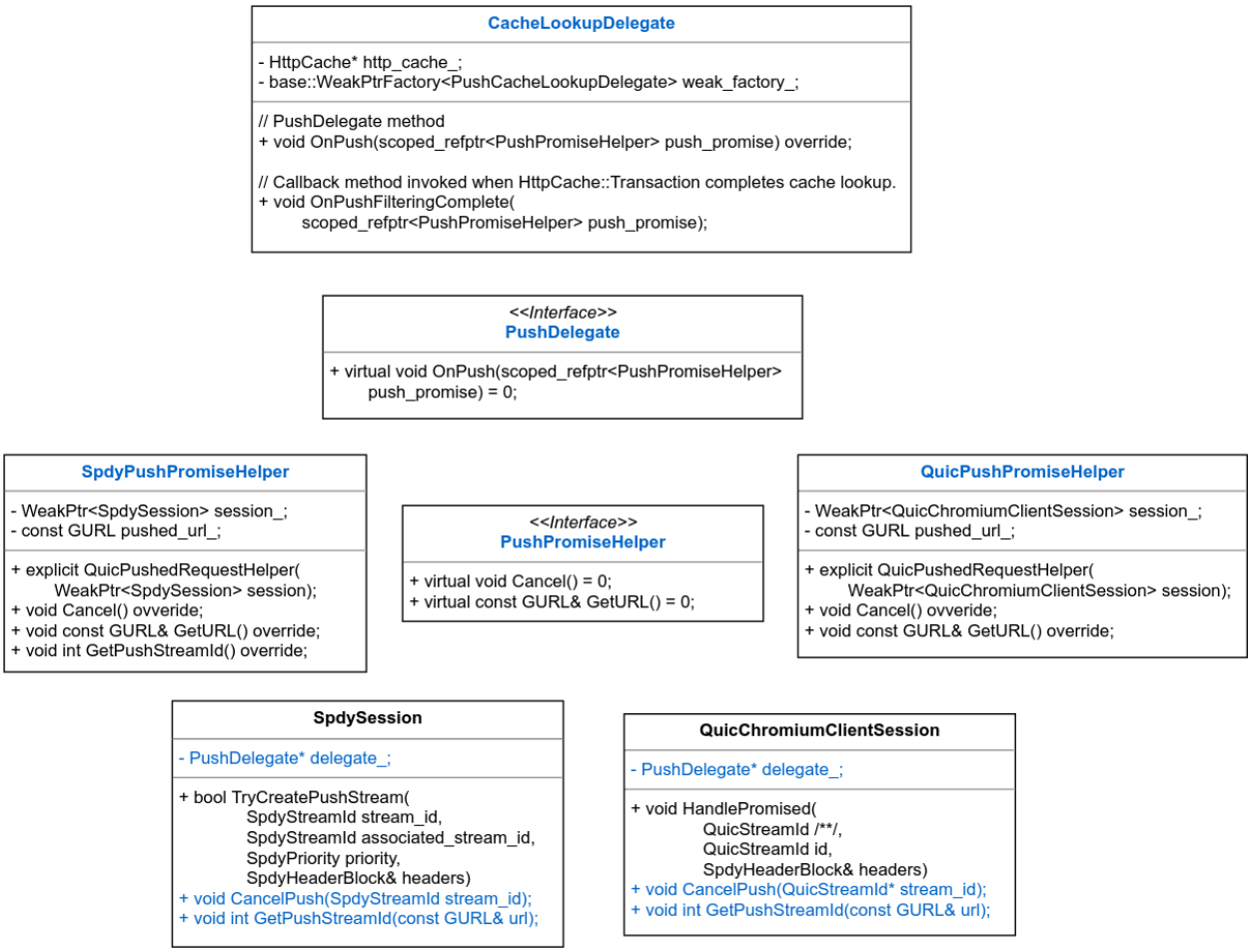

Figure 2. Class Diagram: Cache-Aware Server Push: New classes, members, methods are denoted in blue color. UML class denotes private member with "-" and public with "+".

PushDelegate

```
class PushDelegate {
 public:
  virtual void OnPush(scoped_refptr<PushPromiseHelper> push_promise) = 0;
};
```

PushDelegate provides an API to be notified by the session when a push promise has been received. It is owned by the HttpNetworkSession and references will be passed down to SpdySessionPool (which will pass the reference to SpdySession) and QuicStreamFactory (which will pass the reference to QuicChromiumClientSession).

CacheLookUpDelegate

```
class CacheLookUpDelegate : PushDelegate {
 public:
  void OnPush(scoped_refptr<PushPromiseHelper> push_promise) override;
  void OnPushFilteringComplete(scopted_refptr<PushPromiseHelper> push_promise);
 private:
  HttpCache* http_cache_;
  base::WeakPtrFactory<CacheLookupDelegate> weak_factory_;
}
```

CacheLookupDelegate implements PushDelegate and is responsible to do lookup for pushed requests in the browser cache. A pushed request is considered as cached if its URL matches an entry in the cache.

CacheLookUpDelegate::OnPush

```
void OnPush(scoped_refptr<PushPromiseHelper> push_promise) override {
  ...
  // Create a LOAD_ONLY_FROM_CACHE cache transaction to lookup.
  RequestInfo request_info;
  request_info.url = pushed_request->GetURL();
  request_info.load_flags = LOAD_ONLY_FROM_CACHE;
  int rv = transaction->Start(
      &request_info,
      base::Bind(&OnPushFilteringComplete, weak_factory_.GetWeakPtr(),
Passed(push_promise),
      BoundNetLog());
  ...
}
```

CacheLookupDelegate::OnPush should be overridden to create a LOAD_ONLY_FROM_CACHE HttpCache::Transaction to do a cache lookup for the pushed request. It will pass in a callback function to HttpCache::Transaction when the transaction starts lookup.

CacheLookUpDelegate::OnPushFilteringComplete: callback function

```
void OnPushFilteringComplete(scopted_refptr<PushPromiseHelper> push_promise) {
```

```
  ...
  // Cancel the pushed request.
  push_promise->Cancel();
}
```

The callback function OnPushFilteringComplete will be invoked when the cache transaction completes. It will instruct the PushPromiseHelper to cancel the pushed request if there is a cache hit according to the lookup.

PushPromiseHelper

```
class PushPromiseHelper {
 public:
  virtual void Cancel() = 0;
  virtual const GURL& GetURL() = 0;
};
```

PushPromiseHelper is the helper class that reflects information about the pushed request. It provides APIs to cancel server push, retrieve more detailed information for the pushed request.

SpdyPushPromiseHelper

```
class SpdyPushPromiseHelper : PushPromiseHelper {
 public:
  explicit SpdyPushPromiseHelper(WeakPtr<SpdySession> session, const GURL& url);

  void Cancel() override;
  const GURL& GetURL() override;

 private:
  WeakPtr<SpdySession> session_;
  const GURL request_url_;
};
```

SpdyPushPromiseHelper and QuicPushPromiseHelper(which is not elaborated here) should be similar. They implement PushPromiseHelper and could be nested classes in SpdySession and QuicChromiumClientSession. Both keep the URL of the pushed request as an registered id from session. A WeakPtr referring to the session is also kept so as to identify parent abandonment.

SpdyPushPromiseHelper::Cancel

```
void Cancel() override {
  if (session_) {
    SpdyStreamId stream_id = session_->GetPushStreamId(request_url_);
    if (stream_id != 0)
      session_->CancelPush(stream_id);
  }
}
```

SpdyPushPromiseHelper::Cancel(or QuicPushPromiseHelper::Cancel) provides an API for the PushDelegate to cancel the server push. The SpdyPushPromiseHelper is responsible to figure out the stream id for the push and use the stream id to request session cancel push.

SpdySession/QuicChromiumClientSession

SpdySession/QuicChromiumClientSession deals with push promise headers. It should perform error checkings on the header such as push stream id check at the very beginning. If on error, it should send RST_STREAM to the peer to cancel the push. Otherwise, it creates a PushPromiseHelper and notifies the PushDelegate that a push promise is received with the PushPromiseHelper passed to reflect information on the Push Promise.

SpdySession::CancelPush/ QuicChromiumClientSession::CancelPush

```
void CancelPush(int stream_id);
```

`CancelPush` is the method that cancels push stream with stream_id if that stream is created but not consumed yet. The caller is responsible to figure out the stream_id for which the push needs to be canceled.

Cache Access

Observations from testing showed that Chrome has multiple cache instances sharing the same networking stack(Figure 3): the main cache and the media cache. To maximize the server push cancellation success rate, ideally we want the server push lookup queries go to the cache which is more promising to have the pushed resource cached. In other words, we may not want to access the media cache if the resource should be cached in the main cache when doing server push lookup.

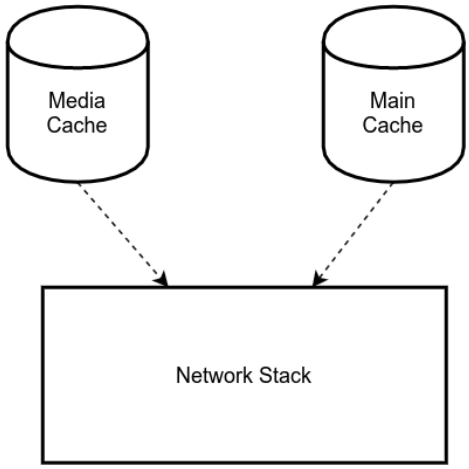

Figure 3. HttpCache(s) and Network Stack.

However, in the current world where cache splits, the network stack layer is ignorant of which cache it has access to, and the decision is made on the layer above network stack, the

ResourceDispatcherHost. Thus when server push cancellation issues a cache lookup from the network stack, there are a few options:

1. Performs cache lookup in the main cache only.
2. Performs cache lookups in both caches, cancels server push if either hits.
3. Add mechanism for network stack to add knowledge for request-cache mapping.

Accessing both caches or adding mechanisms to identify the right cache access will introduces additional costs. It might improve the server push cancel rate as a struggle to find the cached resource though. However, there's a long term goal on the networking team to get rid of the cache split by removing the media cache. The world will eventually end up with a single cache, i.e., the main cache. Rather than investing more to get the right cache access, I propose we proceed with Option 1, which only performs server push lookups in the main cache. Server push is not widely adopted, and the performance impact of server push cancellation is not understood yet. It's hard for us to say, how much gain Option 2 and 3 could give us. With Option 1, we could collect initial data with minimal effort and start analyze the performance. In the longer term, once the media cache goes away and the two cache converges, we won't miss those resource cached in the old media cache.

Metrics and Tools

To measure the improvement of this project bringing to server push, we will use UMA histograms to track server push metrics on the client side:

1. How many times server push is received;
2. How many times server push is canceled due to push-cache match;
3. How many times a real request mismatches the cache due to validation;
4. How many bytes of server push is saved which would be wasted if we do not cancel the cached server push.

With the first two metrics, we will be able to get the data how often server push is canceled. If the metrics indicates we cancel very often for server push with URL matching only while the request-cache mismatch rate is high due to validation, we might cancel too aggressively. And we will add extra header validations in PushDelegate::OnPushFilteringComplete so that we cancel less often.

To help developers understand server push events on the client side, we will also log the cancel event in Netlog tracing so as to provide more information to them and help to improve server push.

Appendix: Alternative Design Discussion

Prior to the current design, we also considered the design below which used a factory pattern.

Design Ideas

Chromium network stack supports both SPDY and QUIC. Thus there are two places where we handle the push promise. And we need a relatively generic design to support push cancellation if it is cached in both SPDY and QUIC. *Figure 1* gives an overview of the Design.

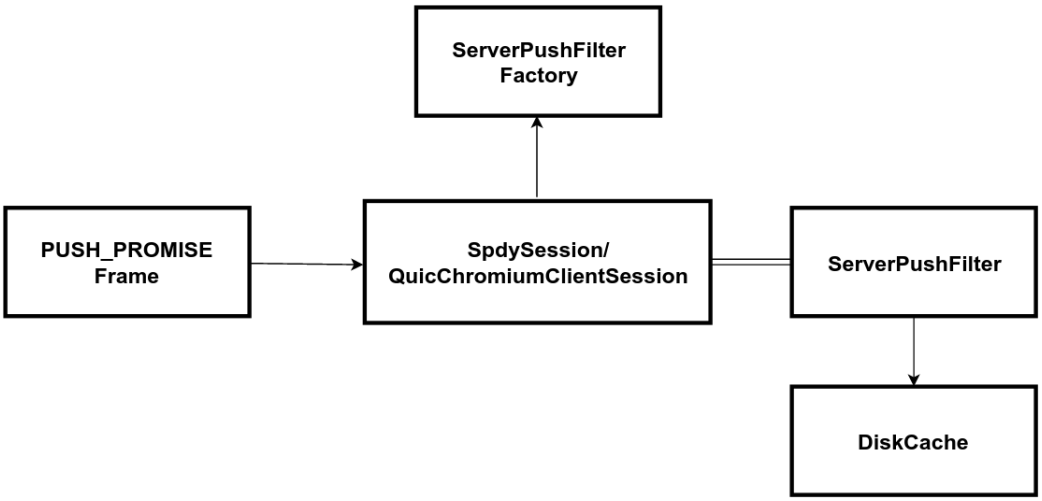

Figure 1. Cache-Aware Server Push Client Design Overview

Overview

When a PUSH_PROMISE frame is received on the client side, it will be parsed into a push promise header and handed to SpdySession/QuicChromiumClientSession. The SpdySession/QuicChromiumClientSession then performs error checkings and sends an RST_STREAM to close the session on error. With cache-aware server push feature implemented on the client side, if no error is encountered before, SpdySession/QuicChromiumClientSession should request a ServerPushFilter from the ServerPushFilterFactory. It will then call the ServerPushFilter with a callback passed over to lookup the DiskCache and see if the pushed request is cached in the browser cache. Once the ServerPushFilter completes filtering on the server push, the callback passed by SpdySession/QuicChromiumClientSession will be invoked. If the ServerPushFilter finds the pushed request is cached, in other words, filters the pushed request to be served, the callback will send an RST_STREAM to reset the server push.

ServerPushFilterFactory

```
class ServerPushFilterFactory {
 public:
  std::unique_ptr<ServerPushFilter> CreateServerPushFilter();
};
```

ServerPushFilterFactotry should provide an API to vend ServerPushFilter to SpdySession/QuicChromiumClientSession. The HttpNetworkSession will own a

ServerPushFilterFactory. It will pass down references to SpdySessionPool (which will pass the reference to SpdySession) and QuicStreamFactory (which will pass the reference to QuicChromiumClientSession).

ServerPushFilter

```
class ServerPushFilter {
 public:
  void MaybeFilterPush(const GURL& url, const Callback& callback);
};
```

A ServerPushFilter filters pushed request that is found to be cached in the browser cache. And a server push promise is considered cached if its request URL matches an entry in the cache.

ServerPushFilter should provide an API, MaybeFilterPush(), to lookup the browser cache and cancel the push if the pushed request is found cached. This method is asynchronous and thus will accept a callback passed through the API from the SpdySession/QuicChromiumClientSession. It will run the callback with a boolean and an URL to itself passed back to SpdySession/QuicChromiumClientSession to indicate whether and which push stream should be canceled.

We could either create a new class or modify the existing HttpCache::Transaction class to implement the ServerPushFilter interface. MaybeFilterPush() might evolve from the HttpCache::Transaction::DoLoop() which currently deals with the disk cache. It will be a minimal set of the states which enable us to get the backend for disk cache, open an entry, do a cache lookup for a given key, and some cleanup. This is expected to the focus of this project.

SpdySession/QuicChromiumClientSession

SpdySession/QuicChromiumClientSession deals with push promise headers. It should perform error checkings on the header such as push stream id check at the very beginning. If on error, it should send RST_STREAM to the peer to cancel the push. Otherwise, it gets an ServerPushFilter from ServerPushFilterFactory, calls the filter to do a lookup for the pushed request. In addition, it will pass in a callback to ServerPushFilter which will be invoked when ServerPushFilter completes filtering.

SpdySession and QuicChromiumClientSession handle push promise for SPDY and QUIC, respectively. More precisely, SpdySession::OnPushPromise() and QuicChromiumClientSession::HandlePromised() handle the push promise headers. Our injection of "cancel push if it's in cache" feature will be living there.

SpdySession::OnShouldCancelPush/ QuicChromiumClientSession::OnPushFilterComplete: Callback Methods

```
void OnPushFilterComplete(
       bool should_cancel, const GURL& url, ServerPushFilter* filter);
```

OnPushFilterComplete is the callback method that will be passed to ServerPushFilter::MaybeFilterPush(). It will be invoked anyway when push filter finishes

MaybeFilterPush(). The boolean is used to identify whether the pushed request is found cached. The url is used to identify which push request that has been processed by the filter. The ServerPushFilter* argument notifies SpdySession/QuicChromiumClientSession which filter has completed work and should be deleted.

Detailed Design

Setting Up

Chromium network stack has different layers. This project adds interaction issued from socket layer to cache layer, thus should be taken good care of to resolve layering issues. *Figure 2* articulates the setting up, explains how we inject the ServerPushFilter interface down to the net stack.

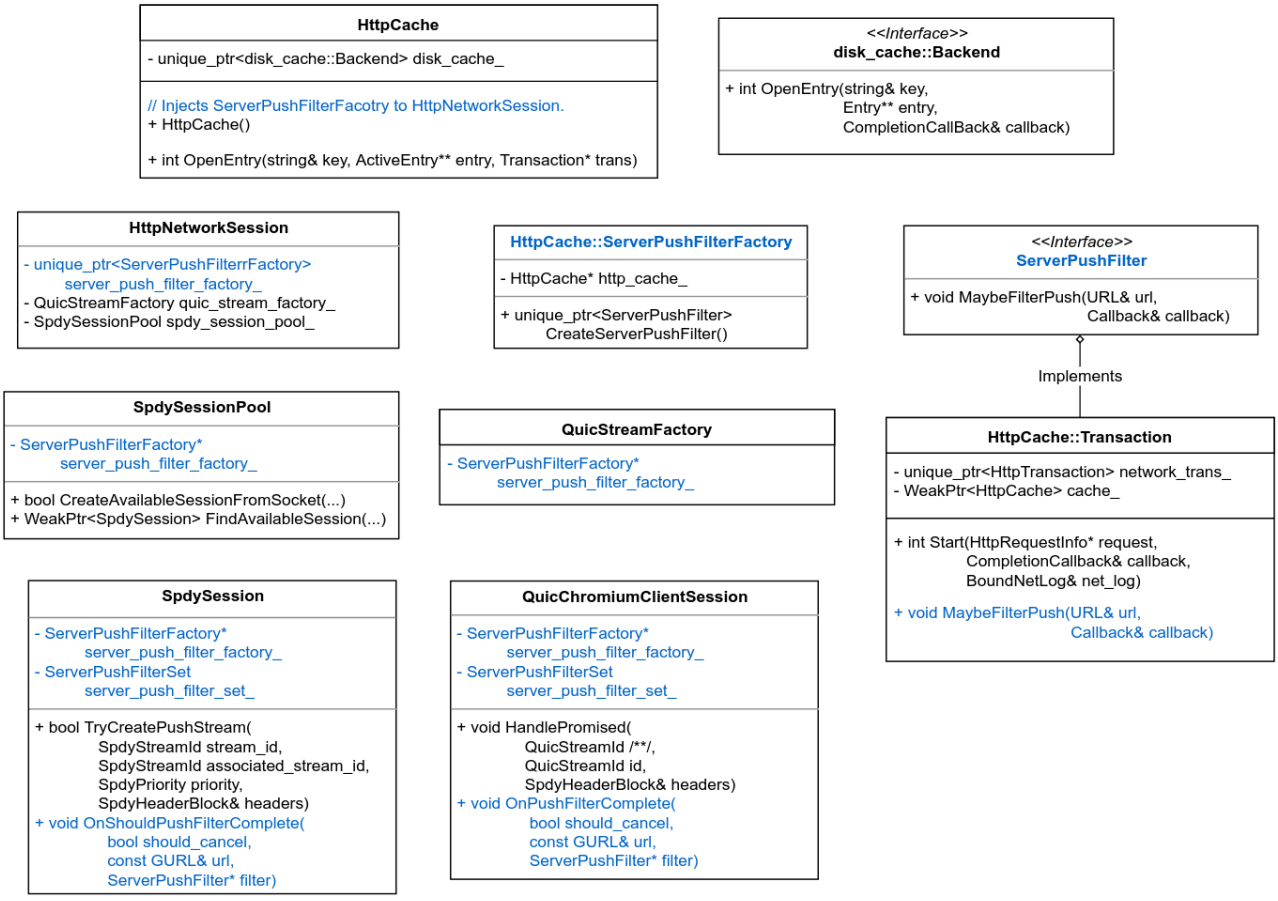

Figure 2. Injecting ServerPushFilterFactory to net layer: new classes, members, methods are denoted in blue color. UML class denotes private member with "-" and public with "+".

Upon the construction of HttpCache, we will create a ServerPushFilterFactory in HttpNetworkSession. The HttpNetworkSession owns the ServerPushFilterFactory and will pass the references of ServerPushFilterFactory to SpdySessionPool and QuicStreamFactory. The SpdySessionPool or QuicStreamFacotry when creating a SpdySession or QuicChromiumClientSession will pass the reference again to SpdySession and

QuicChromiumClientSession. The ServerPushFilterFactory will provide an API to create a server push filter for the caller and the caller could use the filter to lookup the cache.

MaybeFilterPush()

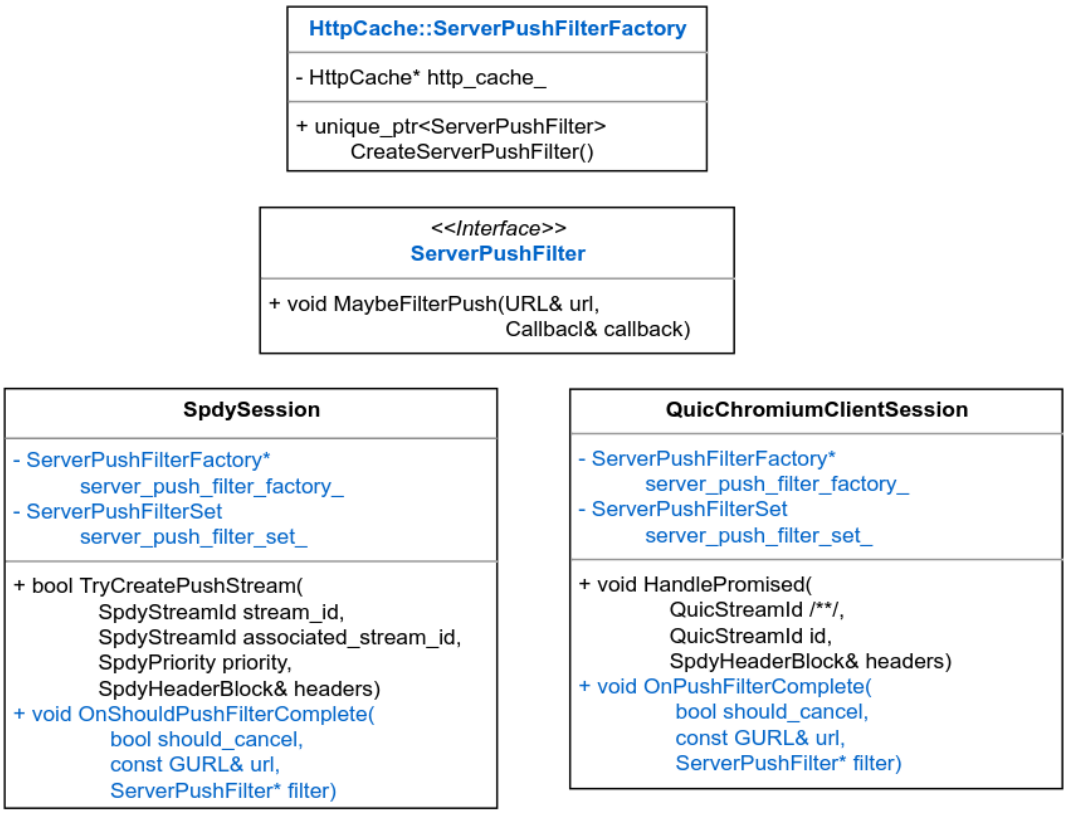

Figure 3. Class Diagram: MaybeFilterPush on the stack:: New classes, members, methods are denoted in blue color. UML class denotes private member with "-" and public with "+".

When upper layer creates a SpdySession/QuicChromiumClientSession, a reference of ServerPushFilterFactory will be passed in. The session can ask the factory to create ServerPushFilter(s) and call MaybeFilterPush when a cache lookup needs to be performed. Note the ServerPushFilter::MaybeFilterPush is asynchronous, the ServerPushFilter should outlive the stack and be deleted when the asynchronous MaybeFilterPush() completes. Therefore, a ServerPushFilterSet which manages ServerPushFilter will be created in SpdySession/QuicChromiumClientSession.