External Processing Filter

Author: gregbrail@google.com
Status: Draft
Originally Proposed: September, 2020 / Last Updated: October, 2020

Problem Statement

Although Envoy is extensible via C++, and although WebAssembly support is progressing well, there are still circumstances in which we would like to be able to have an Envoy proxy call out to an external service that can read and modify all aspects of an HTTP request or response.

For example:

- There are many types of existing systems in the world, such as API gateways, that would benefit from being based on Envoy. These proxies, which are implemented in everything from Java to Node.js, often include flexible extensibility mechanisms. These systems could be reimagined as services that are responsible for processing requests and responses and executing users' configuration, but also as services that leave the work of handling HTTP and TLS to Envoy.
- For many reasons it may not be practical to reimagine these entire systems as WebAssembly modules or Lua scripts.
- In some cases, running WebAssembly modules in a container that is invoked as a remote service may reduce the risk introduced when they run inside the Envoy proxy itself. This could be a benefit in large deployments with complex networks.
- An easy-to-use remote customization capability may make it easy for developers to quickly prototype new types of filters and other ways to extend Envoy.
- Developers have identified other reasons why such a filter could be helpful.

Background

There are a number of ways to integrate custom processing code into Envoy today. We discuss some of them at the end of this doc. However, we feel that a filter purpose-built for external processing is the most helpful.

There are a variety of patterns that such a filter should support, including:

- Examining all or part of the request or response body and determining whether to return an error based on the contents
- Examining all or part of a request and response and manipulating headers based on the

content
- Reading and then replacing the entire message body, and then changing or adding headers to reflect the new content
- Sophisticated proxy software that may choose to request or modify the message body or headers depending on the URL, method, and other parameters
- Monitoring software that normally examines traffic out of band, but which may wish to switch to a synchronous mode temporarily in order to respond to a threat

The last two points drive this design -- in some cases, the service to call is actually a policy execution engine that may use complex configuration to examine all, some, or none of the message body, and may make a different choice depending on HTTP headers and other metadata. Rather than always send the entire HTTP request and response to such a server every time, if the server is allowed to direct how it responds to each request, then it can avoid introducing latency when it is unnecessary.

Requirements and scale

The filter must:

- Allow an external service to receive the HTTP headers, body, and trailers of any HTTP request or response
- Allow the service to modify any and all HTTP headers, body, and trailers
- Allow the service to optionally receive or modify Envoy metadata
- Allow the service to control whether the body and trailers are sent to it or not after it has examined the headers, so that body data is streamed through Envoy without additional processing unless the service requests it
- Allow the service to reject the entire request and return a complete HTTP response immediately to the caller with no further processing. This can obviously be used to return an error but it may be useful in other cases, such as caching.

The filter should operate by invoking the external service via gRPC. It must use standard Envoy facilities for initiating the gRPCs, including support for both the “Envoy” and “Google” gRPC stacks. Careful usage of the gRPC stack in Envoy is important to ensure that we do not see latency spikes or other poor performance. The same pattern used by the “ext_authz” filter should be used here as well.

If the external service does not specifically request that the message body be transmitted, then the filter must allow the message body to stream through with no changes and no buffering.

Design ideas

In order to support a protocol that allows the remote service to be in control over whether it consumes the message body, the interface between Envoy and the service will be a gRPC

stream. Specifically, Envoy will initiate a bidirectional gRPC stream for each HTTP request, and keep it open until either the server closes it or until the entire request and response has been processed. Envoy will send the HTTP headers for the current request or response on the stream, then wait for additional instructions from the remote service.

The remote service can, by sending messages back “up” the stream, request the following things:

- Add, set, or remove an HTTP header or trailer.
- Request that the next chunk of the message body be sent.
- Request that the entire message body be buffered and then sent in a single chunk.
- Request that all the chunks of the message body be sent without waiting for an acknowledgement.
- Immediately return a response back to the client with no additional processing.

The remote service can also close the stream with an error, which will result in an error being returned to the HTTP client, unless a flag is set on the filter to avoid returning an error on a gRPC error.

Finally, the remote service can cleanly close the stream at any point. When that happens, the filter will continue without consulting the remote service for the rest of the filter execution.

Protocol Details

Within a single HTTP request/response (in other words, the lifecycle of a filter instance), the protocol between Envoy and the server is a request/response protocol. Each message sent on the gRPC stream to the server from Envoy must be answered with exactly one response, except in the case of asynchronous processing.

There are three basic types of messages sent from Envoy to the filter server, and each has a specific response.

- Headers
- Body chunk
- Trailers

In addition, each may be delivered in either synchronous mode, in which Envoy will wait for a response from the filter server before continuing executing the filter chain, or asynchronous mode, in which Envoy does not expect a response (any extraneous responses must be ignored). A flag on each message delivered from Envoy to the filter server will indicate whether a response is required, and a per-message timeout will ensure that Envoy can survive a slow or flaky filter.

Note that for the HTTP request, response, and trailer messages, we expect that Envoy will process these in order -- first headers, then body chunks (if any), and then trailers. However, the request and responses may be interleaved, so the protocol and the filter must be careful to

account for this -- for instance, an upstream HTTP server may return headers before Envoy has finished delivering it the entire request body.

The filter also has the concept of a “processing mode,” which defines how each of the request and response headers, body chunks, and trailers are handled. By default, they are handled as follows:

- Request and response headers are sent to the server in synchronous mode
- Body chunks are not sent
- Trailers are not sent

The processing mode is part of the filter configuration, so it is possible to change the default processing mode for a particular filter.

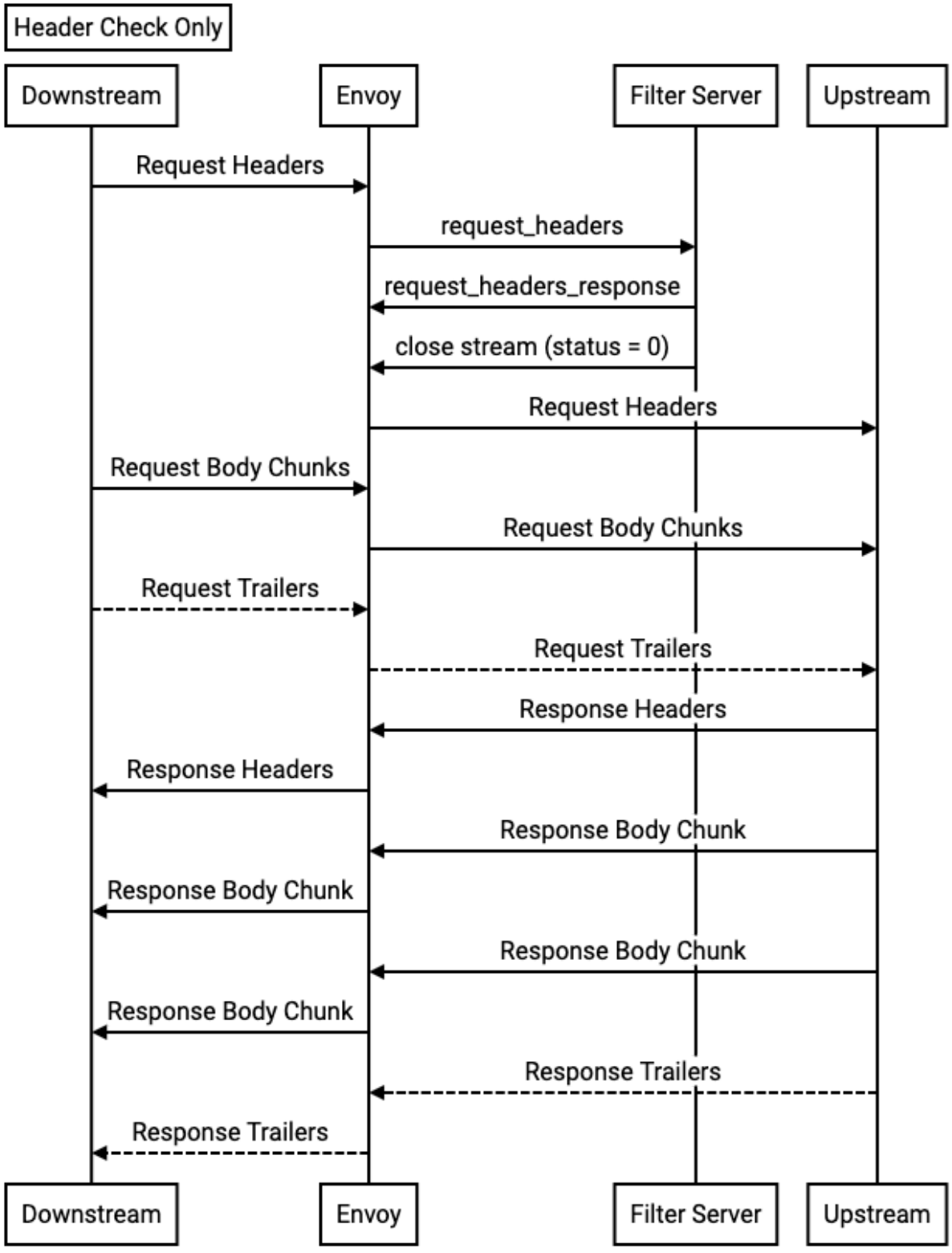
After any message, the server can respond with a new processing mode, which will affect how the rest of the message is processed. A sequence diagram that illustrated this idea can be found in the next section.

In addition, the server may also send a special message (which may be sent out of band on any stream) to ask Envoy to either permanently or temporarily change the processing mode for future requests.

Finally, a server that is “done” with a particular HTTP transaction can simply close the gRPC stream, which means that it won’t be contacted for the rest of the filter lifecycle.

Minimum Example

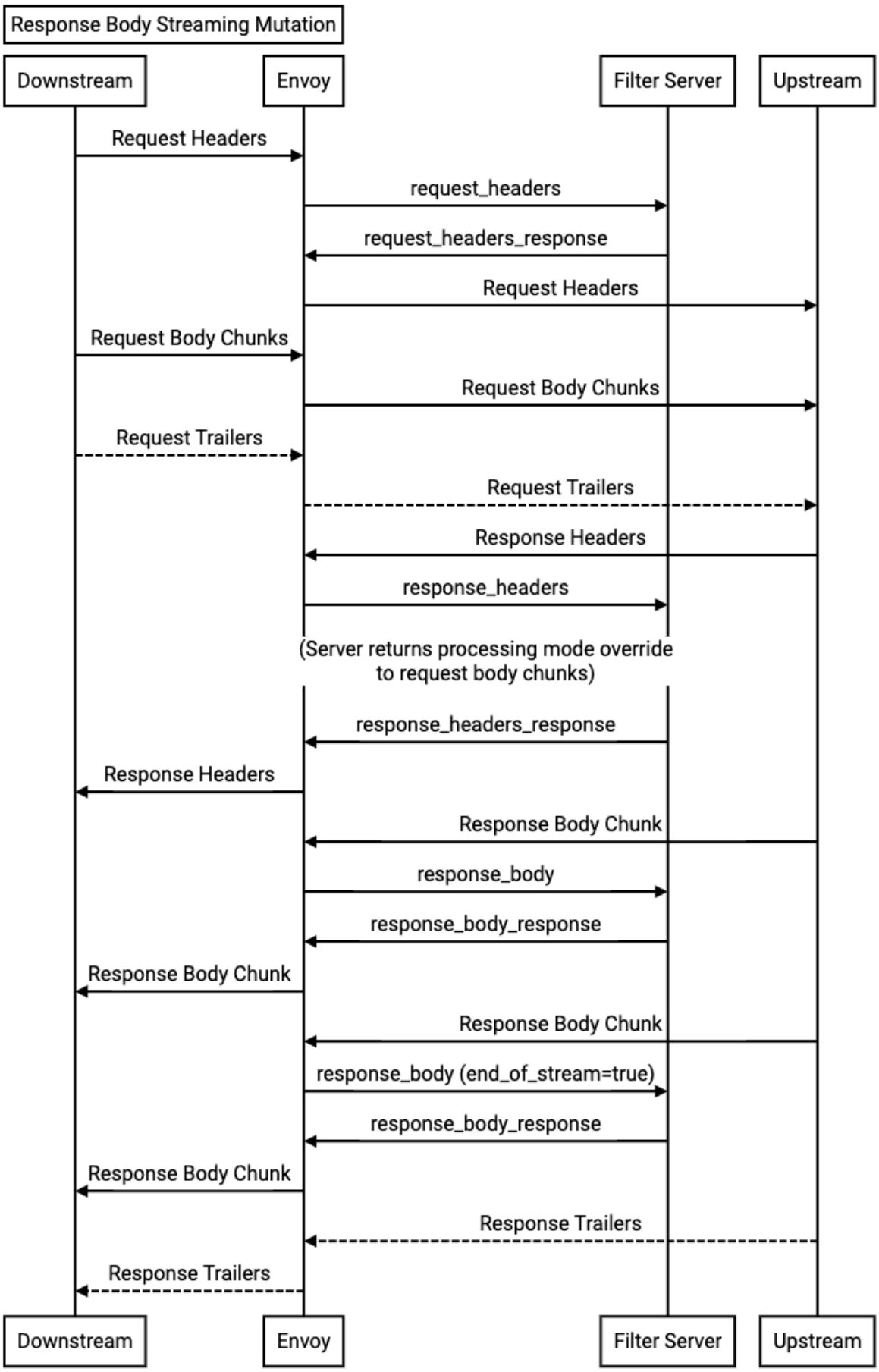
This example shows the simplest possible implementation of the filter -- a filter server receives the HTTP request headers, decides to accept the response (and can optionally modify the headers) so it closes the stream cleanly. At this point it is no longer involved in filter processing.



Example: Streaming Response Mutation

This is a more complex example that shows how a server can override the default processing mode for a particular request.

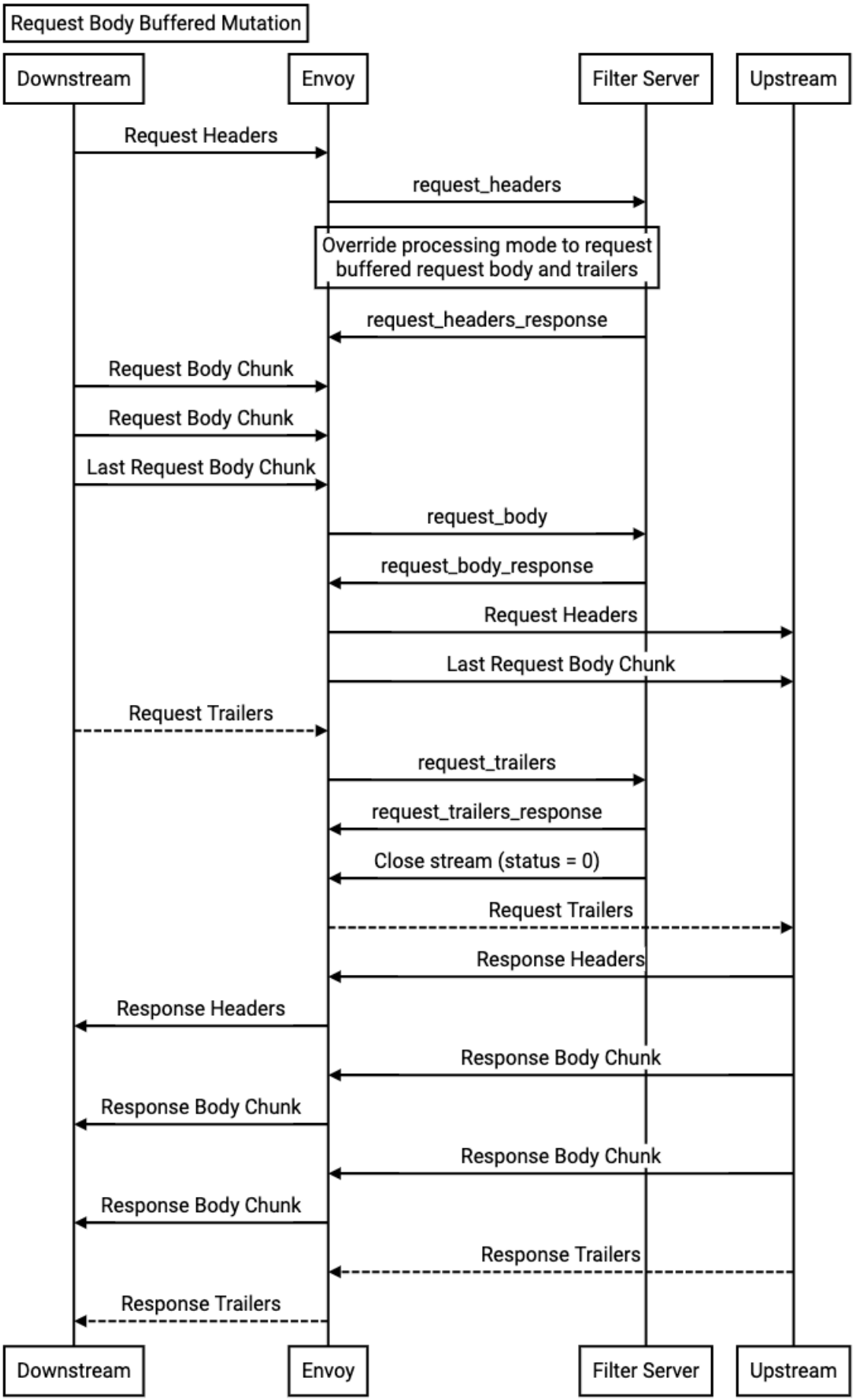
- The server looks at the request headers, but then is not interested in the request body or trailers
- The server looks at the response headers, and decides that it also wants to examine and possibly modify each chunk of the response body
- The server sends back an indication that the processing mode should be changed for the duration of the processing of this request / response.


Response Body Streaming Mutation
Downstream
Envoy
Filter Server
Upstream
Request Headers
request_headers
request_headers_response
Request Headers
Request Body Chunks
Request Body Chunks
Request Trailers
Request Trailers
Response Headers
response_headers
(Server returns processing mode override
to request body chunks)
response_headers_response
Response Headers
Response Body Chunk
response_body
response_body_response
Response Body Chunk
Response Body Chunk
response_body (end_of_stream=true)
response_body_response
Response Body Chunk
Response Trailers
Response Trailers
Downstream
Envoy
Filter Server
Upstream

Example: Buffered Request Mediation

As another example, this is the sequence diagram for a server that wishes to do the following:

- Examine the request headers
- Read and replace the entire body
- Modify request headers after replacing the body
- But is not interested in the response


Request Body Buffered Mutation
Downstream
Envoy
Filter Server
Upstream
Request Headers
request_headers
Override processing mode to request buffered request body and trailers
request_headers_response
Request Body Chunk
Request Body Chunk
Last Request Body Chunk
request_body
request_body_response
Request Headers
Last Request Body Chunk
Request Trailers
request_trailers
request_trailers_response
Close stream (status = 0)
Request Trailers
Response Headers
Response Headers
Response Body Chunk
Response Body Chunk
Response Body Chunk
Response Body Chunk
Response Trailers
Response Trailers
Downstream
Envoy
Filter Server
Upstream

Protocol Definition

Specific .proto definitions may be found in this PR:

https://github.com/envoyproxy/envoy/pull/13304

Advanced Features

These features are important to plan for but are not necessarily included in the first version of the filter.

Per-Route Configuration

Envoy supports a configuration model that allows a filter to have different configuration for different routes, selected by URL or other parameters. In the case of this filter, this would allow, for instance, for a configuration that only invokes this filter for certain requests, or which selects different servers for different routes. This is a straightforward thing to add to the configuration.

Caching

High-volume uses of this service may benefit if responses from the external filter can be cached. For instance, a service that always requests the same changes based on a header or on the URL.

In order to maintain flexibility, a caching strategy for this filter should allow the server to be in control over how the cache entry is selected, and how long it remains in the cache.

There is not one “cache key” that will work for every use case. However, in order for caching to work, the proxy must calculate the cache key without contacting the remote server. So, the cache key must be specified in the filter configuration.

For example, we might add something like the following to the filter configuration:

```
cache_key:
   -  :method
   -  :path
   -  x-special-thing
max_timeout: 300s
max_body_size: 1M
cache_size: 100MB
```

This would result in the following:

- On each request, the cache key is constructed by concatenating the values of the request method, the request path, and the value of the “x-special-thing” header

- Cache entries are stored for a maximum of 300 seconds -- server responses should be able to override this in their response, or mark a response as “not cacheable,” but nothing will be retained longer than this time
- The total size of the response from the server will not be cached if it exceeds one megabyte
- The filter will use up to 100 MB in memory to cache responses

When a response is indeed cached, the proxy must cache everything that came back from the server for that request, and replay it. That means caching all the instructions to modify the request and response headers, trailers, and body. (In practice, this probably means serializing all the protobuf messages that came back from the server and caching them.)

However, caching data in memory in an Envoy server may be potentially destabilizing unless memory usage is carefully controlled. An alternative design would be to leave caching outside Envoy entirely and deploy an external processing filter that can cache in a more flexible way and delegate to additional processing servers.

HTTP/2 Metadata

HTTP/2 allows metadata frames to be transmitted at many points in an HTTP stream, and Envoy allows limited processing of these frames. When we encounter a use for this in an external processing filter, we’ll extend the protocol to support it.

Alternatives considered

This mechanism could be simplified by avoiding the gRPC stream and sending the entire message body in one chunk. This would be easier to implement on the server side. However, that would mean that either the filter always sends the buffered HTTP request body with each request, or it doesn’t. That could be configurable, or even set up using Envoy per-route configuration, but at some point a sophisticated mediation service might want to choose whether it wants the body at runtime based on information from the request.

In addition, there are a few additional ways to solve these basic problems. This proposal offers a more flexible interface that should allow for more sophisticated use cases.

Proxy chaining, or an “Envoy sandwich,” is a way to incorporate external proxy logic inside a flow. For instance, a different type of proxy can be deployed upstream from an Envoy proxy, with another Envoy proxy behind it for outgoing traffic. However, the “middle” proxy in the sandwich must support all desired protocols. If, for instance, the middle proxy does not properly support HTTP/2 streaming, or gRPC, or WebSockets, then the whole stack does not work. Furthermore, Envoy is typically better and faster at handling HTTP and TLS than other bits of software, so this approach adds the most overhead.

The external authorization filter is a built-in Envoy filter that has evolved to allow an external service to accept or reject an HTTP or network request for a variety of reasons, and can even

inspect the message body if desired. We could extend this filter to allow it to run on the response path, and to allow the filter to modify the message body. However, at some point we have stretched this filter way beyond its original use case and built something with too many configuration options.

WebAssembly support is a very promising capability as it allows any developer to plug in all kinds of processing to Envoy. Since WASM modules can make outgoing HTTP and gRPC requests, we can build this filter in WASM today (and have indeed done this partially while prototyping). However, it is a lot of machinery to add if all we want to do is invoke an external service. A native filter will be simpler to use and more efficient. We feel that WASM will be incredibly useful for things like custom message validation, message transformation, additional protocol support, and integrating with custom security schemes.

In addition, in some Envoy deployments, dynamic code-loading environments such as WASM and Lua will not be allowed or will be discouraged for security and stability reasons -- this filter provides an alternative that would allow them to run offboard from the Envoy proxy itself.

Closed Questions

Non-gRPC HTTP support?

No -- gRPC only.

This proposal suggests a gRPC stream, which is much more flexible. Such an architecture is difficult to support in HTTP unless every server-side implementation can support something like HTTP/2 bidirectional streaming, which is more complex in most environments than supporting gRPC.

Chunks or the whole message?

Both. The interface makes it possible for the server to request individual chunks or a buffered body, which should be sufficient for most use cases.

Network filter support?

No -- HTTP only for now.

This proposal suggests an HTTP filter because most use cases we are familiar with are HTTP-based. Such an approach could also be taken for a pure network filter, which would send chunks of data to the server and give it the opportunity to modify them. However, this seems like a very different interface that perhaps should be implemented using a different service.