Федеральное государственное автономное образовательное учреждение
высшего образования
«Московский физико-технический институт
(национальный исследовательский университет)»
Физтех-школа Прикладной Математики и Информатики
Кафедра корпоративных информационных систем

Направление подготовки / специальность: 01.03.02 Прикладная математика и информатика

Направленность (профиль) подготовки: Прикладная математика и компьютерные науки

РАЗРАБОТКА ПОДСИСТЕМЫ, ОТВЕЧАЮЩЕЙ ЗА ИЗМЕНЕНИЯ КОНФИГУРАЦИИ СЕРВИСОВ ДЛЯ SERVICE MESH

(бакалаврская работа)

Студент:

Сагитов Искандер Рустамович

_____

(подпись студента)

Научный руководитель:

Шмарион Максим Юрьевич

_____

(подпись научного руководителя)

Научный консультант:

Буланкин Данил Андреевич

_____

(подпись научного консультанта)

Москва 2022

Аннотация

Данная работа посвящена иследованию существующих решений Service Mesh и проектированию архитектуры для собственной реализации данного приложения. Также в работе реализуется часть спроектированного Service Mesh такие как набора инструментов для упрощенного создания прокси для управления всем траффиком в сети, где будет работать Service Mesh и подсистема Controller, отвечающая за доставление необходимых настроек до этих прокси.

Содержание

1. Введение

1.1. Проблемы микросервисной архитектуры

Компании вс ечаще задумываются над переносом своего монолитного приложения на микросервисную архитектуру. Также многие компании используют облака для развертывания своих микросервисов. Как известно, монолитные приложения сложны при програмировании и у них длинный релизный цикл.

В то время как микросервисная архитектура способствует ускорению релизного цикла и ускорению разработки, оно также добавляет некие новые сложности, например такие, как описанные ниже.

- Безопасность – в отличие от монолитного приложения, где одни функции просто вызывают другие, в микросервисной архитектуре это обычно трансформируется в запросы по сети из одного сервиса в другой с необходимостью отслеживать всю эту коммуникацию.

- Отказоустойчивость – значительно увеличивается граф зависимостей между сервисами, что ведет к замедлению работы программы из-за большого количества последовательных запросов по сети и также повышает вероятность каскадного отказа сервисов из-за отказа всего лишь одного.

- Правила межсервисного взаимодействия – некоторые сервисы могут становиться узкими местами при выполнениях функций других сервисов. Необходимо настраивать квоты и определять политики, которые описывают какие сервисы к каким могут совершать вызовы и как много таких вызовов может совершаться.

- Наблюдаемость – по сравнению с монолитными приложениями, где достаточно файлов лога, в микросервисной архитектуре один полноцен-

ный запрос может проходить через несколько приложений. Получается, что ошибка может возникнуть в любом месте этой цепочки и разработчикам необходимо не только хранить файлы лога, но и иметь возможность следить за сетевыми показателями сервисов, процессом прохождения через цепочку сервисов отдельного запроса и за топологией всего кластера.

1.2. Service Mesh

Service Mesh – отдельный уровень в микросервисной архитектуре, который абстрагирует в себе межсетевое взаимодействие отдельных сервисов и позволяет единообразным образом, независимо от кода отдельных приложений, решать следующие проблемы:

- Обеспечение безопасности – традиционная сетевая безопасность основана на обеспечении надежного периметра для предотвращения доступа злоумышленника внутрь. Service Mesh же, в свою очередь, предоставляет идентификатор для каждого сервиса и уже по этим идентификаторам происходит аутентификация и авторизация, обеспечивается контроль того, как различные сервисы могут общаться между собой. Также Service Mesh позволяет разработчикам использовать различные протоколы, например mTLS для обеспечения зашифрованного трафика между сервисами и предотвращения атак men-in-the-middle.

- Наблюдаемость – из-за особенностей реализации Service Mesh, весь сетевой трафик внутри сети протекает через него. Далее, Service Mesh генерирует необходимые типы телеметрии: метрики, трейсы, логи доступов и складывает в хранилище.

- Балансировка нагрузки – обычно у каждого сервиса запущено несколько инстансов и общая нагрузка распределяется между ними. В качестве

способа балансирования часто используют алгоритм round-robin(равномерное распределение запросов к каждому инстансу) или взвешенный round-robin(то же самое, что и round-robin, но у каждого инстанса есть свой вес) или же случайный что не учитывает, например, того, что разные запросы могут быть разной сложности. Service Mesh же, имея у себя все необходимые метрики, позволяет использовать более умные алгоритмы балансировки нагрузки.

- Поиск сервисов – часто у сервисов меняется количество инстансов или сами инстансы меняют физические хосты. Service Mesh может обрабатывать эти ситуации и самостоятельно перенаправлять запросы к живым, в текущий момент, инстансам используя определенное ранее правило балансировки.

Обычно архитектура Service Mesh делится на 2 концептуальные части:

- Data plane – обычно это набор проксей, которые перехватывают весь входящий и исходящий трафик конкретного сервиса и перенаправляет его в соответствии со своей конфигурацией. Также эти прокси собирают телеметрию и отправляют ее в Service Mesh.
- Control plane – набор приложений в Service Mesh, которые отслеживают состояния всех сервисов и в соответствии с этим переконфигурируют прокси.

1.3. Существующие решения

Рассмотрим самые популярные реализации Service Mesh.

- Istio – open source решение от Google, которое в качестве прокси использует Envoy. Istio имеет широкие возможности конфигурации, но потребляет достаточно большое количество ресурсов. Изначально Istio

спроектировано для работы с кластером Kubernetes, но его также можно использовать и с пользовательской системой развертывания. В итоге, если суметь сконфигурировать Istio, то получиться достаточно универсальный инструмент, который решает перечисленные выше проблемы, но тут ощущается недостаток документации и статей о том, как сделать что-то.

- Linkerd – open source решение, которое в качестве Data Plane использует собственную легковесную прокси, из-за чего накладные расходы от Service Mesh в разы меньше. Linkerd предоставляет меньше возможностей конфигурации, что делает эту самую конфигурацию более простой.

Получается, что Linkerd подходит для тех, кому нужен просто быстрый Service mesh который практически не надо конфигурировать, в случае же когда от Service Mesh требуются специфические настройки, то следует использовать Istio.

Мы же, в свою очередь, хотим настроить Service Mesh для 1С Облака, в котором имеется своя собственная система развертывания и где не используется Kubernetes. Из-за этого Linkerd нам точно не подходит, потому что он не используется вне Kubernetes. Istio нам также не подходит так как, хоть его и можно использовать вне Kubernetes, но для этого не хватает документации.

В итоге мы пришли к тому, что нам необходимо разработать собственное решение данной задачи. Мы остановились на том, что на данный момент из всей функциональности Service Mesh нам необходимо реализовать только следующий набор:

- Мониторинг
- Балансировка
- Автоматическое перенаправление запросов

1.4. Цель и задачи

Целью работы является реализации компоненты Service Mesh, которая отвечает за переконфигурирование sidecar-proxy. В качестве прокси мы решили использовать Envoy так как он позволяет себя динамически переконфигурировать.

- исследовать текущие решения и спроектировать архитектуру Service Mesh
- определить, какую прокси следует использовать в Data Plane
- реализовать набор инструментов для упрощения создания стартовых конфигураций прокси
- реализовать приложение Controller, которое ответственно за доставку новых конфигураций для этих прокси
- протестировать приложение Controller и инструменты для изначального начального запуска прокси

В результате работы были получены следующие результаты:

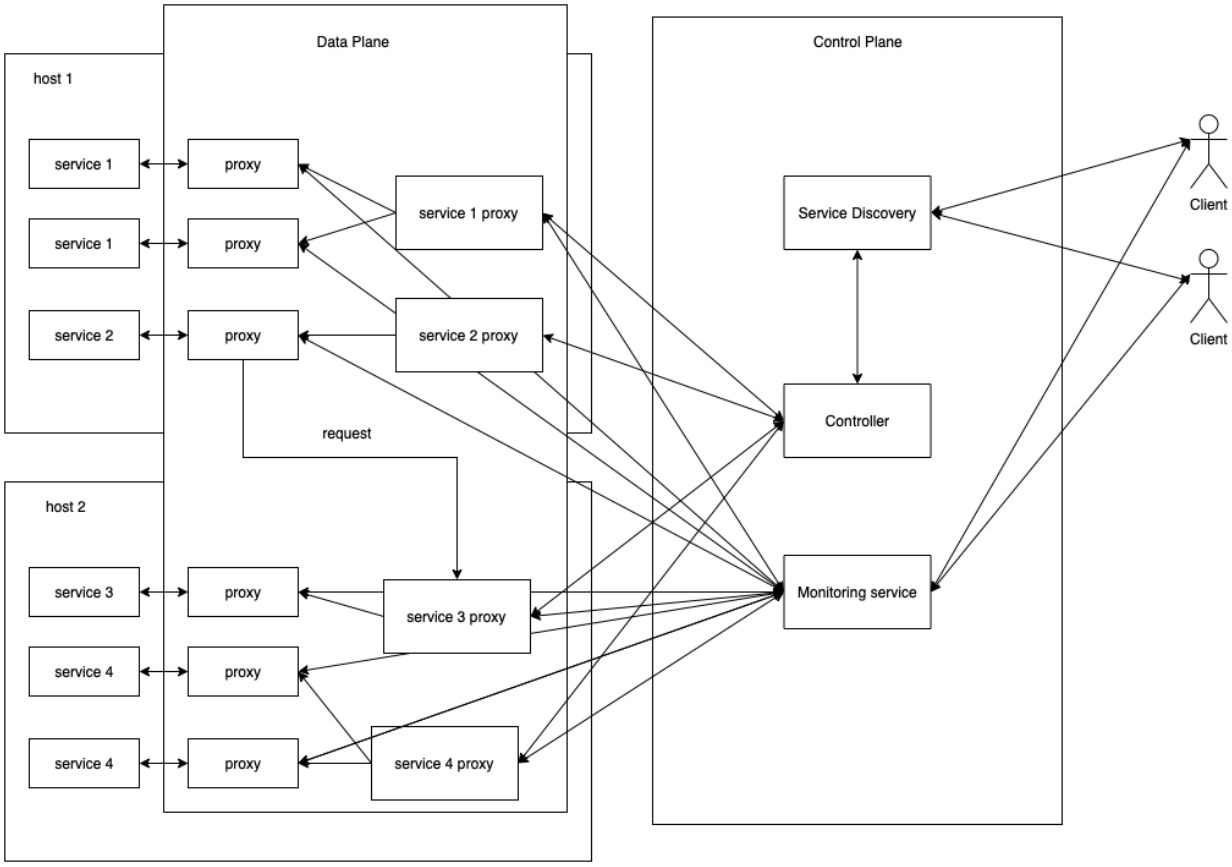
- создана архитектура Service Mesh
- выбран Envoy в качестве прокси для Data Plane
- написан bash-скрипт для создания начальной конфигурации Envoy
- реализовано приложение Controller
- реализовано отдельное приложение для тестирования приложения Controller и bash-скрипта

2. Разработанное решение

2.1. Архитектура Service Mesh

Также, как и в большинстве реализаций, мы разделили разделили все компоненты на 2 концептуальные части:

- Data Plane – для каждого сервиса, у которого есть множество инстансов, будет стоять одна прокси на вход, то есть все поступающие запросы будут проходить через эту прокси и она будет решать, в какой инстанс его направить. Также будет одна прокси на выход, то есть если инстансу необходимо воспользоваться другим сервисом, то он направляет запрос в эту прокси и уже она, зная, где находится прокси на вход у этого сервиса, перенаправляет запрос туда.
- Control Plane – мы решили разделить эту часть на несколько приложений:
    1. Service Discovery – сервис, который знает, на каком хосте и на каком порту находится каждый инстанс каждого приложения и который также умеет получать запросы о том, что какой-то инстанс поднялся, или наоборот - пропал.
    2. Controller – сервис, который от Service Discovery получает более конкретный запрос, что в связи с изменением конфигурации кластера следуют переконфигурировать следующие инстансы Envoy и то, как именно их стоит переконфигурировать
    3. Monitoring service – сервис для отслеживания состояния Envoy
    4. Balancing service – сервис для настройки балансировки между инстансами отдельного сервиса



Как видно на схеме, клиенты обращаются к Service Discovery с верхнеуровневыми запросами, что у какого-то сервиса надо добавить, или наоборот, убрать инстанс. Далее, Service Discovery обрабатывает этот запрос и в процессе создает пачку запросов к Controller, что для некоторых proxy надо поменять конфигурацию, например, в случае удаления инстанса сначала надо убрать из под нагрузки, а потом выключать, а в случае добавления сервиса следует делать наоборот - сначала поднять инстанс и уже потом переконфигурировать прокси, чтобы на этот инстанс тоже шли запросы.

Сам же трафик в сети ходит следующим образом:

1. Выходящие запросы из инстансов сервисов всегда направляются в прокси этих инстансов.
2. Эти прокси перенаправляют запросы к нужным прокси сервисов.
3. Прокси сервиса перенаправляет запрос в прокси нужного инстанса.
4. Прокси инстанса перенаправляет запрос к инстансу.

Также пользователь может получать различные метрики о состоянии кластера. Для этого ему достаточно делать запросы к компоненте Monitoring Service, которая, в свою очередь опрашивает все прокси связанные с нужным сервисом. А прокси имеют все необходимые метрики так как через них проходят все запросы.

2.2. Выбор proxy для Data Plane

Изначально мы рассматривали 3 варианта:

- Nginx [**nginx**] – высокопроизводительный веб-сервер, поддерживающий hot reload(применение изменений без перезагрузки сервера), но его также можно использовать как reverse-proxy, в каччестве кеша или балансировщика нагрузки. Nginx использует асинхронную событийно-ориентированную модельобработки запросов, то есть на каждый запрос не создается отдельный поток выполнения, а все запросы обрабатываются в одном потоке, но не блокирую друг друга. К сожалению, open-source версия nginx имеет ряд ограничений, например hot reload не может выполняться при большой нагрузке и теряет часть пакетов и также отсутствует health checking(периодические проверки инстансов на то, что они могут принимать запросы).

- HAProxy(High Availability Proxy) [**HAProxy**] – популярное программное обеспечение с открытым исходным кодом для балансировки нагрузки HTTP/TCP траффика или выступающее в качестве прокси-сервера. HAProxy также как и nginx использует один процесс и событийно-ориентирова модель обработки запросов, что позволяет при использовании достаточно маалого объема памяти обрабатывать одновременно множество запросов.

- Envoy [**Envoy**] – самый молодой(появился в 2016 году) прокси из этих

трех, но который уже используется в таких компаниях как Lyft, Google, Apple и других. Envoy изначально был спроектирован для работы с микросервисами так как имеет такие функции как hot reload, наблюдаемость, отказоустойчивость и продвинутая балансировка нагрузки. Envoy также может использоваться в ситуации частых изменений конфигураций приложений так как предоставляет не только статическую конфигурацию, но и динамическую, например, с помощью gRPC/protobuf.

В итоге мы решили остановиться на Envoy из-за того, что он поддерживает динамическую переконфигурацию то есть можно поменять конфигурацию без перезагрузки.

Envoy внутри себя определяет несколько ресурсов конфигурации, но нам интересны только следующие:

1. listener – адрес и порт, на которых слушает Envoy. С помощью listener'ов Envoy получает соединения и запросы. Envoy может иметь как и более одного listener'а, также как и один listener может слушать более чем на одном IP-адресе.

2. route – наборы правил, связанные с listener'ами, которые сопоставляют виртуальные хосты с кластерами. В этих правилах можно опираться на метаданные запроса, такие как заголовки или URI и в зависимости от этого определять, на какой кластер его следует направить.

3. cluster – набор вышестоящих хостов, которые обрабатывают получаемые запросы. Также cluster'ы являются теми ресурсами, в которых можно прописывать правила балансировки, таймауты подключения и автоматические выключения.

Envoy поддерживает статическую конфигурацию, то есть когда изначально в .yaml файле прописаны все настройки, и динамическую, то есть когда в .yaml файле прописан способ получения конфигурации.

В свою очередь динамическая конфигурация бывает трех типов:

1. Подписка на файл – самый простой способ получения динамической конфигурации это поместить ее по известному локальному пути и прописать этот путь в конфигурацонный файл. В таком случае Envoy, в зависимости от операционной системы, использует inotify или kqueue для отслеживания изменений в файле. Также в этом случае нет механизма решать конфликты при ошибках обновления конфигурации, поэтому при возникновении ошибок при. обновлении Envoy будет просто откатываться на предыдущую версию.

2. gRPC stream – примерно как предыдущий вариант, но вместо отправления запросов Management Server'у поддерживается gRpc stream с ним.

3. REST – также возможен способ получения обновлений через периодические запросы к Management Server'у с использованием протокола REST, но в этом случае постоянное соединение с сервером не поддерживается. Ожидается, что в любой момент времени существует не более одного запроса к Management Server'у.

В нашей ситуации конфигурация Envoy, который выступает в роли sidecar'а у инстанса сервиса будет состоять из следующих частей:

- Для каждого другого сервиса, который используется этим инстансом мы добавляем пару listener – cluster, чтобы перенаправлять запрос.

- Одна пара listener – cluster для слушания и перенаправления запросов, которые должны передаваться конкретно этому инстансу.

- Также требуется добавить по одному route для каждой пары listener – cluster, описанных выше из-за особенностей Envoy.

Конфигурация Envoy, который выступает в роли прокси для сервиса будет состоять из следующих частей:

- Один cluster для настройки правил в каком случае какому инстансу следует перенаправть конкретный запрос.
- Один listener для слушания запросов, которые должны передаваться конкретно этому сервису.
- Также требуется добавить один route для вышеописанной пары listener – cluster, из-за особенностей Envoy.

Понятно, что статическая конфигурация и динамическая с подпиской на файл нас не устраивали. Разницы между оставшимися вариантами практически нет, но вариант с использованием REST'а больше не поддерживается в третьей версии Envoy'а, так что я решил использовать динамическую конфигурацию с использованием gRPC для получения новой версии.

2.3. Конфигурация Envoy

На более глубоком уровне разделяется 4 варианта транспортного протокола, которые складываются из двух измерений, в каждом из которых можно выбрать один из двух вариантов.

Первое измерение это State of the World (SotW) vs Incremental:

1. State of the World – Envoy определяет ресурсы, на которые он подписывается и каждый раз при запросе новой конфигурации требуется предоставить все ресурсы.
2. Incremental – Envoy также определяет ресурсы, на которые он подписывается, но при получении новой конфигурации следует предоставить только те ресурсы, которые поменялись.

Ключевая разница между ними заключается в том, что если, например, Envoy подписался на изменения 100 ресурсов и изменился ровно 1 ресурс, то в первом случае требуется выслать конфигурацию со всеми 100 ресурсами, а во

втором только изменившийся ресурс. К счастью, в нашем случае разница не так велика.

Заметим, что прокси которые связаны с сервисом имеют только 3 ресурса, так что тут разницы примерно нет, так как это маленькая конфигурация.

А в случае же прокси, которые выступают в роли sidecar'ов к инстансам сервисов есть 2 концептуально различающихся варианта, когда меняется конфигурация:

1. Меняется состояние инстанса – например, он начинает пользоваться новым сервисом или слушать на другом адресе, но тогда вместе с переконфигурацией прокси также происходит передеплой самого инстанса. Но тогда самая дорогая часть всепого передеплоя это как раз само разворачивание инстанса сервиса, ну и тогда мы можем позволить себе полностью переконфигурировать прокси.

2. Меняется адрес прокси другого сервиса, адрес которого присутствует в одном из cluster'ов этого прокси – мы считаем, что наши прокси "прибиты гвоздями" и их адреса практически не меняются. Поэтому этим случаем тоже можно пренебречь.

Получается, что лучшее решение это использовать Incremental подход но в конкретно нашей задаче при использовании State of the World проигрыш в производительности практически незаметен, поэтому я решил остановиться на последнем ради упрощения реализации и поддержки в будущем.

Другим же измерением является выбор между использованием отдельного gRPC-stream'а для каждого типа ресурса или же использовать один gRPC-stream для передачи всех ресурсов. Изначально Envoy поддерживал только первый вариант, то есть использование различных gRPC-stream'ов для разных типов ресурсов, но тогда при неаккуратной отправке ресурсов Envoy мог терять часть трафика, например, из-за следующей проблемы. Представим, что у нас был listener A, который перенаправляет запросы в cluster B.

Дальше пришло обновление конфигурации, из-за которого listener A начинает перенаправлять запросы в кластер C, но cluster'а C в конфигурации нет. И получается, что весь этот трафик будет пропадать до тех пор, пока в gRPC-stream'е для cluster'ов не придет cluster C. Выглядит, что не стоит поддерживать несколько gRPC-stream'ов между двумя сервисами, когда можно обойтись одним и также мы не хотим терять трафик из-за таких незначительных проблем , поэтому для решения этих проблем я буду использовать второй вариант – единый gRPC-stream для всех типов ресурсов и также для решения последней проблемы порядок между ресурсами будет следующим:

1. cluster – сначала отправляем все необходимы кластеры так как они не используют другие типы ресурсов

2. listener – в наших конфигурациях listener'ы используют только cluster'ы, поэтому после доставки cluster'ов можно отправить listener'ы. На самом деле listener еще неявно использует route ресурс, раз я всегда обновляю всю конфигурацию, то необходимые route'ы я буду отправлять вместе с listener'ом, для которого они требуются.

Вышеописанные рассуждения можно проиллюстрировать следующей таблицей, добавив, что я реализую вариант из левого нижнего угла.

2.4. Изначальная настройка конфигурации Envoy

Как было описано выше, наша цель заключается в том, что мы хотим динамически подгружать все необходимые ресурсы, но для этого необходимо не только запустить приложение, которое будет поставлять ресурсы, но и сообщить Envoy'у, как он будет себя идентифицировать, в каком формате будут поступать ресурсы, с помощью какого протокола будет происходить общение с сервером и также определить, по какому адресу будет админ панель, которая нужна для Monitoring Service.

	State of the World (SotW)	Инкрементальный подход
Отдельный gRPC-stream для каждого типа ресурса	Всегда обновляется вся конфигурация и для разных типов ресурсов используются разные gRPC-stream'ы	Обновляются только изменившиеся ресурсы и для разных типов ресурсов используются разные gRPC-stream'ы
Общий gRPC-stream для всех ресурсов	Всегда обновляется вся конфигурация и используется единственный gRPC-stream для передачи всех ресурсов	Обновляются только изменившиеся ресурсы и используются единственный gRPC-stream

Таблица 1: Варианты транспортного протокола

Панель администратора нужна для двух вещей. Во первых, она позволяет получать приватную информацию о состоянии инстанса Envoy, и, во вторых, она предоставляет возможность администраторам управлять состоянием инстанса, например выключить его. В данной ситуации конфигурация достаточно проста и просто содержит адрес, на котором Envoy будет слушать запросы к панели администратора.

```
admin:
  address:
    socket_address:
      address: <admin_address>
      port_value: <admin_port>
```

Management Server может поддерживать gRPC-stream'ы сразу с несколькими инстансами Envoy, поэтому необходим механизм для того, чтобы определять, какому инстансу Envoy какая конфигурация соответствует. Для реше-

ния этой задачи в изначальной конфигурации можно определить 2 строковых поля node.cluster и node.cluster. Названия этих полей хорошо сочетаются с выбранной нами архитектурой для Service Mesh так как у нас инстансы Envoy тоже поделены на кластера по принципу один кластер инстансов Envoy для множества инстансов одного сервиса, а внутри кластера они также поделены на sidecar-proxy отдельных инстансов сервиса и на отдельный Envoy-balancer.

```
node:
  cluster: <proxy_cluster>
  id: <proxy_id>
```

Далее следует определить протокол, по которому будут передаваться ресурсы с Management Server'а. Мы будем пользоваться V3 версией транспортного протокола, так как это самая новая, на данный момент, версия и также будем использовать единый gRPC-stream для получения всех типов ресурсов. Здесь также требуется указать сами настройки общения с Management Server'ом, но пока можно сказать, что это будет ресурс типа cluster с именем xds_cluster и мы определим его чуть ниже. Также здесь необходимо указать протокол получения ресурсов типов cluster и listener(параметры dynamic_resources.cd и dynamic_resources.lds_config), которые в нашем случае получаются через агрегированный стрим и что будет использоваться V3 версия api.

```
dynamic_resources:
  ads_config:
    api_type: GRPC
    transport_api_version: V3
    grpc_services:
    - envoy_grpc:
        cluster_name: xds_cluster
  cds_config:
    resource_api_version: V3
    ads: {}
  lds_config:
    resource_api_version: V3
    ads: {}
```

Настройка же ресурса xds_cluster находится в статической части конфигурации. В ней следует указать адрес Management Server'а, настроить протокол и указать таймауты. В качестве протокола я решил использовать http2, в каче-

стве адреса Management Server'а следует указывать адрес Controller'а, откуда впоследствии будут получаться новые конфигурации. В качестве таймаутов были выбраны дефолтные значения в виде 5 и 30 секунд и при тестировании приложение было решено их оставить.

```
static_resources:
  clusters:
  - connect_timeout: 5s
    typed_extension_protocol_options:
      envoy.extensions.upstreams.http.v3.HttpProtocolOptions:
        "@type": type.googleapis.com/envoy.extensions.upstreams.http.v3.HttpProtocolOptions
        explicit_http_config:
          http2_protocol_options: {}
            connection_keepalive:
              interval: 30s
              timeout: 5s
    name: xds_cluster
    load_assignment:
      cluster_name: xds_cluster
      endpoints:
      - lb_endpoints:
        - endpoint:
            address:
              socket_address:
                address: <controller_address>
                port_value: <controller_port>
```

Как мы видим, у нас в системе есть много Envoy-proxy, каждый из которых требует свой изначальный конфигурационный файл. Также можно заметить, что каждый из этих файлов содержит совпадающие части такие как настройка транспортного протокола Envoy, "почти" совпадающие части такие как таймауты, так как дефолтные таймауты будут хороши почти всегда, но в некоторых ситуациях их будет необходимо дополнительно настраивать и части конфигурации, которые из раза в раз будут меняться, например, идентификатор node.cluster и node.id. Поэтому, для упрощения написания конфигурации для Envoy я написал bash-скрипт, который можно найти по адресу https://github.com/iskander232/Controller/blob/master/create_envoy_config.sh в качестве аргументов принимает опции конфигурации Envoy, такие как:

- –cluster-id – cluster.id
- –node-id – node.id
- –controller-port – порт, через который устанавливается gRPC-stream с

Controller’ом

- –controller-address – адрес, через который устанавливается gRPC-stream с Controller’ом
- –admin-api-port – порт панели администратора
- –admin-api-address – адрес панели администратора

Скрипт можно запускать следующей командой, после исполнения которой в stdout будет написан корректный файл конфигурации созданный по вышеописанным правилам, который, например, с помощью перенаправления потока можно записать в файл и потом на основе этого конфига запустить Envoy. Также на основе вывода скрипта видно, что опции –admin-api-port и –admin-api-address опциональны и в случае отсутствия хотя бы одного из них панель администратора будет отсутствовать.

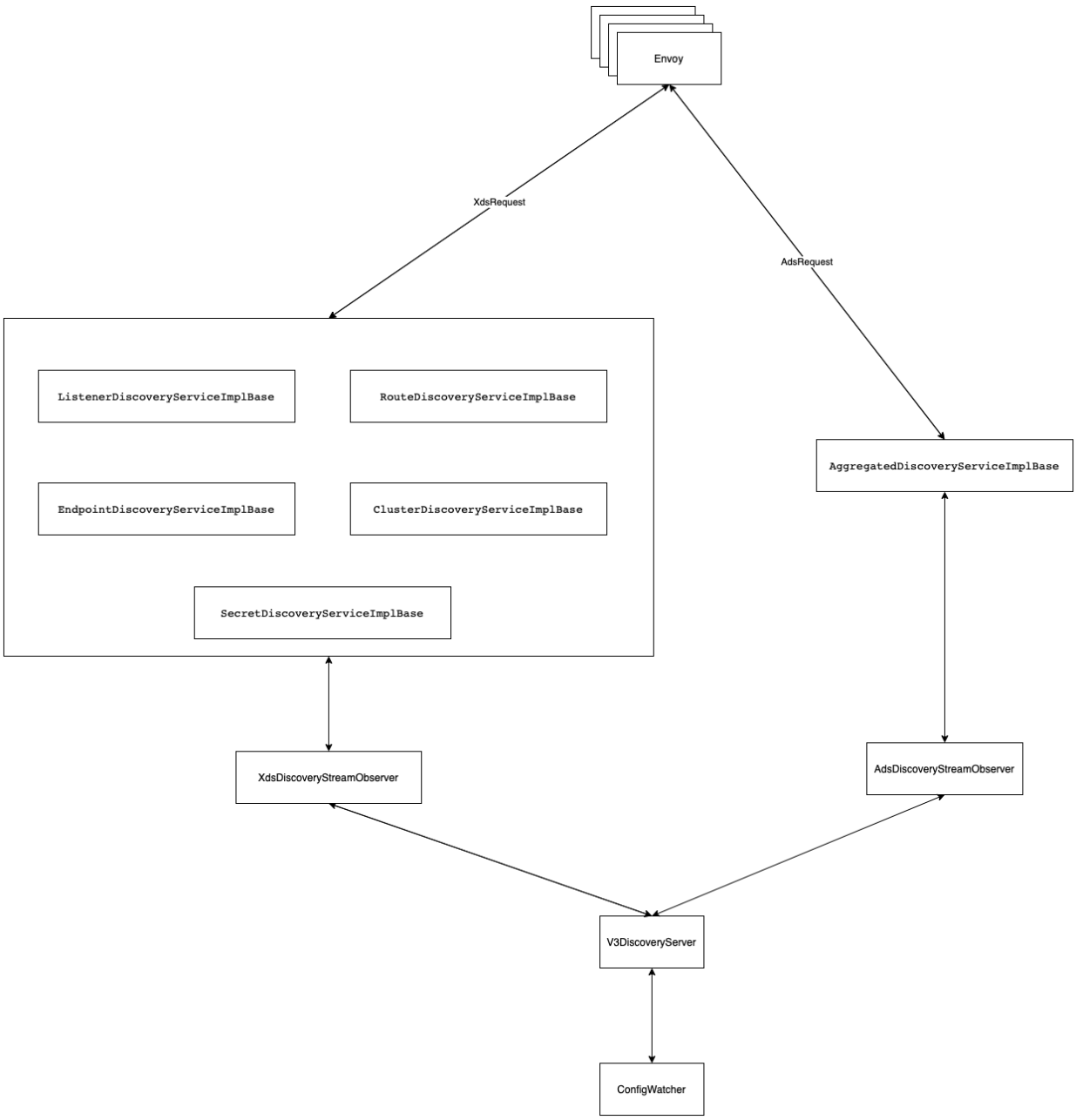
```
bash$ ./create_envoy_config.sh --cluster-id=\"1\" --node-id=test-node --controller-port=18000 --controller-address=127.0.0.1
node:
  cluster: "1"
  id: test-node

dynamic_resources:
  ads_config:
    api_type: GRPC
    transport_api_version: V3
    grpc_services:
    - envoy_grpc:
        cluster_name: xds_cluster
  cds_config:
    resource_api_version: V3
    ads: {}
  lds_config:
    resource_api_version: V3
    ads: {}

static_resources:
  clusters:
  - connect_timeout: 5s
    typed_extension_protocol_options:
      envoy.extensions.upstreams.http.v3.HttpProtocolOptions:
        "@type": type.googleapis.com/envoy.extensions.upstreams.http.v3.HttpProtocolOptions
        explicit_http_config:
          http2_protocol_options: {}
            connection_keepalive:
              interval: 30s
              timeout: 5s
    name: xds_cluster
    load_assignment:
      cluster_name: xds_cluster
      endpoints:
      - lb_endpoints:
        - endpoint:
            address:
              socket_address:
                address: 127.0.0.1
                port_value: 18000
```

2.5. Протокол общения между Envoy и Controller

Ранее было показано как происходит изначальная конфигурация Envoy и теперь можно перейти к тому, каким образом реализовано общение между Envoy и Controller. В качестве основы была взята библиотека java-control-plane [**EnvoyControlPlane**], где реализованы основные классы для создания control plane для Envoy и также приведена демонстрационная реализация, в которой все необходимые конфигурации Envoy лежат в HashMap. К сожалению, эта реализация нам не подходит. В нашей реализации Service Mesh есть компонента, ответственная за хранение информации о состояниях инстансов сервисов и которая умеет из этого знания определять, каким образом надо конфигурировать Envoy-proxy и также хранит у себя эти файлы. Конечно же, файлы, которые составляет Service Discovery сильно отличюется по формату от файлов, которые необходимы Envoy'у так как мы хотим инкапсулировать логику общения с Envoy в компоненту Controller, а в запросах к Controller'у мы хотим описывать в более простом формате требования к Envoy-proxy.

Сравнение запроса Service Discovery $\rightarrow$ Controller и Controller $\rightarrow$ Envoy.

Рассмотрим, как эта библиотека обрабатывает запросы Envoy'а на получение актуальной конфигурации, что происходит при создании нового gRPC-stream или при его рестарте и также на то, как происходит обновление конфигурации Envoy при изменении элементов в HashMap с конфигурациями Envoy в демонстрационной реализации.



- Envoy поддерживает двунаправленный gRPC-stream с несколькими gRPC сервисами из XDiscoveryServiceBase или с одним gRPC сервисом AggregatedDi Я использую агрегированный gRPC-stream для всех типов ресурсов Envoy, поэтому это будет второй вариант.

- Далее запрос из gRPC-сервера обрабатывается с помощью XdsDiscoveryStream или с помощью AdsDiscoveryStreamObserver. В этой части происходит сведение обеих V2 и V3 типов запросов к единому интерфейсу и обогащения запроса информацией о последнем предыдущем запросе.

- V3DiscoveryServer в этой цепочки выступает связывающим звеном. В нем храняться callback'и, которые следует вызывать при закрытии gRPC-stream'а и который передает запрос дальше в ConfigWatcher.

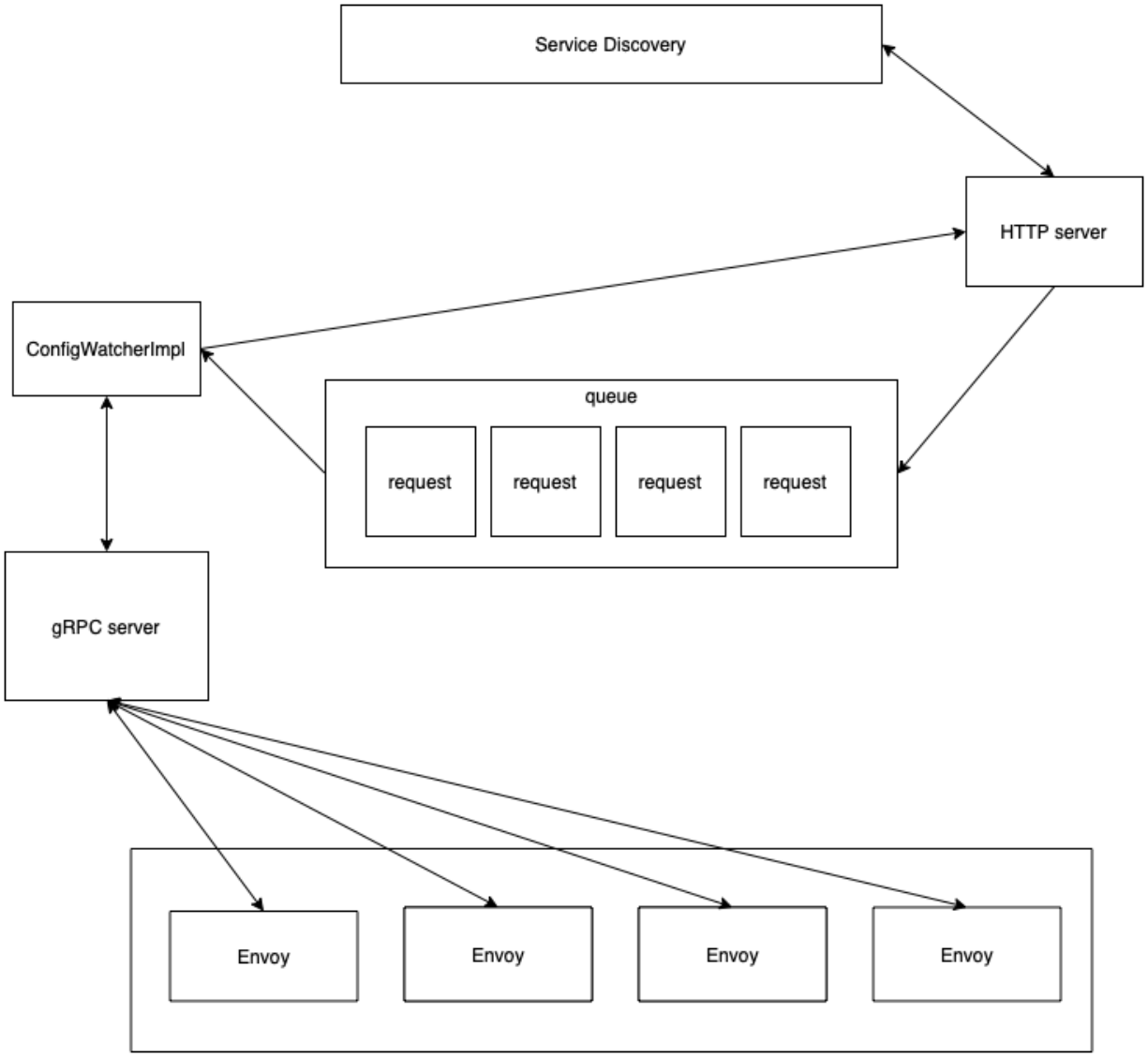
- ConfigWatcher выступает основной компонентой, которая следит за изменением конфигурации и до которой доходит вся предобработанная информация об исходном запросе. Именно здесь определяется, какую конфигурацию требуется отправить в Envoy. Для этих целей создается структура Watch, в которой хранится обработанный запрос от Envoy и функция, которую необходимо вызвать, чтобы получить требуемую конфигурацию.

Выше рассматривалась только ситуация, когда запрос поступает с Envoy, но мы также должны обрабатывать ситуации изменения конфигурации со стороны клиента, собственно, для этих целей и проектируется данное приложение. Для решения этой задачи достаточно в ConfigWatcher посмотреть на список всех зарегестрированных Watch'ей, найти среди них тот, который отвечает за необходимый нам тип ресурса Envoy, с помощью него отправить запрос обновления конфигурации с помощью той же цепочки и удалить этот Watch так как после получения этого запроса Envoy обратно отправит сообщение о том, смог ли он перейти на новую версию.

Получается, что в этой цепочке надо сделать свою имплементацию ConfigWatcher которая не будет внутри себя хранить HashMap, а будет следовать необходимой нам логике.

2.6. Архитектура созданного приложения

Рассмотрим, на какие компоненты делится приложение и как происходит обработка запроса на обновление конфигурации конкретного инстанса Envoy.



- Для начала нам нужен HTTP-server для прослушивания запросов, далее этот сервер может напрямую передовать эти запросы в ConfigWatcherImpl, но мы решили, что для Service Discovery было бы удобно, если запросы поступают батчами так как стоит учитывать что каждый из запросов выполняется не мгновенно, и также учитывая то, что иногда размер этого батча может равняться количеству инстансов некоторого сервиса, например, когда изменяются зависимости этого сервиса, а количество инстансов сервиса может быть достаточно велико, особенно в микросервисной архитектуре. Получается, что нам нужна очередь, чтобы складировать ресурсы, поэтому вместо передачи запросов в ConfigWatcherImpl он будет лежать в очереди.

- Далее возникает вопрос, что же использовать в качестве очереди. Здесь можно использовать или отдельное приложение для этого, например Kafka или RabbitMQ или пойти по более простому варианту и использовать встроянный в язык Java контейнер, такой как ArrayDeque. Я решил остановиться на последнем, так как сейчас нет предпосылок к тому, чтобы эта очередь не влезала в оперативную память и также у нас есть еще один ограниченный ресурс – количество одновременно поддерживаемых gRPC-stream'ов и непонятно, какой из них заканчивается быстрее.

- У этой очереди будет обработчик, который по одному из сигналов 1) в пустую очередь добавился элемент; 2) обработан предыдущий элемент и очередь не пустая; будет брать следующий элемент из очереди, трансформировать его в валидную Envoy-конфигурацию и перенаправлять в ConfigWatcherImpl.

- ConfigWatcherImpl, как было описано в предыдущей части, через envoy-control-plane определяет, что необходимо послать в инстанс Envoy

- Через gRPC-server происходит передача сообщения Envoy и получение от него ответа.

- Ответ из gRPC-server'а передается в ConfigWatcherImpl, который его трансформирует в ответ для Service Discovery о том, что переконфигурирование произошло успешно или нет.

Также стоит показать, что происходит, когда запрос актуальной конфигурации приходит от самого Envoy. Это может произойти, например, при первичном создании gRPC-stream'а или когда этот gRPC-stream пересоздается. В данном случае обработка запроса происходит следующим образом.

- gRPC-server принимает запрос, обрабатывает его, как было описано выше и передает в ConfigWatcherImpl

- ConfigWatcherImpl делает запрос к Service Discovery о получении необходимой конфигурации для инстанса Envoy с конкретным идентификатором ноды
- Через gRPC-server полученная конфигурация доставляется к инстансу Envoy и тот отвечает результатом переконфигурирования
- Результат переконфигурирования отправляется к Service Discovery

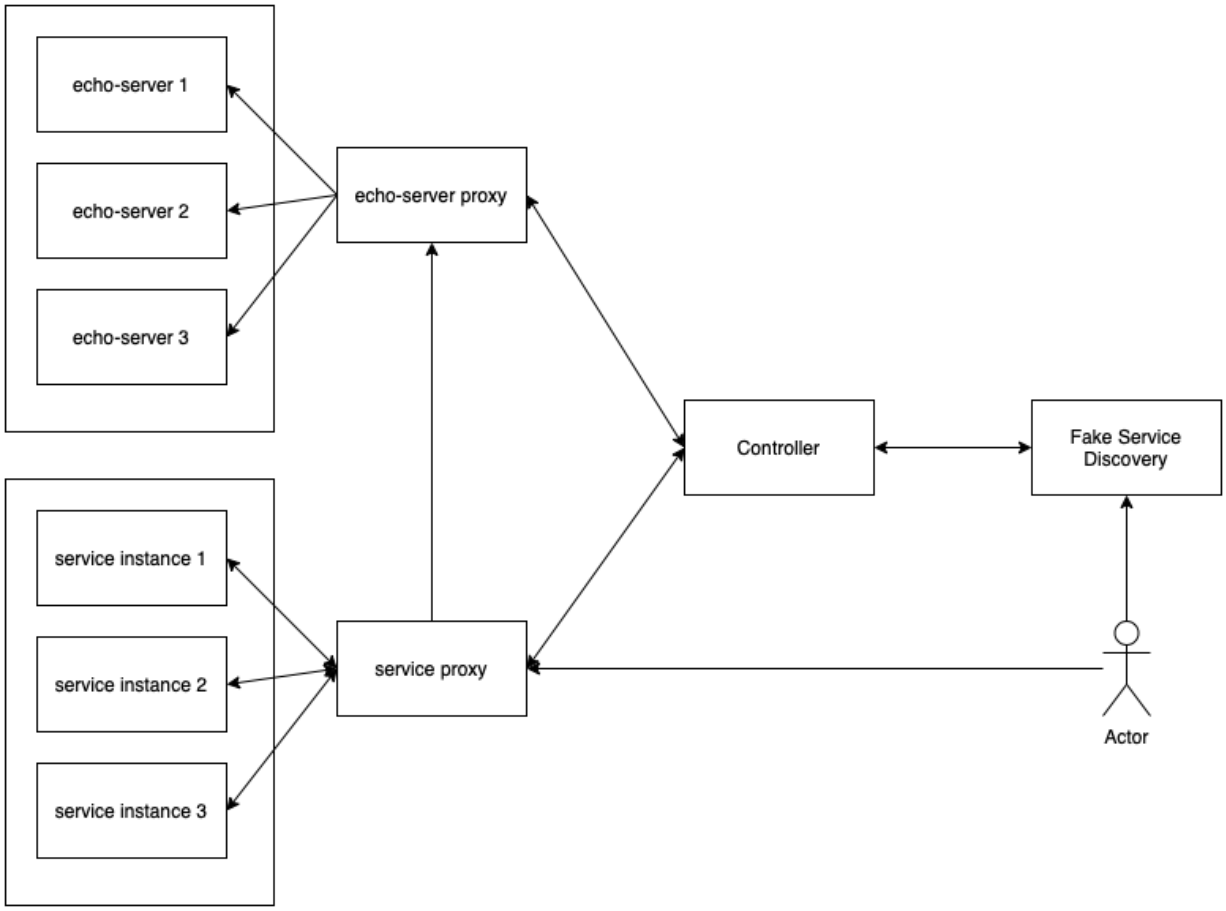
2.7. Тестирование

Заметим, что для тестирования нашего приложения нет необходимости создавать мини-систему, над которой работает Service Mesh, а достаточно просто запустить несколько приложений и для каждого из них динамически конфигурировать Envoy-proxy и смотреть, что они правильно работают. Также, к сожалению, не получится просто отправлять Http-запросы к Controller, так как у него есть функционал, который отправляет запросы в Service Discovery, который тоже требуется протестировать. Получается, что необходимо проверить следующие функции приложения Controller:

1. Можно конфигурировать Envoy как egress-proxy, то есть прокси, которая будет правильно обрабатывать запросы которые выходят из приложения и идут через прокси.
2. Можно конфигурировать Envoy как ingress-proxy, то есть прокси, которая получает запросы от других сервисов и должна перенаправить приложению
3. Корректность Http-запросов от Controller к Service Discovery.

Для тестирования было написано приложение, эмулирующее Service Discovery следующим образом. Приложение Fake Service Discovery хранит в HashMap

известные ему конфигурации Envoy и отдает их по запросу Controller. Также оно имеет Http-интерфейс, через который оно получает от пользователя новые конфигурации Envoy и запросы от Controller. Также было написано 2 приложения на Python, первое из которых является echo-server'ом, а второе выступает в роли Http-servera и в процессе работы использует этот echo-server.



Тестированаие проводилось следующим образом. Изначально было поднято 3 инстанса приложения service, 3 инстанса приложения ech-service и 2 Envoy proxy - по одному для каждого из них. Далее посылались запросы к service через его прокси, потом выбранный прокси инстанс сервиса отправлял запрос к echo-server, который изначально шел к service-proxy, который перенаправлял его к echo-service proxy и который, соответственно, перенаправлял его к нужному инстансу echo-server. Отправляя таким образом запросы проверялось, что прокси может выступать и как прокси на вход и как прокси на выход. Далее, через Fake Service Discovery посылались запросы на

изменение колличества инстансов, о которых знают прокси и у echo-server и у service. Так проверялся протокол передачи сообщений между Controller и Service Discovery.

3. Результаты

Приложения Controller и приложение для тестирования Fake Service Discovery были написаны на языке Java 17 с использованием фреймворка Spring и библиотек io.envoyproxy.controlplane, io.grpc. Скрипт для создания шаблонной конфигурации Envoy был написан на языке bash. И приложения echo-server и service для тестирования были написаны на языке Python с использованием flask.

4. Заключение

В рамках данной работы проводилось исследование существующих подходов к реализации Service Mesh и на основе рассмотренных подходов была предложена архитектура Service Mesh для 1С облака. Также была реализовано приложение Controller, входящие в Service Mesh Control Plane и набор инструментов, позволяющих с легко настраивать Service Mesh Data Plane.

При дальнейших исследованиях планируется проверить колличество потребляемых ресурсов данного приложения при большей нагрузке и при негативных результатах можно рассматривать такие улучшения как смена транспортного протокола Envoy на то, чтобы в рамках запроса на обновление конфигурации не происходила полная переконфигурация, а только разница между текущей конфигурацией и новой. Также очередь запросов может переполниться и может потребоваться вынести ее в отдельное приложение, такое как Kafka или RabbitMQ.

Другим вектором для будущих исследований может быть добавление новой функциональности для Service Mesh, например, обеспечение безопсности.