FLOW TRADERS

Проблемы long-lived gRPC стримов

Yury Vasyutinskiy
Team Lead, FLOW TRADERS NL

Обо мне

- 10+ пишу на Java
- 3+ года работаю во Flow Traders
- Тимлид команды Trading Engine
- Принес в компанию gRPC
- Мечтал сделать доклад на Joker

О компании

- Лидер в операциях с ETP, расширяемся в другие классы инструментов
- Market-maker на крупнейших мировых рынках
- Офисы в Амстердаме, Нью-Йорке и Гонконге
- Пишем в основном на Java, C++. Есть Python, VHDL и TypeScript
- Мы ищем разработчиков: https://www.flowtraders.com/careers

Почему gRPC

- Широкоизвестная технология
- Поддержка нескольких языков
- Продуманный подход к эволюции протоколов
- Хороший тулинг

Почему gRPC

- Простой и понятный API
- Хорошая документация
- Стримы на любой вкус из коробки

Пример проекта

Пишем движок расчета цен

Генерим цены целый день для определенного списка инструментов

Клиенты подписываются и слушают цены

Вместе с ценами передаются дополнительные поля: достоверность расчитанной цены, временные метки, сигналы для расчета и т.д.

Определим сервис

```
service PricingService {
    rpc SubscribeForPriceUpdate (PriceSubscriptionRequest) returns (stream PriceUpdate) {
    }
}

message PriceSubscriptionRequest {
    string clientId = 1;
}

message PriceUpdate {
    int64 instrumentId = 1;
    double price = 2;
    bytes auxiliaryInfo = 3;
}
```

Пишем по инструкции

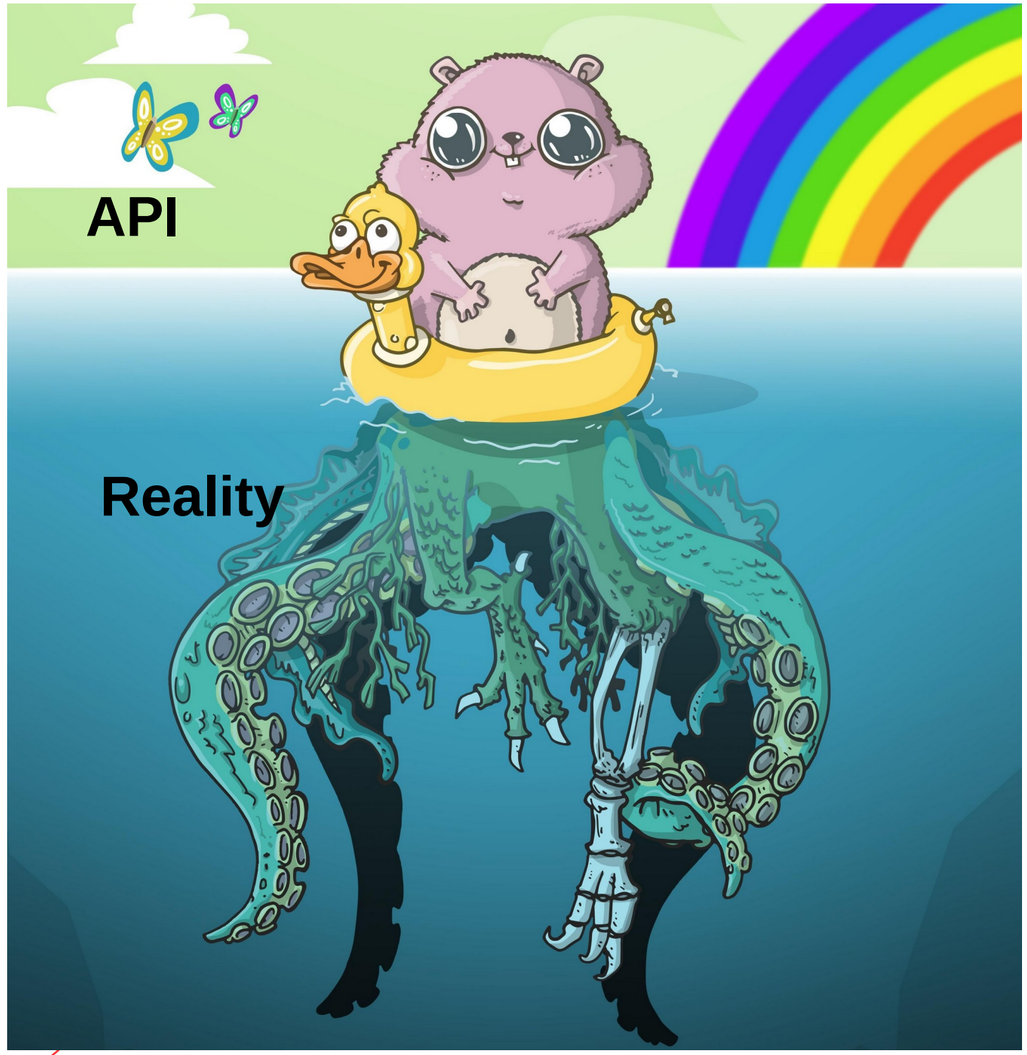
```
private final Collection<StreamObserver<PriceUpdate>> observers = ConcurrentHashMap.newKeySet();

private final PricingServiceGrpc.PricingServiceImplBase delegate =
        new PricingServiceGrpc.PricingServiceImplBase() {
            @Override
            public void subscribeForPriceUpdate(PriceSubscriptionRequest request,
                                                StreamObserver<PriceUpdate> responseObserver) {
                observers.add(responseObserver);
            }
        };
```

Пишем по инструкции

```
public void publishMessage(PriceUpdate update) {
    for (StreamObserver<PriceUpdate> observer : observers) {
        try {
            observer.onNext(update);
        } catch (Throwable e) {
            observers.remove(observer);
            observer.onError(Status.INTERNAL
                    .withCause(e)
                    .withDescription(e.getMessage())
                    .asException());
        }
    }
}
```

Красивый API



О чем поговорим

Проблемы серверного стриминга
Java реализации gRPC

- на стороне сервера
- на стороне клиента
- в web приложениях

Деплоим в прод

Решение

Приложение падает с OOM

- Природа проблемы случайна, что сильно затрудняет поиск причины
- Анализ heap dump и stacktrace уходит в недра Netty
- Верхушка айсберга видна в StreamObserver нашего сервиса

StreamObserver

void onNext(V value)

void onError(Throwable t)

void onComplete()

Разбираемся

- onNext() - асинхронный и накапливает очередь в Netty, если клиент медленный или сервер резко отправляет множество сообщений
- Контроля за очередью нет. Выстрелил и забыл
- В интерфейсе Observer нет никаких средств, чтобы это предотвратить

Как мы починили

Кастим StreamObserver в ServerCallStreamObserver

Перед отправкой сообщения проверяем флаг isReady

Если флажок выставлен – отправляем сообщение, иначе ждём

Как это работает

- Флаг isReady, на самом деле, просто проверяет размер очереди
- Внутренняя очередь Netty должна быть меньше лимита (32Кб)
- Настроить лимит, к сожалению, нельзя (пока)

Постойте

Решение

- Что значит ждать?
- Движок расчёта цен не может притормозить расчёты ради одного медленного клиента
- Складывать в памяти все сообщения мы не можем – иначе возвращаемся к проблеме 1

Выход: Backpressure

Решение

Своя стратегия для каждого клиента

- Торговый
- Логирующий
- Аналитический

Типы клиентов

Торговый клиент

- интересует только последняя цена на инструмент
- достаточно быстрый
- находится на критическом пути
- при отключении от сервера остнавливает весь трейдинг

Типы клиентов

Логирующий клиент

- пишет все цены в базу/DataLake для анализа
- медленный
- нетребовательный к задержкам

Типы клиентов

Аналитический клиент

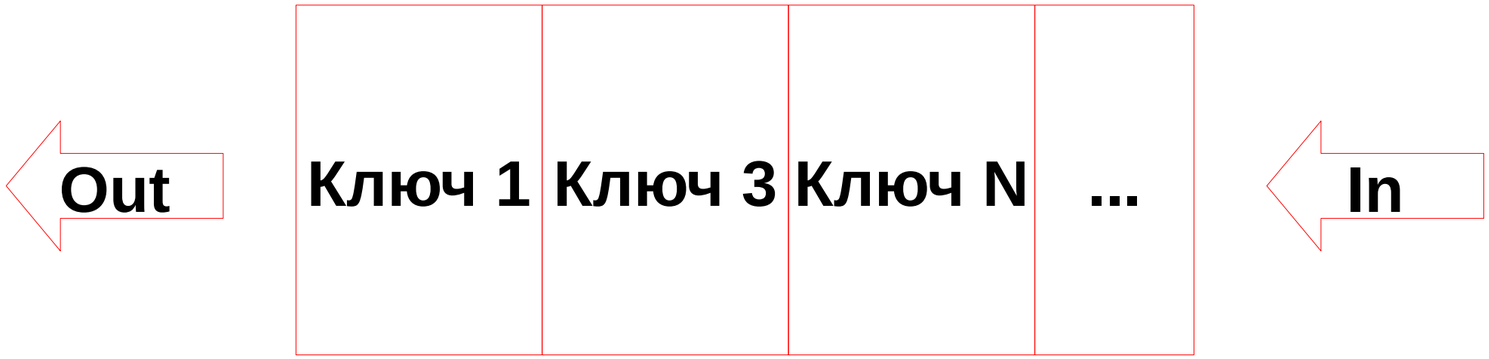
- считает разные метрики (1 клиент 1 метрика)
- очень быстрый
- заинтересован в полном временном ряду
- допускает пропуски и отключения

Решение

Throttling stream

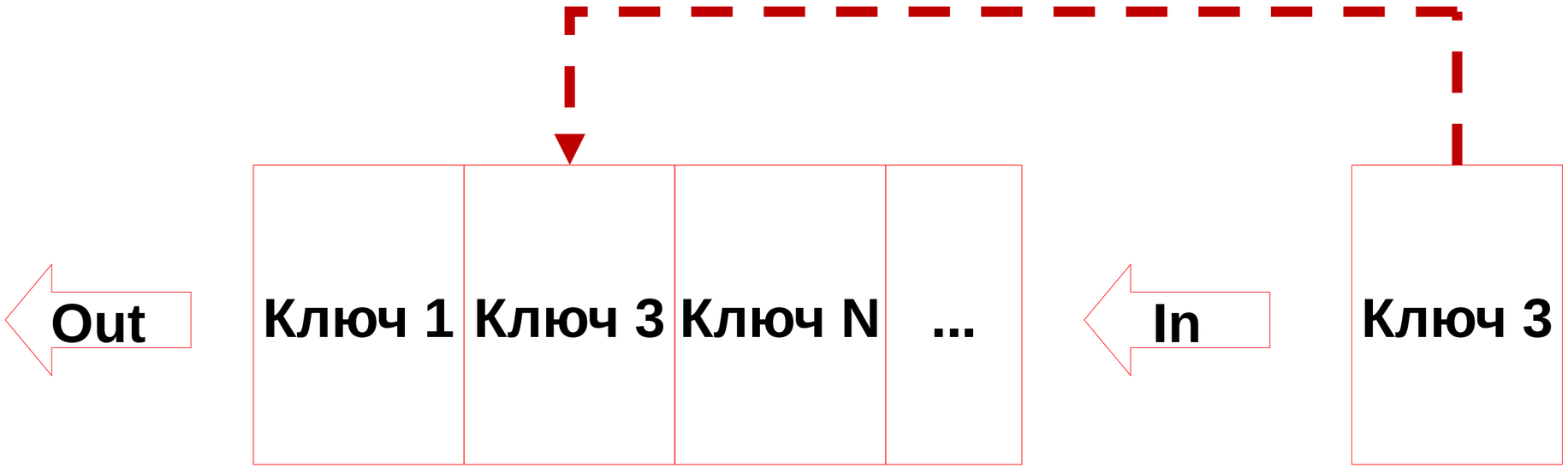
- Для каждого сообщения определяем ключ (например Id инструмента)
- Для каждого ключа храним только последнее сообщение
- Размер очереди ограничен пространством ключей
- Порядок следования ключей сохраняется

Throttling stream

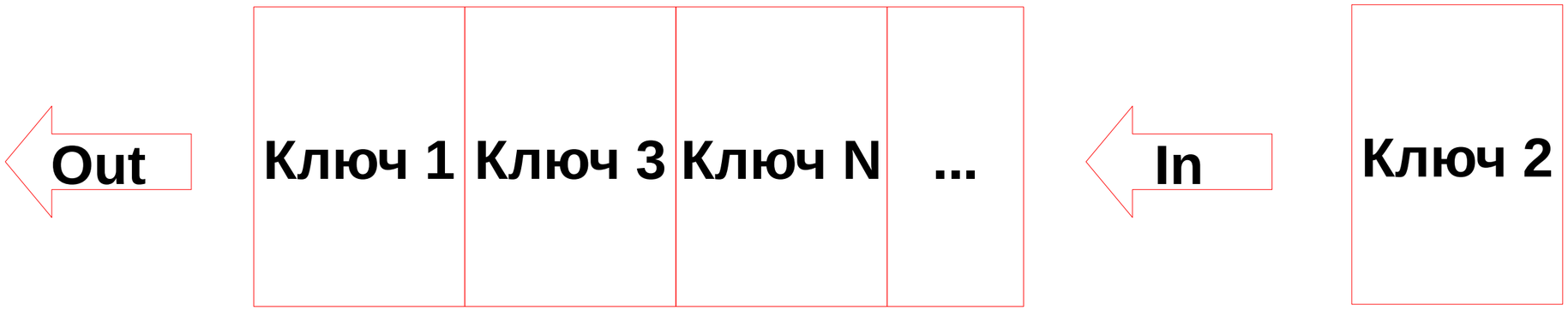


Throttling stream

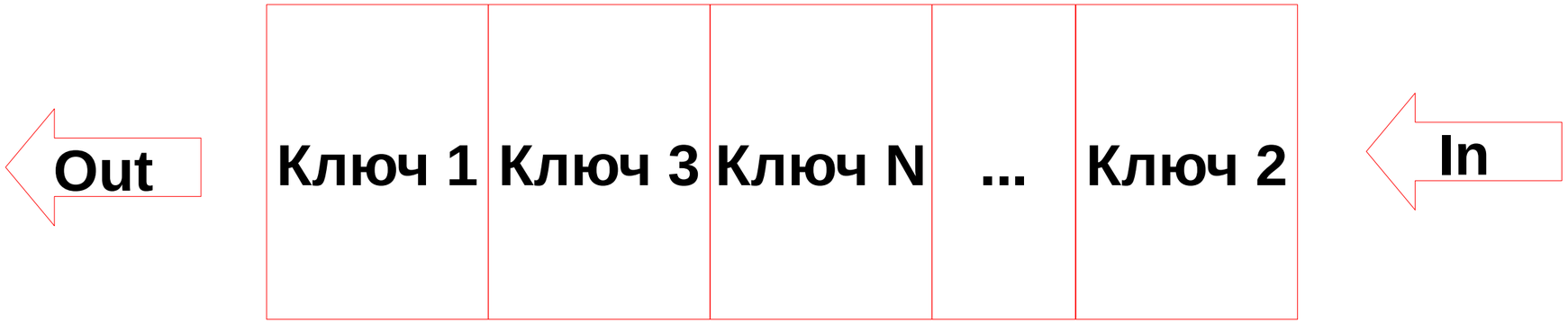
Throttling stream



Throttling stream



Throttling stream



Best practices

Throttling stream для торгового клиента

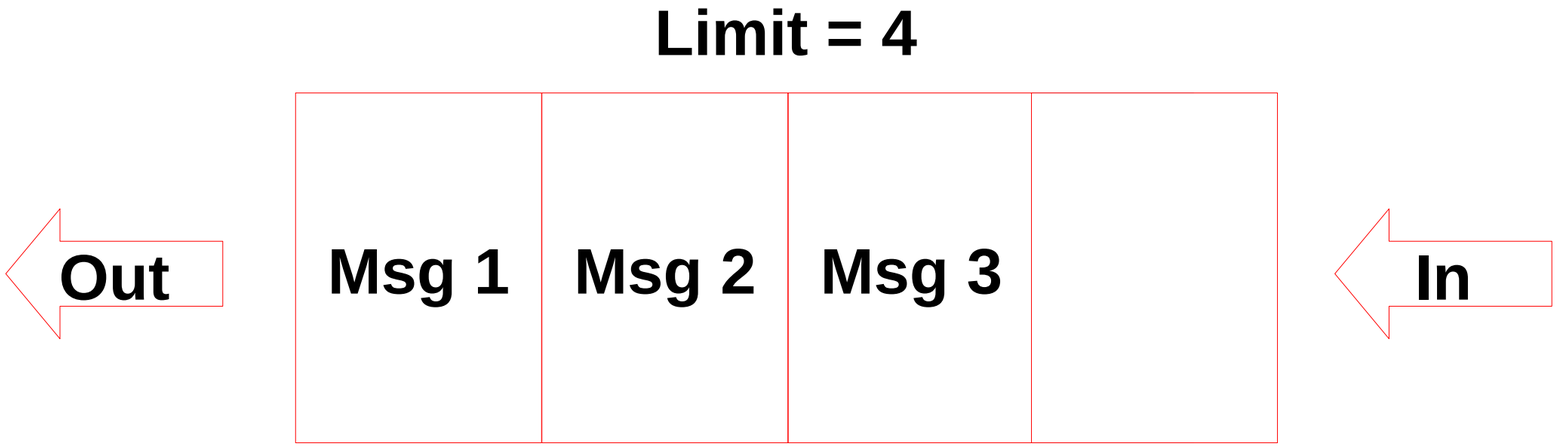
- Хороший ключ – Id инструмента
- Плохой ключ – временная метка, константа

Решение

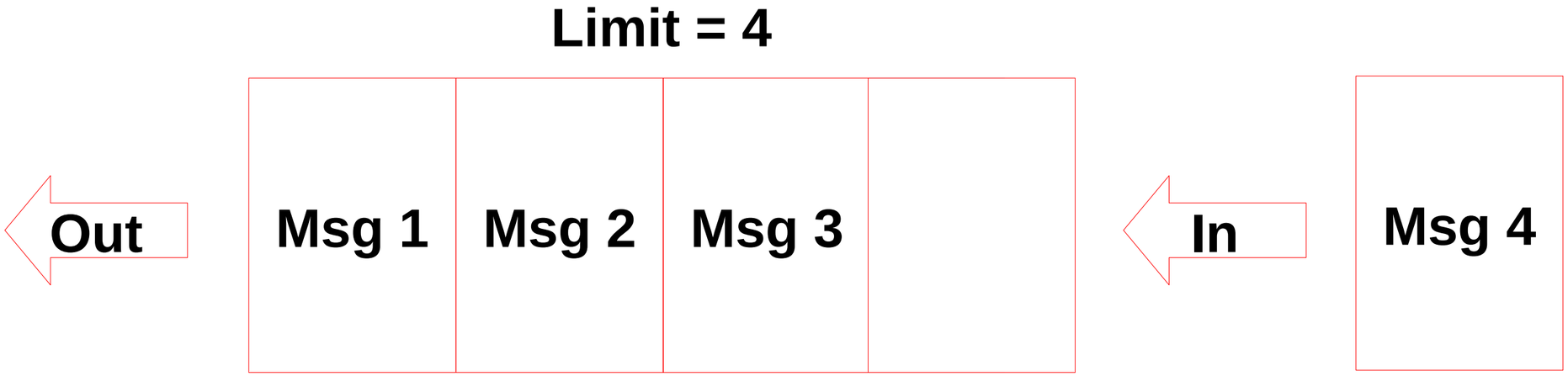
Fullflow stream

- Для каждого клиента при подключении создаем очередь в памяти
- Определяем лимит на количество сообщений
- Очередь не блокируем
- При превышении лимита отключаем клиента

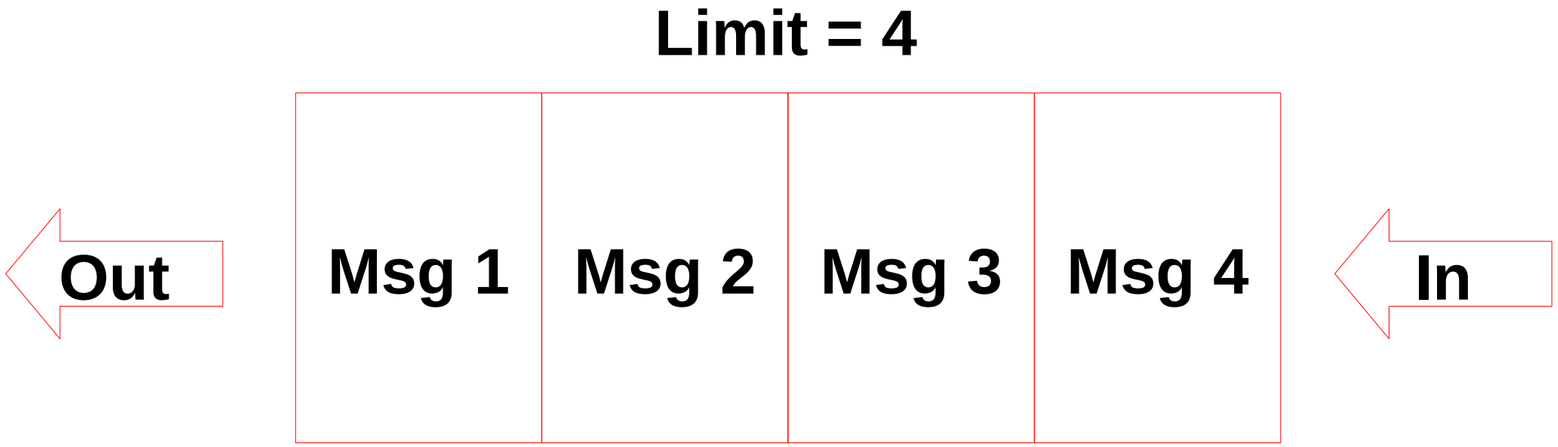
Fullflow stream



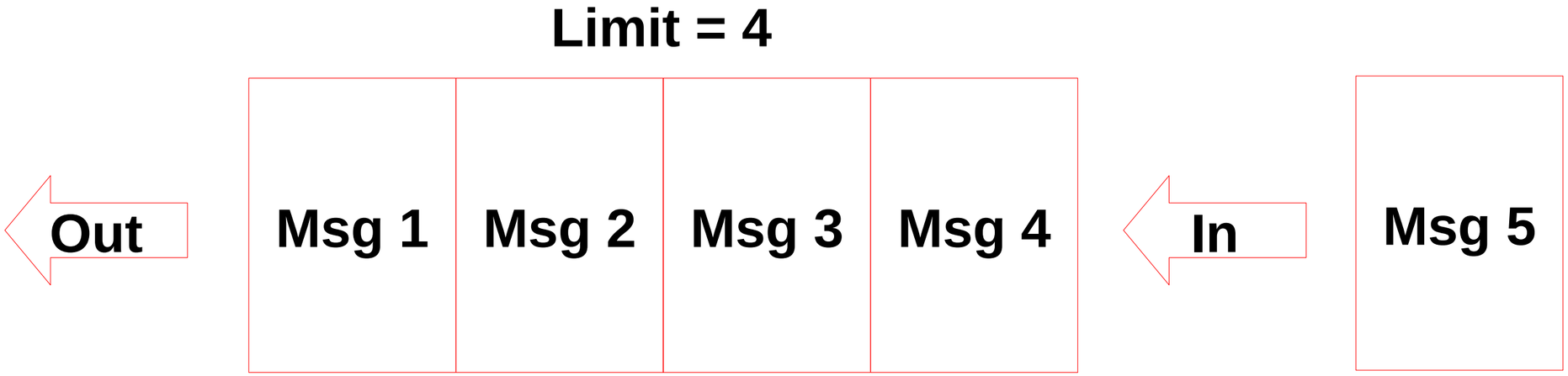
Fullflow stream



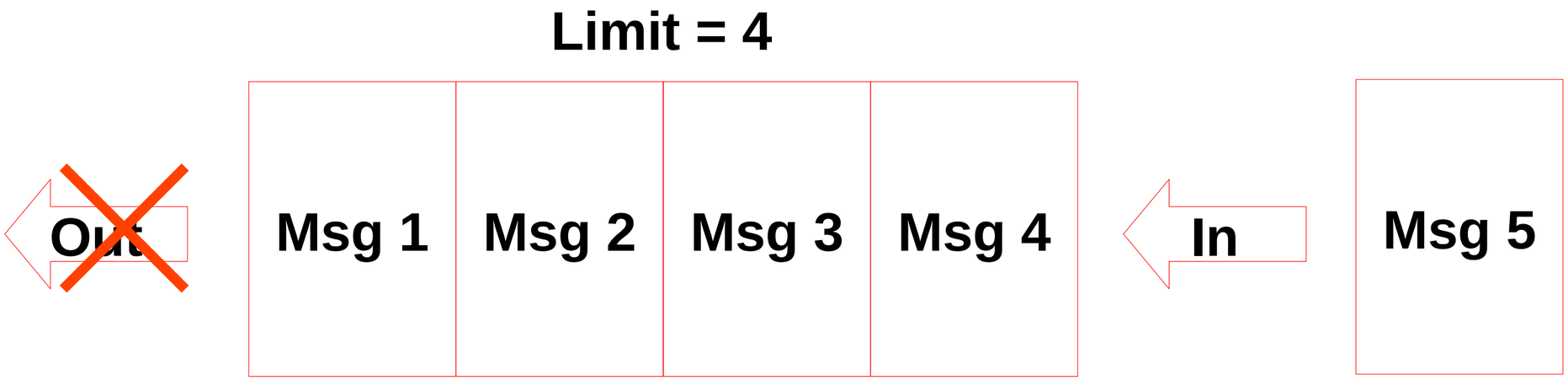
Fullflow stream



Fullflow stream



Fullflow stream



Best practices

Fullflow stream для аналитического клиента

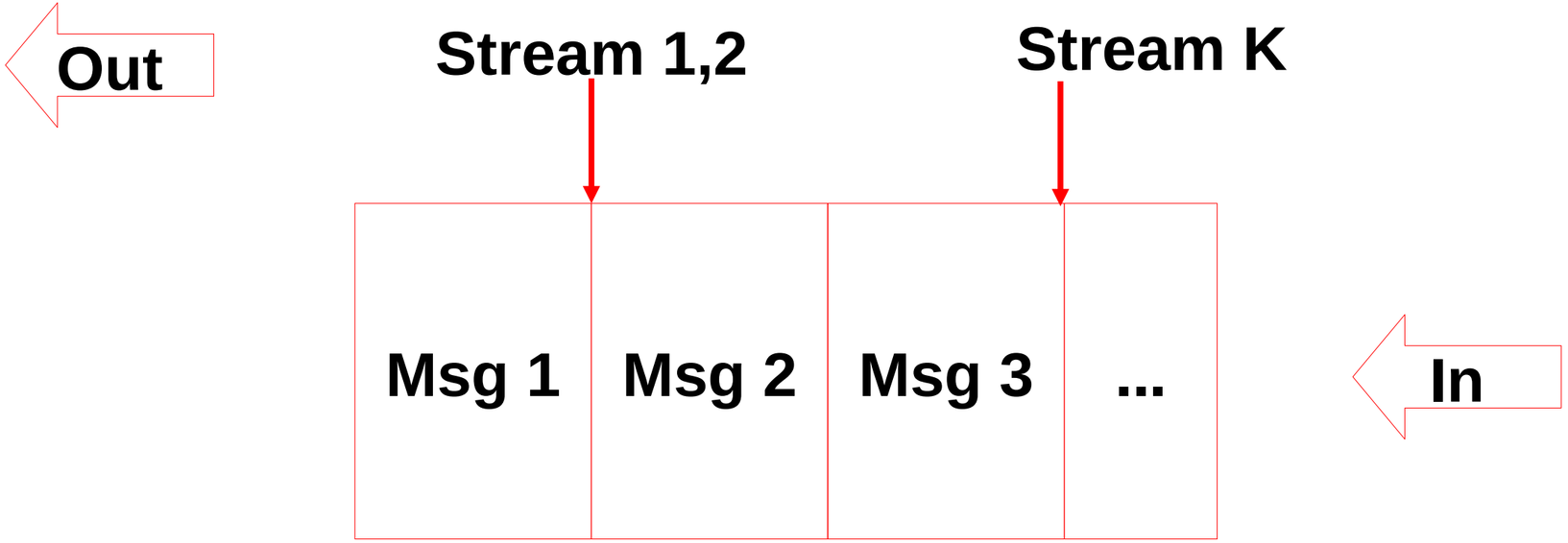
- Мониторим количество отключений
  - либо увеличиваем лимит
  - либо оптимизируем клиента
- Если клиент не успевает, то начинает заново, т.о. метрики всегда свежие

Решение

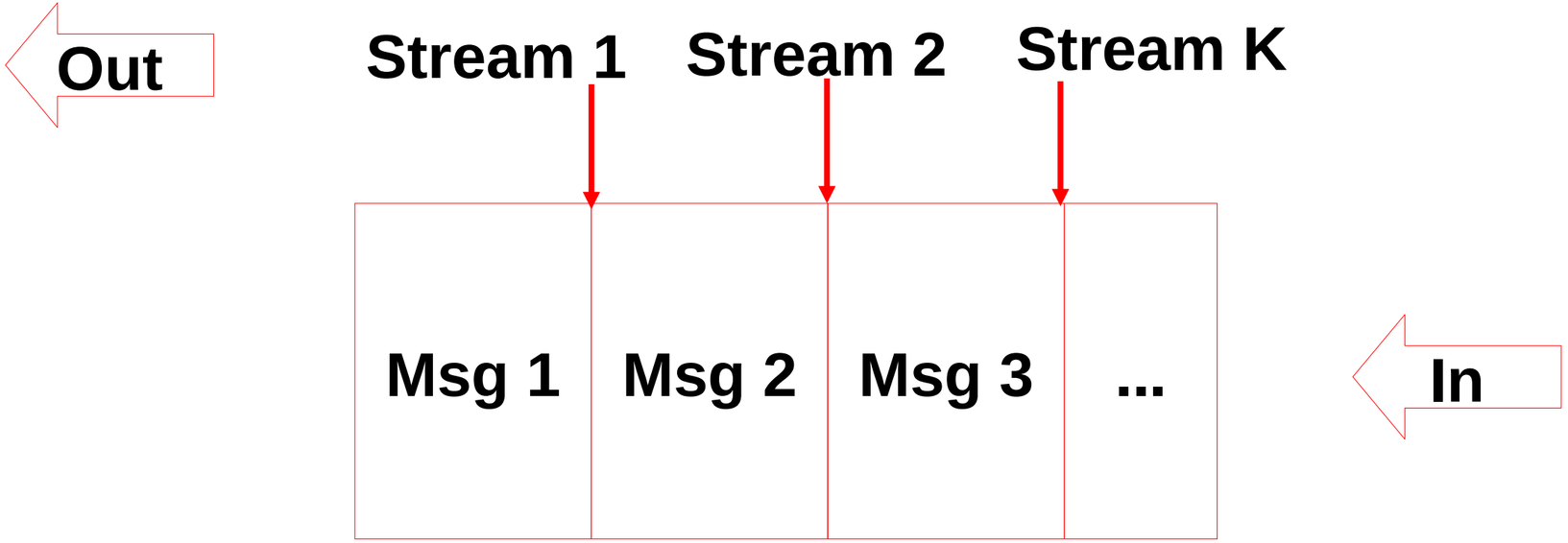
Persisted stream

- Кладем сообщения в memory-maped file с интерфейсом очереди
- Все файлы write-once, легко пользоваться в многопоточной среде
- Каждый стрим имеет указатель на свою позицию в файле
- Возникновение сообщения и его обработка разнесены по времени
- Можем эмулировать FullFlow стратегию

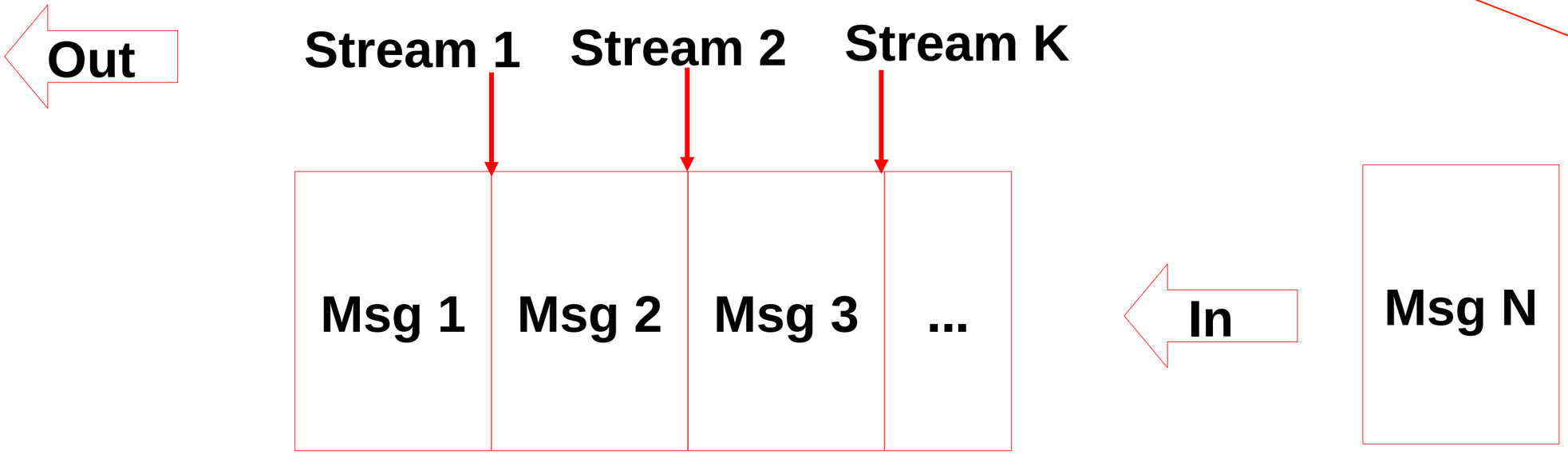
Persistent stream



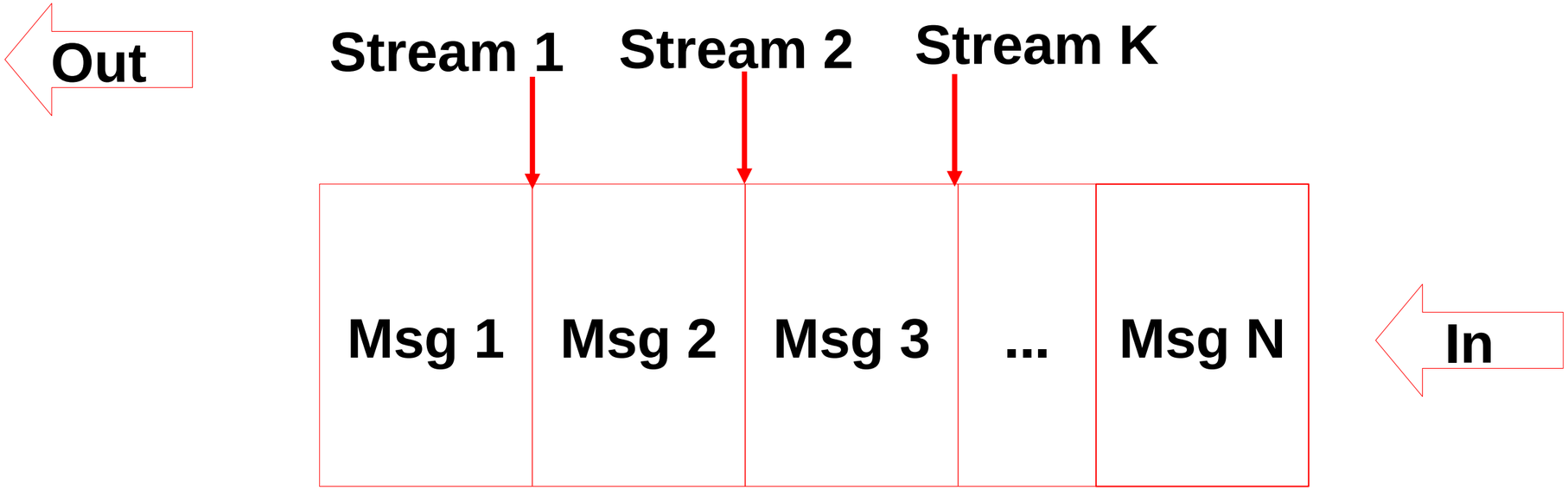
Persistent stream



Persistent stream



Persistent stream



Best practices

Persistent stream для логирующего клиента

- Мониторим диск (излишне активная запись убивает бокс)
- Удаляем старые файлы, чтобы не забить весь диск
- Можем перепослать все сообщения

Решение

Внедряем стратегии в сервис

- Определяем интерфейс Subscription
- Каждая стратегия реализует этот интерфейс
- Сервис может выбирать, какую стратегию применить

Немного кода

```
public interface GrpcSubscription<U> {

    void processUpdate(U update);

    boolean isActive();

    void onError(Throwable t);

    void onCompleted();
```

Решение

Streamer

- Все стратегии и проверки заворачиваем в класс Streamer
- Streamer обрабатывает все запросы к одному стриминговому RPC
- Принимает в себя сообщения для всех клиентов и раздает их подпискам

Итоговый сервис

```
public void subscribeForPriceUpdate(PriceSubscriptionRequest request,
                                    StreamObserver<PriceUpdate> responseObserver) {
    switch (request.getSubscriptionType()) {
        case THROTTLING:
            streamer.subscribeThrottling(request.getClientId(), responseObserver,
                    PriceUpdate::getInstrumentId);
            return;
        case PERSISTENT:
            streamer.subscribePersistent(request.getClientId(), responseObserver);
            return;
        default:  //By default, we serve clients as FullFlow
            streamer.subscribeFullFlow(request.getClientId(), responseObserver);
    }
}
```

Итоговый протокол

```
enum SubscriptionType {
    FULL_FLOW = 0;
    THROTTLING = 1;
    PERSISTENT = 2;
}

service PricingService {
    rpc SubscribeForPriceUpdate (PriceSubscriptionRequest) returns (stream PriceUpdate) {
    }
}

message PriceSubscriptionRequest {
    string clientId = 1;
    SubscriptionType subscriptionType = 2;
}
```

Проблема 3

Cancellation

- Что делать, когда клиент умер/передумал?
- Нужно уметь отлавливать это событие и корректно его обработать
- Добавляем обработчик onCancelHandler и проверяем isCancelled флаг

Решение

- Удаляем из списка подписчиков отлетевшего клиента
- Подчищаем все очереди
- Закрываем все внутренние gRPC объекты

Почему это важно

- Клиенты могут иметь баги, их сложно контролировать
- 1-2 зловредных клиента могут сильно замедлить сервер или вывести его из строя
- Все клиенты на критическом пути должны продолжать получать апдейты

Выход: жесткий контроль за подключениями и постоянный мониторинг

Summary

- Следим за размером внутренней очереди
- Используем подходящую стратегию backpressure
- При отключении клиента очищаем все объекты

Сервер молодец

Посмотрим на клиента

```
public void subscribe() {
    client.subscribeForPriceUpdate(PriceSubscriptionRequest.newBuilder().build(),
            new StreamObserver<>() {
                @Override
                public void onNext(PriceUpdate update) { processMessage(update); }

                @Override
                public void onError(Throwable throwable) { hadOnError.set(true); }

                @Override
                public void onCompleted() { hadOnComplete.set(true); }
            });
}
```

Проблема 1

Termination

- Сервер может закрыть стрим по onComplete
- В случае проблем на сервере, стрим завершается по onError
- Нас не устраивает любой из этих вариантов
- Мы хотим уметь переподключаться в любой ситуации

Хорошие новости

Переподключение это просто

- Разработчики gRPC позаботились о нас в этом случае
- Хотя клиент и асинхронный, gRPC гарантирует, что onComplete и onError могут быть вызваны только один раз
- Имплементация тривиальна

Resubscribe onComplete

```
@Override
public void onCompleted() {
    LOGGER.info("Received stream complete...");
    scheduleSafeResubscribe(channel);
}
```

Секунду, а onError?

Проблема 2

Errors

- Не все ошибки одинаково полезны
- Сервер gRPC возвращает коды ошибок. Их можно разделить на две группы: *временные* (INTERNAL ERROR, RESOURCE_EXHAUSTED, UNAVAILABLE) и *постоянные* (UNIMPLEMETED, PERMISSION DENIED)
- По коду можно понять, нужно ли пробовать переподключиться

Имплементация

```
@Override
public void onError(Throwable throwable) {
    if (isRecoverable(throwable)) {
        LOGGER.debug("Received error ...");
        scheduleSafeResubscribe(channel);
    } else {
        isActive = false;
        LOGGER.error("Received unrecoverable error. ...");
        onUnrecoverableError(throwable);
        submitSafeUnsubscribe();
    }
}
```

Проблема 3

Reconnect

- Для клиента предусмотрена только 1 стратегия переподключения – ExponentialBackoff
- Паузы между попытками увеличиваются эспоненциально
- Если сервис оживет через продолжительное время, клиент может уснуть достаточно глубоко и продлить время outage
- Настроить нельзя. Это идет вразрез с документацией и другими имплементациями

Решение

- MAX задержка – 2 минуты. Если это слишком долго, то выход один: пересоздаем канал
- Следим за количеством попыток
- Превышаем лимит – убиваем канал и создаем новый
- По умолчанию лимит = Integer.MAX_VALUE

Имплементация

```
private void safeResubscribe(Channel channel) {
    safeUnsubscribe();
    if (reconnectionAttempts.decrementAndGet() <= 0) {
        closeChannel();
        openChannel();
    } else {
        safeSubscribe(channel);
    }
}
```

Summary

- Переподключаемся при завершении стрима
- В случае ошибки на сервере проверяем, стоит ли пробовать еще
- Лечим долгий backoff сбросом канала

Клиент доволен

Как стримить в web?

- gRPC работает поверх HTTP2, но нет нормальной поддержки в браузерах
- Нет трейлеров
- Из коробки web не поддерживается, нет официального компилятора в JS
- Тулинг и комьюнити выручают: вариантов масса

Что подошло нам

- gRPC-web – проект экосистемы gRPC
- Преобразует HTTP2 в HTTP1.1 chunked-encoded стрим с кастомными трейлерами
- Необходим прокси, который будет преобразовывать бинарный протокол gRPC в Base64 encoded строки
- На выбор 2 варианта: in-process java прокси и внешний прокси envoy с gRPC-web фильтром

Привет envoy

- Java прокси не поддеживает долго живущие стримы (стоит отсечка в полсекунды)
- Envoy умеет в стримы по-долгому
- Написан на C++ – эффективный и быстрый
- Достаточно простой код фильтра

Проблемы с ним

- Есть проблема подвисающих стримов, энвой не спешит закрывать стрим на сервере, даже если клиент уже мертв
- Клиент не узнает, если умер сервер
- Кастомизация фильтра недостаточная, очень мало настроек по сравнению со стандартным клиентом
- Отдельный компонент, который надо менеджить

Что мы сделали

Переписали Java прокси: внутри Jetty с маршалингом protobuf → Base64

Добавили асинхронную обработку запросов

Убрали отсечку в полсекунды

Добавили кастомизации: размер буфера обмена, мониторинг живости клиента и т.д.

Что стоит учесть еще

- Keep-alive – если стрим не слишком активный: одно-два сообщения в час\день, то прокси могут его закрывать. Нужно либо расширять keepalive timeout или слать HeartBeat сообщения.
- Мы шлём хартбиты – пустые сообщения

Что стоит учесть еще

- Десериализация на клиенте дорогая, поэтому лучше использовать projections. Слать только нужные поля,
- Фильтрацию потока лучше оставить на сервере

Резюмируем

- Проблем в grpc стримах хватает
- Если покопаться в исходниках, то можно найти решение
- Комьюнити и труд все перетрут

Ссылки

gRPC документация:
https://grpc.io/docs/languages/java/

gRPC-web проект:
https://github.com/grpc/grpc-web

Репозиторий с примерами и имплементацией:
https://github.com/Falland/grpc-long-lived-streams

Вопросы?