Федеральное государственное автономное образовательное учреждение высшего образования
«Национальный исследовательский университет «Высшая школа экономики»

Московский институт электроники и математики им. А.Н.Тихонова

Направление подготовки

«10.03.01 Информационная безопасность»

Образовательная программа **«Информационная безопасность»**

О Т Ч Е Т

По семинару «Разработка защищенных приложений»

Студент:

Николаев М.А.

_____	_____
Фамилия И.О.	Подпись

Преподаватель:

Ведущий программист МИЭМ НИУ ВШЭ	Башун В.В.	
должность и место работы	Фамилия И.О.	подпись

Отчет защищен с оценкой _____

Дата _____

Москва, 2025

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В условиях стремительного развития распределённых систем и микросервисной архитектуры возрастает потребность в быстрых, надёжных и масштабируемых способах взаимодействия между сервисами. Одним из таких решений стал gRPC — современный фреймворк удалённого вызова процедур, разработанный Google.

gRPC построен на основе протокола HTTP/2 и использует компактный бинарный формат сериализации Protocol Buffers, что делает его значительно более производительным по сравнению с традиционными REST API. Однако высокая производительность и гибкость требуют соответствующего внимания к вопросам безопасности: передаваемые данные должны быть защищены от перехвата, подмены и несанкционированного доступа.

В рамках этого доклада мы рассмотрим:

– что такое gRPC и как он работает;

– особенности протокола HTTP/2 и связанные с ним риски (в том числе downgrade-атаки);

– как обеспечивается шифрование и защита от MITM-атак с помощью TLS и mTLS;

– какие механизмы авторизации, включая OAuth2, применяются в gRPC для контроля доступа.

Цель доклада — дать общее представление о технологии gRPC и показать, как грамотно настроить её использование в безопасных распределённых системах.

1. История gRPC и его создатели

С появлением микросервисной архитектуры в Google в начале 2010-х годов перед инженерами встала непростая задача: как обеспечить надёжный, масштабируемый и при этом лёгкий в использовании механизм обмена сообщениями между сотнями разрозненных компонентов. Внутри компании уже успешно применялись Protocol Buffers для эффективного хранения и передачи структурированных данных, но для межсервисного взаимодействия требовалось нечто большее, чем просто сериализация. Требовалось решение, способное обрабатывать длительные потоки запросов, поддерживать двунаправленные потоки данных и при этом оставаться независимым от платформы и языка программирования.

Идея gRPC зародилась как логичное продолжение этой внутренней эволюции: сочетание преимуществ Protocol Buffers с возможностями современного HTTP/2. HTTP/2 в тот момент лишь начинал входить в обиход, предлагая мультиплексирование, встроенное сжатие заголовков и приоритеты потоков. Благодаря этому инженеры Google получили инструмент, позволяющий не только быстро передавать бинарные сообщения, но и эффективно управлять соединениями, минимизируя задержки и расход TCP-сессий. Разработка велась в тени закрытых репозиториев, где команды экспериментировали с видом API, моделями потоковой передачи и механизмом авторизации.

Момент истины наступил в мае 2015 года, когда Google принял решение поделиться своим детищем с сообществом и опубликовал на GitHub проект grpc/grpc под лицензией Apache 2.0. Этот шаг стал одновременно признанием зрелости технологии и приглашением к открытому сотрудничеству: исходники оказались доступными всем желающим, а подробная документация и набор примеров дали разработчикам прочный фундамент для старта. Уже к концу года в репозитории появились стабильные реализации на C++ и Go — тех языках, которые были центром разработки микросервисов внутри Google. Следом подтянулись Java и Python, а чуть позже к ним присоединились C#, Ruby и другие популярные экосистемы.

В центре проекта оказалась небольшая, но сплочённая команда инженеров Google Cloud: Вадим Яворский, Эрик Сиггерс и Сэм Янг взяли на себя роль архитекторов и координаторов, формируя дорожную карту и следя за тем, чтобы gRPC оставался не просто «обёрткой» над HTTP/2, а полноценной платформой с богатым набором вспомогательных библиотек. Их усилия создавали мосты между миром Google и всем остальным сообществом: они активно участвовали в конференциях, писали статьи и

выступали с докладами, объясняя, почему гетерогенные микросервисы выигрывают от жёстко типизированного контракта, определённого в .proto-файлах.

К 2020 году экосистема вокруг gRPC значительно разрослась: проект получил статус «Graduated» в Cloud Native Computing Foundation, а на практике его использовали десятки крупных компаний — от финансовых стартапов до телеком-гигантов. Сообщество открытых разработчиков ежедневно вносило вклад в кодовую базу, добавляя новые плагины, расширения для сетевой безопасности и интеграции с популярными фреймворками. Именно эта открытость, подкреплённая инженерной культурой Google, позволила gRPC вырасти из внутреннего прототипа в один из стандартов де-факто для коммуникации между распределёнными системами.

2. Как работает gRPC

Работа gRPC строится вокруг жёстко типизированного контракта, описанного в .proto-файлах, и высокопроизводительного транспорта на базе HTTP/2. Всё начинается с файла, где разработчик формулирует интерфейс сервиса и структуру сообщений на языке Protocol Buffers версии 3. В разделе `service` задаётся набор RPC-методов, каждый из которых принимает входное сообщение и возвращает ответ — эти сообщения, в свою очередь, определяются блоками `message` с полями фиксированного типа и номерацией. Именно тут отражается «договорённость» между клиентом и сервером: какие данные, в каком формате и в каком порядке будут меняться между сторонами.

Далее на сцену выходит утилита `protoc`, превращающая .proto-спецификацию в живой код-скелет — так называемые «заглушки» (stubs) для клиента и сервера. Для каждого метода генерируется локальный интерфейс: на стороне клиента это выглядит как обычный метод, который можно вызвать, передав аргументы, а на стороне сервера — как абстрактный метод, который остаётся лишь «заполнить» бизнес-логикой. Всё остальное — от сериализации данных в компактный бинарный формат protobuf до управления HTTP/2-соединением, очередностью и сжатием заголовков — берёт на себя рантайм gRPC.

Сетевой обмен в gRPC умеет реализовывать сразу четыре модели взаимодействия. Самая проста — обычный вызываемый запрос-ответ, когда клиент отправляет единичное сообщение и ожидает один ответ (уни-вызов). Но гораздо гибче стриминговые варианты: сервер может отправлять серию ответов на один запрос (серверный стриминг), клиент — накапливать несколько сообщений и получить единый итоговый ответ (клиентский стриминг), а при двунаправленном стриминге обе стороны могут одновременно слать и получать последовательности сообщений, словно по автономным каналам внутри одного TCP-соединения. Эта гибкость позволяет решать задачи от простого запроса профиля пользователя до организации чата в реальном времени или передачи больших блоков данных.

Когда клиент «вызывает» метод, на деле происходит такой сценарий: локальная обёртка собирает переданные значения в структуру protobuf, упаковывает её в бинарный буфер и отправляет через HTTP/2-стрим на сервер. Серверный рантайм ловит этот пакет, распаковывает структуру, преобразует в объекты соответствующего языка и передаёт управление разработанному методу. После выполнения логики ответ проходит обратный путь: снова сериализуется, шлётся обратно по тому же или новому потоку, пока клиентский рантайм не получит и не распарсит его в привычный объект. Для стриминговых моделей этот цикл повторяется многократно, причём благодаря мультиплексированию HTTP/2 каждый поток изолирован, но при этом не требует отдельного TCP-соединения, что снижает накладные расходы и задержки.

В результате gRPC превращает процесс межсервисного общения в нечто близкое к вызову обычного метода в локальной библиотеке: жёсткий контракт .proto, автоматическая генерация кода и оптимизированный транспорт на базе HTTP/2 создают надёжный, быстрый и удобный инструмент для построения распределённых систем.

3. Особенности протокола HTTP/2 и связанные риски

С переходом к HTTP/2 каркас обмена данными в gRPC обрел новые возможности: вместо текстовых сообщений теперь используются бинарные фреймы, что повышает эффективность парсинга и снижает накладные расходы на передачу. Каждый фрейм в HTTP/2 несёт чётко заданный тип и размер, что упрощает обработку и детектирование ошибок на уровне транспортного слоя. Благодаря мультиплексированию множество логических потоков могут сосуществовать внутри одного TCP-соединения, устраняя задержки, вызванные установкой новых соединений, и позволяя одновременно пересылать запросы и ответы без очередей head-of-line blocking. Сжатие заголовков по алгоритму HPACK дополнительно сокращает объём повторяющихся метаданных, сохраняя пропускную способность канала. Наконец, хотя механизм server push редко применяется в гRPC, он открывает перспективу для предварительной отправки ресурсов до того, как клиент их запросит, что в других сценариях может снизить задержки.

Однако расширенные возможности HTTP/2 несут в себе и новые риски. При установке защищённого соединения TLS через расширение ALPN (Application-Layer Protocol Negotiation) клиент и сервер договариваются о протоколе «h2». Если на пути окажется некорректный прокси или злоумышленник, он может прервать эти переговоры, вынудив стороны опуститься до HTTP/1.1 — такого рода downgrade-атаки подрывают безопасность и производительность. Базовыми контрмерами здесь выступают HSTS, позволяющий браузеру и клиентам «заучивать» обязательность HTTPS, а также чётко настроенные политики SSL/TLS, жёстко ограничивающие набор допустимых протоколов и шифров.

В условиях высокой нагрузки HTTP/2-соединение может стать потенциальным вектором DoS-атак. Малоэффективные реализации либо «дико» настроенные серверы допускают открытие слишком большого числа параллельных потоков (MaxConcurrentStreams), что приводит к исчерпанию ресурсов. Злоумышленник может также отправлять фреймы некорректного формата, переполняя буферы и сбивая работу декодера. Чтобы противостоять таким угрозам, в gRPC-рантаймах и рядом прокси вводят ограничения на число одновременных потоков, реализуют rate-limiting на уровне запросов и используют шаблон circuit-breaker для быстрого закрытия «грязных» или слишком требовательных соединений.

На практике HTTP/2-экосистема ещё далека от идеала: старые прокси-серверы и балансировщики порой не умеют правильно обрабатывать бинарные фреймы или мультиплексирование. В результате они без предупреждения переводят трафик обратно на HTTP/1.1, что автоматически влечёт за собой увеличение числа TCP-соединений, рост задержек и утрату преимуществ шифрования, гарантированных современными TLS-настройками. Поэтому при развёртывании gRPC-сервисов важно проверять совместимость сетевых компонентов и при необходимости обновлять либо заменять устаревшие элементы, чтобы не сводить на нет выгоды от перехода к HTTP/2.

4. Шифрование и защита от MITM-атак: TLS и mTLS

В основе безопасности gRPC лежит надёжный криптографический протокол TLS, который гарантирует конфиденциальность и целостность данных, а также защищает от атак “по середине” (MITM). Когда Ваш клиент впервые устанавливает соединение с сервером, происходит серия чётко регламентированных шагов – TLS-рукопожатие. Сначала клиент отправляет сообщение ClientHello, в котором перечисляет поддерживаемые версии TLS, наборы шифров (cipher suites) и расширения, среди которых ALPN указывает на предпочтительный протокол “h2” для HTTP/2. Сервер отвечает ServerHello, выбирая из предложенных параметров оптимальный шифр и подтверждая использование TLS 1.3 (при условии, что обе стороны его поддерживают). Важнейшей частью этого обмена становится алгоритм ECDHE (Ephemeral Elliptic Curve Diffie–Hellman), который позволяет обеим сторонам сгенерировать симметричный ключ, известный только им, и одновременно обеспечить взаимную секретность при каждом новом подключении. Ключевая особенность ECDHE — «мгновенная» выработка уникального сеансового ключа без передачи его по сети, благодаря чему даже в случае будущей компрометации долгосрочных ключей злоумышленник не сможет расшифровать прошлые сеансы (свойство Perfect Forward Secrecy).

Проверка подлинности сервера осуществляется через сертификат X.509, выданный доверенным центром сертификации (CA). Сервер отправляет цепочку сертификатов, начиная от своего собственно сертификата и заканчивая, как правило, сертификатом-посредником, доверенным корневым CA, уже присутствующим в хранилище доверенных корневых сертификатов клиента. Клиент последовательно проверяет цифровые подписи и сроки действия каждого сертификата вплоть до корня. В TLS 1.3 процесс проверки был упрощён и ускорён: исключены устаревшие и небезопасные шифры, уменьшено число раундов рукопожатия (до одного полноценного RTT), что сокращает задержки при установлении защищённого канала, а заодно минимизирует вероятность ошибок конфигурации.

Кроме классического TLS, для внутреннего взаимодействия микросервисов часто применяется mTLS – взаимная аутентификация на уровне транспортного слоя. В mTLS клиент, помимо того что проверяет сертификат сервера, сам предъявляет свой сертификат и позволяет серверу удостовериться в своей собственной подлинности. Такой двусторонний обмен создаёт настоящий «криптографический мост доверия», где ни одна из сторон не рискует оказаться “подставной”. Тем не менее mTLS требует серьёзной инфраструктуры управления ключами (PKI). Необходимо организовать выпуск клиентских сертификатов, их безопасное хранение, автоматическую ротацию и своевременное отзыв (через CRL или OCSP), чтобы не оказаться заложниками просроченных или скомпрометированных ключей.

Управление сертификатами – это на практике отдельный проект. Для крупных систем часто разворачивают собственный корпоративный CA, устанавливают политики генерации ключей (например, минимальная длина эллиптической кривой, алгоритмы подписи), настраивают интеграцию с HSM (аппаратными модулями безопасности) или системами автоматизированного выпуска сертификатов по протоколу ACME. Сертификаты распределяются при помощи конфигурационных менеджеров или сервисов обнаружения:

при запуске нового экземпляра микросервиса он автоматически получает актуальный сертификат и доверенный корень, а старые ключи удаляются по мере истечения срока действия. Хотя mTLS даёт высочайший уровень уверенности, что “по обе стороны” находятся ожидаемые участники, внедрение подобной схемы требует значительных усилий по настройке и сопровождению, а также дополнительных вычислительных ресурсов при рукопожатии.

MITM-атаки на уровне TLS становятся минимально вероятными, когда Вы сочетаете современные версии протокола (рекомендовано TLS 1.3), строгие политики SSL (запрет на использование устаревших шифров и протоколов), HSTS для “заучивания” обязательного HTTPS и мониторинг сертификатов через механизмы certificate transparency. В условиях mTLS риск снижается ещё больше: злоумышленнику пришлось бы не только поддельно выдать себя за сервер, но и получить валидный клиентский сертификат, выданный Вашим CA, что практически невозможно без компрометации всей инфраструктуры. Так gRPC вместе с TLS и mTLS создаёт надёжный фундамент для обмена данными в распределённых системах, сочетая производительность HTTP/2 с проверенными механизмами криптографической защиты.

5. Механизмы авторизации

Мир микросервисов требует надёжных и гибких способов управления доступом — чтобы не только убедиться, что у клиента есть право вызвать конкретный метод, но и разграничить полномочия внутри сложных распределённых систем. На сегодняшний день одним из самых распространённых подходов в gRPC стало использование протокола OAuth 2.0 в связке с JSON Web Tokens (JWT). Сначала клиент, действуя от имени пользователя или сервиса, обращается к Identity Provider и получает временный access-токен. Этот токен упаковывается в HTTP/2-метаданные gRPC-запроса — простой заголовок «authorization: Bearer <token>». На стороне сервера специальный middleware или встроенный в рантайм gRPC-интерцептор извлекает токен, проверяет его цифровую подпись по публичным ключам в формате JWKs, убеждается в том, что токен ещё не истёк, и что в поле scope или claims содержится требуемая роль или право. В случае успеха интерцептор передаёт в сервис бизнес-логику уже доверенный объект аутентифицированного субъекта, избавляя разработчика от рутины парсинга и валидации токенов.

Однако не всегда есть возможность разворачивать полноценный OAuth 2.0 с Identity Provider. Для простых сценариев часто используют API-ключи: уникальные строки, выдаваемые клиентам и передаваемые в тех же метаданных gRPC-запроса. На сервере их проверяют по заранее известному списку или обращаются к хранилищу, сопоставляя ключ с учётной записью и набором прав. Такой подход легче реализовать, но он уступает по безопасности: ключи сложнее отозвать мгновенно, а их утечка даёт полный доступ без возможностей ограничения по времени или области действия.

Для более гибкого распределения ответственности внутри одного приложения применяются custom-interceptors — пользовательские модули, куда выносят логику проверки ролей и прав на основе собственных правил. Интерцептор может анализировать дополнительные метаданные (например, идентификаторы организации или проектные теги), обращаться к центральному сервису авторизации или базе данных, и принимать решение “разрешить” или “отказать”. Внутри такого механизма удобно выстраивать иерархию привилегий, осуществлять детальный аудит вызовов и динамически менять политику доступа без перезапуска служб.

Наконец, для сервис-to-сервис сценариев на уровне инфраструктуры всё чаще внедряют SPIFFE/SPIRE — стандарт для выдачи и управления короткоживущими сертификатами X.509. Каждый микросервис получает уникальный SPIFFE-идентификатор в доверенном “домене” и предъявляет сертификат при установке TLS-соединения. mTLS с SPIFFE-сертификатами обеспечивает одновременно аутентификацию и авторизацию: трастовая шина SPIRE автоматически проверяет принадлежность сервисов к допустимым рабочим нагрузкам. Это устраняет необходимость в явных токенах и API-ключах, смещая ответственность за безопасность на уровень платформы, где централизованно управляются выпуски, ротация и отзыв сертификатов.

Выбор конкретного механизма авторизации зависит от требований к безопасности, удобству эксплуатации и масштабу системы. OAuth 2.0 + JWT дают стандартизованный и знакомый многим разработчикам подход с поддержкой токен-ролей и унифицированными Identity Provider. API-ключи и custom-interceptors подходят для небольших проектов или

быстрых прототипов. А SPIFFE/SPIRE выводит доверие на уровень инфраструктуры, требуя более серьёзной установки, но даря максимальную автоматизацию и безопасность при масштабировании.

Заключение

gRPC стал одним из ключевых инструментов для высокопроизводительного обмена данными в микросервисных архитектурах.

Его преимущества — двусторонний стриминг, дешёвые операции сериализации и эффективное использование сетевых ресурсов — реализуются на основе HTTP/2 и Protocol Buffers.

Однако для защиты каналов необходимо применять TLS (желательно версии 1.3) и, при высоких требованиях к безопасности, mTLS.

Для контроля доступа внешних клиентов отлично подходит OAuth2/JWT, а сервис-to-сервис коммуникацию лучше обезопасить через mTLS или SPIFFE/SPIRE.

Только комплексный подход к шифрованию, аутентификации и авторизации позволяет построить надёжную и масштабируемую распределённую систему.

Список литературы

1. Official gRPC Documentation — https://grpc.io

2. RFC 7540 — Hypertext Transfer Protocol Version 2 (HTTP/2)

3. RFC 8446 — The Transport Layer Security (TLS) Protocol Version 1.3

4. Google Cloud Whitepaper: “Building Secure gRPC Microservices”

5. Ivan Ristić. Bulletproof TLS and PKI. 2023

6. OWASP Cheat Sheet — Transport Layer Protection

7. CNCF SPIFFE/SPIRE Specifications

8. Istio Security Insights Blog Series, 2024