МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ

Федеральное государственное автономное образовательное
учреждение высшего образования
ЮЖНЫЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ

Институт математики, механики и компьютерных наук
имени И. И. Воровича

Направление подготовки 02.03.02 – «Фундаментальная информатика и информационные технологии»

Ковтун Э.А., 4 курс, 9 группа

Выпускная квалификационная работа на степень бакалавра

СИСТЕМА ИНСПЕКЦИИ СЕТЕВОГО SSL/TLS ТРАФИКА И ИМИТАЦИИ ПЛОХИХ КАНАЛОВ ПЕРЕДАЧИ ДАННЫХ ДЛЯ ТЕСТИРОВАНИЯ МЕЖСЕТЕВОГО ВЗАИМОДЕЙСТВИЯ ПРИЛОЖЕНИЙ ПО ПРОТОКОЛУ TCP

Научный руководитель:
кандидат физико-математических наук, доцент кафедры информатики и вычислительного эксперимента К. Ю. Гуфан

Допущено к защите:
Заместитель директора ИММКН
По направлению ФИИТ

_____ В. С. Пилиди

Ростов-на-Дону
2020

Оглавление

Введение

В настоящее время, в связи с развитием сетевых технологий и постоянным ростом скорости каналов передачи данных, все больше разрабатываемых приложений начинают взаимодействовать друг с другом с использованием сети. В попытках стандартизации сетевого взаимодействия и форматов передачи данных возникли спецификации такие как, например, REST или SOAP. Все чаще используются такие понятия как микросервисная архитектура, экосистема и т.п. В подобных средах разные компоненты систем могут взаимодействуют по сети друг с другом без явного участия пользователя. Компоненты подобных систем, в том числе находящихся в стадии эксплуатации, могут независимо друг от друга изменяться, дорабатываться, а также в рамках одной инфраструктуры одновременно функционировать несколько сетевых компонент разных версий.

Еще одним трендом настоящего времени является усиление требований к безопасности и повсеместный переход к использованию защищенных способов сетевого взаимодействия с использованием протоколов SSL/TLS, обеспечивающих шифрование данных, передаваемых по сети, и возможность аутентификации пользователей.

Выполнение тестирования и отладки отдельных компонент в подобных сетевых системах, в том числе, когда нужно иметь возможность полностью восстановить протокол взаимодействия нескольких компонент системы за какой-либо интервал времени, чтобы подвергнуть его детальному анализу или изучить влияние характеристик каналов передачи данных на работу системы, является актуальной задачей.

В данной работе будет рассмотрен и реализован один из подходов к тестированию компонент в сетевых средах подобной архитектуры с использованием промежуточного прокси-сервера.

Постановка задачи

Имеется эксплуатируемая вычислительная система, находящаяся в стадии постоянной поддержки и развития, построенная по правилам клиент-серверной архитектуры. Серверное программное обеспечение предоставляет API (Application Programming Interface) для взаимодействия к клиентским программным обеспечением по заранее заданным правилам. Основным протоколом взаимодействия клиентов и серверов является HTTP/HTTPs (HyperText Transfer Protocol). Сервер производит обмен информацией с клиентом по правилам REST (Representational State Transfer) – архитектурного стиля взаимодействия между распределенными системами. О самих правилах API ничего неизвестно.

Часть взаимодействий в данной системе выполняется по защищенным каналам при помощи протоколов SSL/TLS.

Требуется провести исследование к подходу тестирования данной вычислительной системы и разработать программное обеспечение, предоставляющее следующие возможности:

1) журналирование сеансов взаимодействия выбранных пар клиентов и серверов (с возможностью ведения протокола их взаимодействия с привязкой ко времени),

2) инспекцию зашифрованного трафика,

3) имитацию проблем в канале связи между клиентом и сервером (низкая пропускная способность, задержки и т.д.).

1 Исследование поставленной задачи

1.1 Исследование возможностей по тестированию и отладке сетевого взаимодействия клиент-серверных приложений

Архитектура типа "Клиент-сервер" – сетевая или вычислительная архитектура, в которой сетевая нагрузка или вычисления разделены между узлами, называемые серверами, которые, как правило, расположены на разных вычислительных ресурсах. Клиенты взаимодействуют с серверами через вычислительную сеть посредством сетевых протоколов по правилам, заданным со стороны серверного программного обеспечения (Рисунок 1).

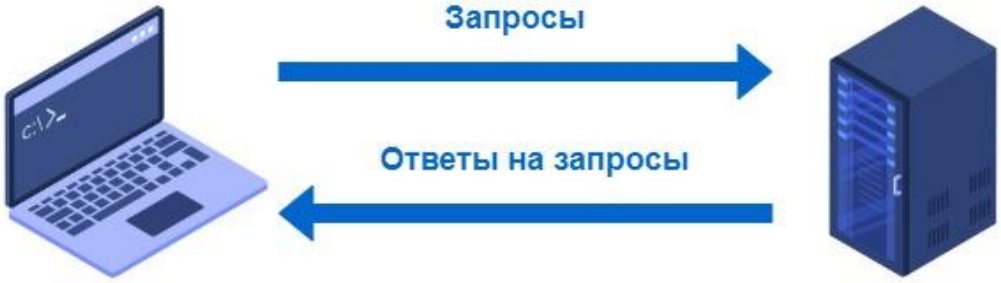

Рисунок 1 – Клиент-серверная архитектура

Рассмотрим простую схему клиент-серверной архитектуры, обозначенную на рисунке 1.

1. Клиент (заказчик услуг) – программное обеспечение, нацеленное на взаимодействие с пользователем через пользовательский интерфейс. На основание пользовательских команд запрашивает данные у сервера или просит его произвести необходимые вычисления на основе отправляемых ему данных и вернуть результат вычисления.

2. Сервер (поставщик услуг) – программное обеспечение, принимающее набор команд, на основание правил собственного API, и производящее их исполнение.

В клиент-серверной архитектуре абсолютно все необходимые вычисления могу быть выполнены на серверном программном и аппаратном обеспечении, которое, как правило, имеет более высокую вычислительную мощность по сравнению с клиентским.

Одним из подходов к тестированию серверного программного обеспечения в клиент-серверной архитектуре являет использование промежуточного прокси-сервера (Рисунок 2).

Рисунок 2 – Промежуточный сервер

Промежуточный сервер (прокси-сервер) выполняет роль посредника, позволяя клиентам осуществлять косвенные запросы (принимать и передавать их через прокси-сервер) к целевому серверу и принимать от него ответы.

Поскольку в процессе работы прокси-сервер пропускает через себя все данные, передаваемые между клиентом и сервером, имеется возможность не только непосредственного их наблюдения, но и модификации. Таким образом промежуточный сервер в общем случае может быть использован для:

1) журналирования сеансов взаимодействия выбранных пар клиентов и серверов,
2) инспекции зашифрованного трафика,
3) имитации проблем в канале связи между клиентом.

При этом, программное обеспечение для промежуточного сервера, предназначенного для решения поставленной задачи, целесообразно разрабатывать на основе одного из существующих прокси-протоколов, поддерживаемых сетевыми клиентами.

1.2 Протокол SOCKS версии 5

SOCKS [1] – сетевой протокол, позволяющий прозрачно обмениваться данными между клиентом и сервером через SOCKS-прокси-сервер. Прокси-серверы, взаимодействующие с клиентами по протоколу SOCKS, обычно производят передачу данных (в обоих направлениях) без их анализа или

модификации. В подобном подходе есть одно важное преимущество для решаемой задачи: сервер не может определить, являются ли поступающие данные переданными напрямую от клиента или они поступают от промежуточных прокси-серверов.

В общих чертах этапы работы SOCKS протокола выглядят следующим образом:

1) клиент подключается к прокси-серверу и посылает приветствие, содержащие поддерживаемые клиентом методы аутентификации;

2) прокси-сервер выбирает один из присланных методов аутентификации или отвечает отказом в случае, если ни один из присланных методов не поддерживается прокси-сервером;

3) в зависимости от выбранного метода аутентификации между клиентом и прокси-сервером происходит обмен данными, содержащими сведения для аутентификации;

4) после успешной аутентификации клиент посылает команду на подключение к целевому серверу со всей необходимой для этого информацией (адрес сервера, порт сервера, тип соединения с сервером);

5) прокси-сервер на основе полученной информации устанавливает связь с целевым сервером;

6) после успешного подключения к целевому серверу прокси-сервер отправляет клиенту сообщение об успехе;

7) цепочка связи успешно установлена; на этом этапе прокси-сервер начинает пересылает через себя данные между клиентом и сервером прозрачно.

В рамках решаемой задачи с помощью создания прокси-сервера на базе протокола SOCKS, мы получаем доступ к данным, передаваемым между клиентом и целевым сервером, и можем осуществить выполнение перечисленных в подразделе 1.1 пунктов по инспекции сетевого трафика, журналированию передаваемых данных и имитации сетевых проблем.

1.3 Журналирование сетевого трафика

Так как прокси-сервер является промежуточным сервером между двумя взаимодействующими узлами, то все данные, которыми обмениваются узлы, проходят через него. Подобно MITM (Man in the middle) атаке [6], прокси-сервер имеет возможность перед отправкой данных не только анализировать и сохранять их на внутренние носители информации или в оперативную память (Рисунок 3), но и модифицировать.

Так как данные, передаваемые между клиентом и сервером, могут быть зашифрованными, то для возможности инспекции и модификации их следует расшифровать. В момент поступления данных от клиента или сервера прокси-сервер расшифровывает поступающую информацию, используя закрытый симметричный ключ, который согласован с отправителем данных в рамках SSL/TLS [7] сессии. После обработки и журналирования данные шифруются ключом, согласованным с получателем этих данных, и посылаются к нему. Механизм установки защищенного соединения между клиентом, промежуточным и целевым сервером, позволяющий получать доступ к незашифрованным данным между клиентом и целевым сервером, описан в главе 1.4.

При приёме данных от одного узла прокси-сервер расшифровывает их (в случае защищенного соединения с узлом) и записывает на носитель информации в расшифрованном виде. После записи данные шифруются (в случае защищенного соединение с узлом-получателем) и перенаправляются на второй узел.

При таком поведение весь трафик, проходящий между клиентом и сервером, может быть сохранен на носитель информации в читабельном для человека виде. Подобная функциональность необходимо для отладки и анализа проблем взаимодействия между клиентом и сервером.

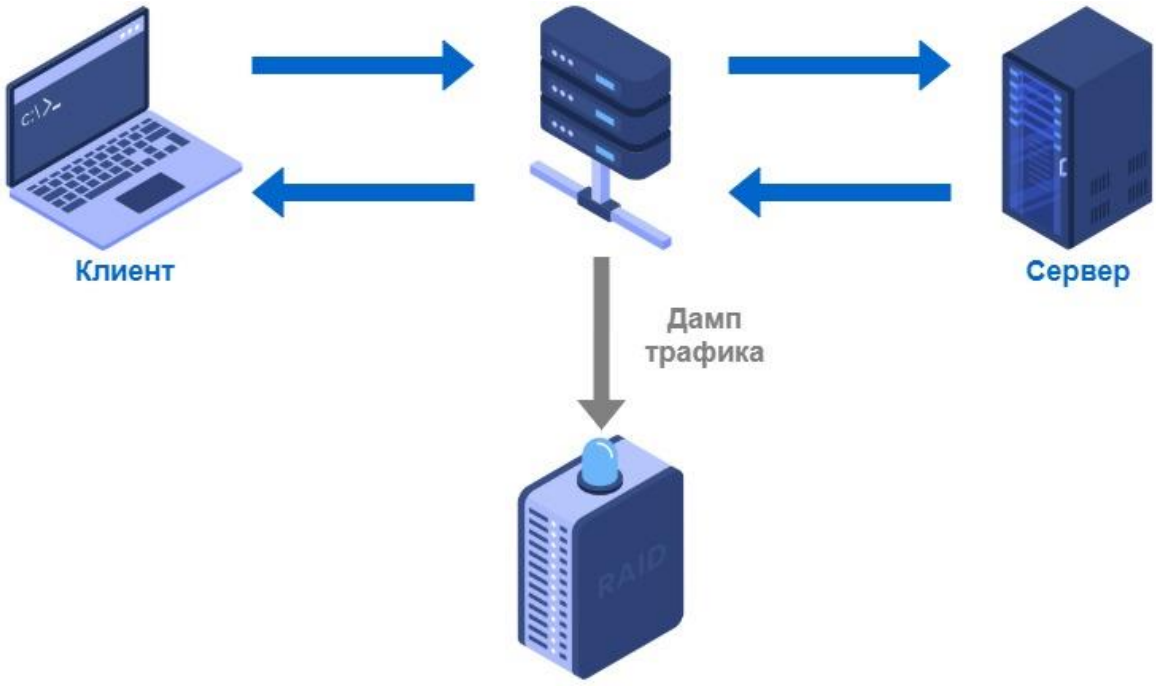

Рисунок 3 – Сохранение информации между клиентом и сервером.

1.4 Инспекция сетевого трафика между клиентом и сервером

Инспекцией сетевого трафика называют возможность просмотра содержимого сетевого трафика, проходящей по каналу передачи данных (в рамках решения поставленной задачи – трафика, проходящего через разрабатываемый прокси-сервер). Если между клиентом и сервером обмен выполняется в зашифрованном виде, то содержимое трафика недоступно.

Для выполнения инспекции трафика в данном случае необходимо реализовать «разрыв» зашифрованного соединения, для чего необходимо дать некоторые пояснения по поводу работы протоколов безопасности SSL/TLS.

Протоколы SSL/TLS являются протоколами прикладного уровня и обеспечивают:

– шифрование данных, передаваемых между клиентом и сервером (для клиента и сервера данный функционал является прозрачным);

– подтверждение аутентичности сервера;

– подтверждение аутентичности клиента (опционально).

Шифрование данных выполняется непосредственно перед отправкой данных в сеть и всегда обеспечивается между двумя конечными точками взаимодействия (клиентом и сервером).

Перевод соединения в режим шифрования данных при помощи протокола SSL/TLS может выполняться как непосредственно после установления соединения по протоколу TCP, так и в любое другое время в процессе работы.

Процесс согласования данных (параметров и ключей шифрования) для выполнения шифрования и проверки аутентичности клиента и сервера называется рукопожатием (handshake). В процессе выполнения рукопожатия стороны вырабатывают сеансовый ключ шифрования, который используют для шифрования/расшифрования передаваемых данных вплоть до разрыва TCP-соединения. Для шифрования полезных данных используются симметричные криптографические алгоритмы. В ходе выработки сеансового ключа и согласования других параметров шифрования используются ассиметричные криптографические алгоритмы.

Проверка аутентичности сервера выполняется при помощи цифрового сертификата, публикуемого сервером, который может быть проверен клиентом с помощью алгоритма проверки цифровой подписи.

Схема установки TLS соединения через обычный прокси-сервер (Рисунок 4) выглядит следующим образом:

1) клиент подключается к прокси-серверу и отправляет ему команду, запрашивающую подключение к целевому серверу;

2) прокси-сервер от своего имени выполняет подключение к целевому серверу и, в случае успеха, отправляет клиенту ответ об успешном подключении;

3) прокси-сервер начинает прозрачно пересылать данные между клиентом и целевым сервером;

4) клиент инициирует процедуру TLS-рукопожатия со стороны клиента;

5) целевой сервер инициирует процедуру TLS-рукопожатия со стороны сервера;

6) выполняется подтверждение аутентичности сервера по его сертификату и соединение переводится в защищенный режим.

Поскольку во время процедуры рукопожатия данные шифруются с помощью методов ассиметричной криптографии, прокси-сервер с этого момента теряет доступ к данным, передаваемым между клиентом и сервером.

Рисунок 4 – Схема работы обычного прокси-сервера

Для возможности инспекции SSL/TLS трафика, в прокси-сервере должен быть реализован специальный режим работы, позволяющий выполнять перехват процедуры рукопожатия. Для его работы необходимо быть уверенным, что он будет использоваться только для сеансов, защищенных протоколом SSL/TLS.

Необходимо реализовать следующий алгоритм работы:

1) клиент подключается к прокси-серверу и отправляет ему команду, запрашивающую подключение к целевому серверу;

2) прокси-сервер от своего имени выполняет подключение к целевому серверу и, в случае успеха, отправляет клиенту ответ об успешном подключении;

3) прокси-сервер от своего имени инициирует процедуру TLS-рукопожатия со стороны клиента;

4) целевой сервер инициирует процедуру TLS-рукопожатия со стороны сервера и предоставляет свой сертификат, подписанный доверенным центром сертификации;

5) выполняется подтверждение аутентичности сервера по его сертификату и вторая часть маршрута, а именно соединение **между прокси-сервером и целевым сервером**, переводится в защищенный режим, при этом данные, приходящие от целевого сервера, будут расшифрованы на стороне прокси сервера;

6) клиент инициирует процедуру TLS-рукопожатия со стороны клиента;

7) прокси-сервер инициирует процедуру TLS-рукопожатия со стороны сервера и предоставляет подменённый цифровой сертификат, выданный также на адрес целевого сервера и подписанный доверенным центром сертификации (как правило внутренний сертификат эксплуатирующей организации);

8) выполняется подтверждение аутентичности сервера по подменённому сертификату и первая часть маршрута, а именно соединение **между клиентом и прокси-сервером**, переводится в защищенный режим, при этом данные, приходящие от клиента, будут расшифрованы на стороне прокси-сервера;

9) прокси-сервер начинает пересылать данные между клиентом и целевым сервером, но при этом ему доступно их содержимое, которое может быть сохранено в файлы журнала или модифицировано.

Схема установления TLS-соединения с включенной инспекцией представлены на рисунке 5.

Рисунок 5 – Схема работы прокси-сервера с включенной инспекцией

Таким образом возможна реализация инспекции трафика даже при использовании между клиентом и сервером протоколов шифрования трафика.

1.5 Имитация проблем связи при сетевом взаимодействии

Работа прокси-сервера сводится к перенаправлению данных от одного узла к другому, что подразумевает удержание сетевого подключения с двумя узлами одновременно. Владея двумя сокетами одновременно прокси-сервер может контролировать и управлять количеством записанной/считываемой информации из данных сокетов. Благодаря этой возможности, внутренний механизм прокси-сервера может имитировать проблемы связи при сетевом взаимодействии, ограничивая количество записываемой информации в сокет или устанавливать свои задержки на чтение информации из сокета.

Ограничением количества данных, принимаемых от узла отправителя и оправляемых к узлу-получателю, достигается имитация низкой пропускной способности канала связи, при котором количество передаваемой информации в секунду искусственно занижено.

На рисунке 6 представлена общая схема искусственного ограничения пропускной способности сети между промежуточным и целевым сервером. Это достигается за счёт медленной записи и медленного чтения информации из сокета данного соединения. Таким образом, используя механизмы ожидания на промежуточном сервере перед началом записи/чтения информации из сокета, можно имитировать нестабильные каналы связи и низкую пропускную способность. Подход к построению подобного механизма с помощью алгоритма «Leaky Bucket» представлен в главе 2.4.

Рисунок 6 – Искусственное ограничение пропускной способности сети между промежуточным и целевым серверами

Используя описанные выше механизмы, можно наблюдать за поведением сервера (или клиента), и в, случае возникновения проблем с сетью и снижением её пропускной способности, можно выполнять тестирование узла на готовность к подобным реальным ситуациям при сетевом взаимодействии.

2 Разработка промежуточного прокси-сервера

Разработка программного обеспечения выполнена на языке C++ с использованием QT фреймворка версии 5. Для работы с сетью использовались классы *QTCPServer*, *QTCPClient*, *QSSLSocket*. Для работы с файловой системой использовались классы *QFile*, *QDir*.

QT фреймворк позволяет реализовать паттерн сигнального взаимодействия между объектами классов: у каждого класса могу быть назначены сигналы и слоты в виде методов класса. Подключения сигналов объекта одного класса к слотам объекта другого класса осуществляется с помощью функции *connect* класса *QObject*, от которого наследуются все остальные классы, поддерживающие паттерн сигнал-слот. Представленная ниже архитектура построения промежуточного сервера в общем объёме реализовано на основе сигнального взаимодействия между объектами.

2.1 Разработка архитектуры прокси-сервера

Для разрабатываемого программного средства была реализована архитектура, основанная на модульном подходе.

Модульная архитектура позволяет легко вносить и изменять поведение системы благодаря представлению внутренних механизмов системы, как программных блоков (модулей). Каждый из программных модулей наследуется от одного общего интерфейса (виртуального класса), который в свою очередь определяет две важных составляющих модуля: точку входа в модуль и точку выхода из модуля (входной и выходной интерфейс модуля). Связывая подряд идущие модули через их точки входы/выхода общая концепция архитектуры программного обеспечения начинает напоминать конвейер, информация в котором, проходя через все модули поочередно, модифицируется или наблюдается каждым из них по отдельности.

Основные компоненты разрабатываемого сервера, представлены на рисунке 7 и описаны далее.

Рисунок 7 – Архитектура промежуточного сервера

Менеджер входящих соединений – модуль, отвечающий за установку соединения с клиентом и перевод его в режим шифрования, если необходимо выполнять инспекцию зашифрованного трафика. В свою реализацию включает серверный модуль, отвечающий за прослушивание серверного TCP-порта и установки соединения, модуль взаимодействия с клиентом по протоколу SOCKS, и конструктор, инициализирующий и связывающий модули 1,2,3 в цепочку.

Исполнительные модули 1, 2 … n – модули, наследующие общий интерфейс, который в свою очередь имеет точку входа и точку выхода для запросов от клиента и ответов от сервера. Исполнительные модули предназначены для наблюдения за данными, проходящими между клиентом и сервером (реализованы модуль журналирования, модуль профилирования трафика)

Менеджер исходящих соединений – модуль, обеспечивающий подключение к целевому серверу и шифрование данных, передающихся между прокси-сервером и целевым сервером при помощи протокола SST/TLS,

если выполняется работа с инспекцией трафика. Создание исходящих соединений выполняется по команде модуля взаимодействия по протоколу SOCKS, входящего в состав менеджера входящих соединений.

При получении запроса менеджером входящих соединений, запрос отправляется во входной интерфейс первого модуля, после его обработки первый модуль отправляет запрос следующему модулю по цепочке через свой выходной и его входной интерфейсы. После прохождения всей цепочки модулей запрос отправляет на сервер менеджером исходящих соединений. При получении ответа на запрос менеджер исходящих соединений отправляет ответ в обратном направление цепочки модулей. Таким образом запросы и ответы на них проходят через всю цепочку модулей, которые могут наблюдать за каждым из них, модифицировать или сохранять.

2.2 Менеджеры соединений

Менеджеры соединений отвечают за установку соединения между клиентом и сервером и, при необходимости, перевод его в режим шифрования данных при помощи протоколов SSL/TLS.

Работа обоих менеджеров проходит следующим образом:

1) клиент подключается к открытому порту менеджера входящих соединений и устанавливает TCP соединение;

2) после установки TCP соединения клиент и менеджер входящих соединений производят обмен пакетами по протоколу SOCKS;

3) после получения от клиента SOCKS-команды на подключение к целевому серверу, в которой присутствуют целевой адрес и порт, менеджер входящих соединений, в свою очередь, передает команду менеджеру исходящих соединений на создание новой TCP-сессии с целевым сервером;

4) менеджер исходящих соединений открывает TCP соединение с указанным сервером; после установки соединения, если включена опция инспекции, менеджер инициирует клиентскую процедуру TLS-рукопожатия с сервером;

5) после установления TCP-сеанса с сервером и перевода его в режим шифрования менеджер исходящих соединений сообщает об этом менеджеру входящих соединений.

6) менеджер входящих соединений сообщает клиенту об успешном подключение к серверу, использую ответ согласно протоколу SOCKS, после чего, в случае активной инспекции, инициирует с клиентом серверное TLS-рукопожатие, используя при этом доверенный сертификат;

7) на этом этапе соединение с клиентом и сервером успешно установлено с двух сторон; далее содеются экземпляры классов модулей обработки и все данные передаваемые в каждом из направлений направляются через модули обработки.

При получении данных со стороны клиента они расшифровываются и направляются первому модулю в цепочке. При получении ответа от первого модуля в цепочке менеджер выполняет их шифрование и отправляет их клиенту.

При получении запроса от последнего модуля обработки в цепочке, менеджер исходящих соединений шифрует данные и отправляет серверу. При получении ответа от сервера менеджер расшифровывает данные в ответе и отправляет последнему модулю в цепочке.

2.3 Модуль журналирования

Модуль журналирования в своей реализации использует классы QT фреймворка для работы с текстовыми файлами и директориями.

После получения запроса модуль получает из запроса адрес/порт отправителя и адрес/порт получателя. На основе этих данных модуль создаёт каталог на носителе и открывает текстовый файл с именем адреса/порта получателя и временем установления сеанса связи. Все данные запроса модуль записывает в открытый файл и отправляет запрос далее по цепочке модулей.

При получении ответа модуль записывает данные ответа в открытый файл (тем самым дополняя его) и отправляет ответ предыдущему модулю в цепочке модулей.

После закрытия сеанса связи модуль сбрасывает буферизированные данные в файл и закрывает его, освобождая дескриптор.

Таким образом, в каталоге создаются каталоги с именем хостов/портов клиентов, от которых приняты запросы, а в каталогах сохраняются текстовые файлы для каждого соединения, в которых содержаться данные проведенных TCP-сессий.

2.4 Модуль имитации проблем связи при сетевом взаимодействии. Алгоритм «Leaky Bucket».

Сам по себе модуль имитации проблем связи не входит в цепочку модулей системы, а является частью реализации менеджера исходящих или менеджера входящих соединений, так как работает непосредственно с сокетом соединения. Для реализации механизма имитации медленных каналов передачи данных и канальных проблем разновидность алгоритма "Leaky Bucket" ("протекающее ведро"). Программная реализация использует алгоритм "протекающего ведра" для ограничения суммарной полосы пропускания в каждом направлении, равномерного её распределения между всеми активными TCP-сессиями.

Рассмотрим подключение от клиента к ресурсу, состоящее из n соединений. Каждое соединение проходит через прокси-сервер, где оно анализируется и изменяется (рисунок 8).

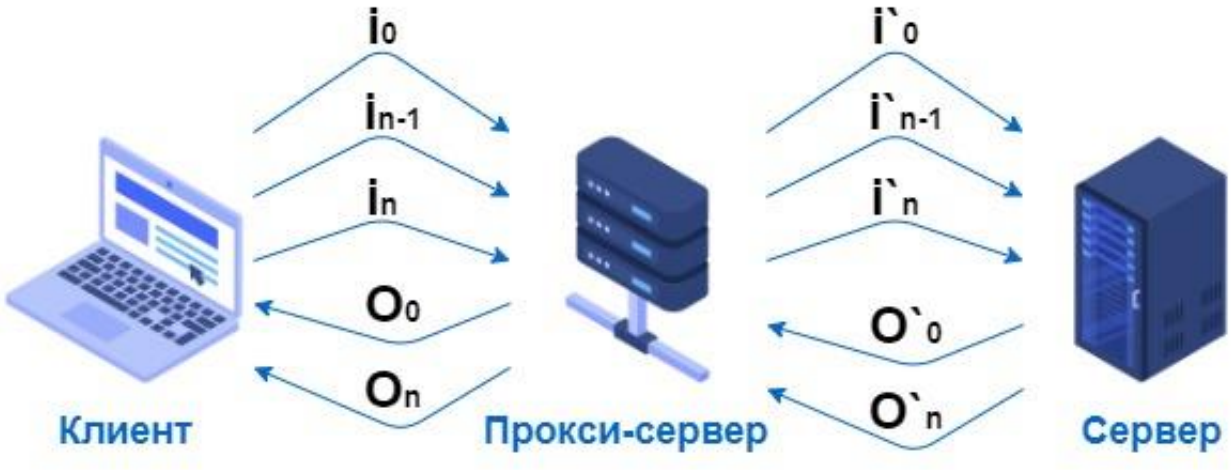

Рисунок 8 – Принцип работы имитации проблем связи.

На изображении:

$i_0, i_1, \dots, i_n$ – трафик от клиента до прокси-сервера;

$i'_0, i'_1, \dots, i'_n$ – трафик от прокси-сервера до целевого сервера;

$o_0, o_1, \dots, o_n$ – трафик от прокси-сервера до клиента;

$o'_0, o'_1, \dots, o'_n$ – трафик от целевого сервера до прокси-сервера.

Весь трафик, приходящий через соединения $i_1, i_2, \dots, i_n$ и $o'_1, o'_2, \dots, o'_n$ поступает в соответствующие буферы (по одному на каждое направление каждого TCP-соединения). С определенным интервалом (в текущей реализации используется значение 100 миллисекунд), модуль анализирует количество данных, находящихся в каждом буфере, и отправляет данные серверу через соединения $i'_1, i'_2, \dots, i'_n$ и $o_1, o_2, \dots, o_n$ так, чтобы суммарное количество передаваемого от прокси-сервера трафика находилось в границах установленной полосы пропускания. Отправка данных привязывается всегда к сигналу таймера (100 милисекунд).

Между соединениями распределения размеров отправляемых блоков данных выполняется по алгоритму, описанному ниже.

Обозначим:

$buffers$ – структура данных, хранящая буферы каждого из соединений;

$capacity$ – объем трафика, которое требуется посылать каждый интервал времени (100мс);

$connections$ – текущие соединения с сервером.

Выполнение алгоритма повторяется через равные промежутки времени (например, 100 мс). В каждый такой интервал времени ведется расчет количества данных, которые необходимо отправить в каждое из установленных соединений.

Псевдокод алгоритма:

```
void sendData() {
     buffers.sort();
     int i = 0;
     foreach (QSSLSocket connection: clientConnections) {
          int sendLength = capacity / (buffers.size() + i);
          QByteArray buffer = buffers.get(connection);
          int sendFact = min(buffer.size(), sendLength);
          connection.write(buffer, sendFact);
          buffer.removeFirst(sendFact);
          capacity -= sendFact;
          i += 1;
     }
}
```

Рассмотрим пример:

Допустим, имеется 3 соединения, по которым принимается трафик от клиента. В текущий момент в приёмных буферах первого соединения находится 100 кбайт, второго соединения 150 кбайт и третьего соединения – 200 кбайт. В данном примере для наглядности предполагаем, что больше данные от клиента не поступают. Подключение от клиента к прокси-серверу работает на полную пропускную способность, подключение от proxy до сервера ограничено производительностью 60 кбайт за 100 мс (600 кбайт/с). Тогда данные от клиента к серверу будут приходить следующим образом (Таблица 1).

Таблица 1 – Пример отправки данных при работе алгоритма

Время с начала отсчета, мс	**В буфере / отправлено**	**Номер соединения / трафик, кбайт**		
		№ 1, данные	**№2, данные**	**№3, данные**
0	в буфере	100	150	200
	отправлено	20	21	21
100	в буфере	80	129	179
	отправлено	21	21	22
200	в буфере	59	108	157
	отправлено	21	22	22
300	в буфере	38	86	135
	отправлено	20	20	20
400	в буфере	18	66	115
	отправлено	18	23	23
500	в буфере	0	43	92
	отправлено	0	30	31
600	в буфере	0	13	61
	отправлено	0	13	49
700	в буфере	0	0	12
	отправлено	0	0	12
800	в буфере	0	0	0
	отправлено	0	0	0

Таким образом, скорость передачи данных от прокси-сервера до целевого сервера будет колебаться в районе от 600 кбайт/с.

Используя функцию *write* класса *QSSLSocket* можно писать данные в сокет частями, контролируя тем самым скорость потока записываемых данных.

Для генерации сигналов, к которым привязывается вызов функции отправки очередной порции данных класс *QTimer*.

Для имитации проблем с каналами связи случайным образом соединений выполняется пропуск одного или нескольких циклов отправки данных по таймеру.

Наблюдая за поведением сервера и клиента при имитации проблем связи, можно наблюдать как серверное или клиентское программное обеспечение ведёт себя при реальных подобных проблемах с сетевым каналом связи.

3 Анализ результатов и тестирование

Во время выполнения данной работы был разработано программное средство - прокси-сервер, работающий по протоколу SOCKS версии 5, имеющий следующую функциональность:

1) журналирование трафика между клиентом и сервером для дальнейшего анализа и отладки проблем сетевого взаимодействия;
2) инспекция зашифрованного сетевого трафика (SSL/TLS-сессий);
3) контроль пропускной способности канала связи между клиентом и сервером с возможностью имитации нестабильного соединения и слабой пропускной способности;

Тестирование работоспособности разработанного программного средства (прокси-сервера) выполнялось путем организации выгода через него следующего программного обеспечения:

- интернет-бразуера;
- почтового клиента Mozilla Thunderbird;
- FTP/SFTP клиент Filezilla.

Возможность работы существующих программных средств с разработанным прокси-сервером подтверждает корректность реализации протокола SOCKS и процедуры инспекции трафика.

Во время выполнения данной работы был изучен принцип работы прокси-серверов, принцип работы проксирования, протокол SOCKS версии 5, принципы работы протоколов SSL/TLS и предоставляемые Qt фреймворком инструменты для работы с сетевыми соединениями.

3.1 Ручное тестирование

Для тестирования программного средства был использован веб-браузер «Mozilla Firefox» версии 71.0, предоставляющий настройки проксирования через указанный в виде адреса и порта SOCKS-прокси-сервера с возможностью аутентификации по логину и паролю или без аутентификации.

Во время тестирования веб-браузер успешно установил сеанс связи с прокси-сервером и использовал данный сеанс для выхода в сеть интернет.

Для тестирования логирования, API и имитации проблем сети использовалось отдельно разработанное серверное программное обеспечение с готовым набором API правил и схемами корректного поведения.

Серверное программное обеспечение успешно прошло тестирование с помощью разработанного прокси-сервера, успешно были получены журналы сеансов связи с сервером и протестировано поведение сервера при возникновении сетевых проблем.

Заключение

В процессе выполнения настоящей дипломной работы:

– выполнено исследование подходов к обеспечению тестирования и отладки сетевого взаимодействия в программно-аппаратных системах, построенных на принципах клиент-серверной архитектуры;

– изучены детали реализации протокола SOCKS версии 5 и принципы работы прокси-серверов, функционирующих по протоколу SOCKS;

– изучены принципы работы протоколов шифрования сетевого трафика SSL/TLS, широко применяемых в сети Интернет;

– изучены методы обеспечения «разрыва» канала передачи данных, создаваемого протоколами SSL/TLS;

– разработана модульная архитектура прокси-сервера;

– изучены базовые и сетевые возможности фреймворка Qt и принципы асинхронного сетевого взаимодействия;

– разработано программное средство – SOCKS-прокси-сервер, предназначенное для тестирования и отладки сетевого взаимодействия клиент-серверных приложений;

– проведено тестирование разработанного программного средства.

Благодаря использования асинхронных подходов для организации сетевого взаимодействия прокси-сервер обеспечивает:

– высокую сетевую производительность даже при работе в однопоточном режиме;

– возможность одновременной работы с множеством (до 400-600) сетевых соединений с включенной инспекций и управлением скоростью передачи.

Модульная архитектура разработанного программного средства модуля позволяет удобно выполнять расширение его функционала без изменения архитектуры, поскольку каждый модуль получает доступ к проходящим в

прямом и обратном направлении данным, и обеспечивает упрощённую поддержку.

Список использованных источников

1. RFC 1928 «SOCKS Protocol Version 5». — URL: https://tools.ietf.org/html/rfc1928 (дата обр. 27.02.2020).
2. Qt: Signals & Slots. — URL: https://doc.qt.io/qt-5/signalsandslots.html (дата обр. 18.03.2020).
3. Network Programming with Qt. — URL: https://doc.qt.io/qt-5/qtnetwork-programming.html (дата обр. 20.03.2020).
4. Proxy server. — Wikipedia. — https://en.wikipedia.org/wiki/Proxy_server (дата обр. 20.04.2020).
5. SOCKS. — Wikipedia. — https://en.wikipedia.org/wiki/SOCKS (дата обр. 20.04.2020)
6. Man-in-the-middle attack. — Wikipedia. — https://en.wikipedia.org/wiki/Man-in-the-middle_attack (дата обр. 13.05.2020)
7. RFC 8446 «The Transport Layer Security (TLS) Protocol Version 1.3». — URL: https://tools.ietf.org/html/rfc8446 (дата обр. 13.05.2020).