Санкт-Петербургский государственный университет

Кафедра системного программирования
Группа 22.Б11-мм

Улучшение окружения кода miminet для дальнейшего масштабирования

Зайцев Дмитрий Сергеевич

Отчёт по производственной практике

Научный руководитель:
старший преподаватель СПбГУ Зеленчук И. В.

Санкт-Петербург
2025

Оглавление

Введение

Miminet [1] [2] — веб-тренажёр, предназначенный для обучения основам компьютерных сетей. Он позволяет создавать виртуальные компьютерные сети, настраивать устройства и эмулировать процессы сетевого взаимодействия.

Проект активно развивается, на данный момент курс «Компьютерные сети» [3], который читается на математико-механическом факультете Санкт-Петербургского государственного университета, основан на примерах из этой обучающей платформы. Miminet также используется для проверки знаний в области компьютерных сетей. Например, при промежуточном тестировании студентов курса или при отборе стажёров на «YADRO ИМПУЛЬС 2023-2025».

При этом разработка приложения не стоит на месте. Каждый семестр в Miminet приходят новые студенты, желающие получить опыт в работе над реальным проектом. Каждый из них добавляет новую функциональность, либо расширяет уже имеющуюся. Изменения в уже написанный код также вносятся, например, для исправления найденных ошибок, уязвимостей или просто для улучшения производительности и читаемости кода, повышения его масштабируемости и адаптации к новым требованиям. Например, в осеннем семестре разработкой занимались пять новых человек, и суммарно в проект было добавлено свыше пяти тысяч строк кода.

Однако процесс разработки усложняется отсутствием полноценного тестирования приложения. Каждое внесение изменений всегда сопряжено с риском повреждения уже работающего кода. При этом проверить новую функциональность на безопасность можно только вручную, самостоятельно используя каждую функцию в интерфейсе.

Из этого можно сделать вывод, что отсутствие полноценного автоматического тестирования создаёт лишние риски и неудобства. Ручная проверка каждой функции трудоёмка (от пяти минут до получаса на одно изменение, включая построение сетей, их настройку, проверку отдельных кнопок) и не гарантирует полного выявления проблем. Поэтому разработка метода автоматического end-to-end тестирования Miminet является важной задачей для обеспечения стабильности и надёжности приложения, а также для ускорения процесса разработки и снижения рисков, связанных с внесением изменений в код.

Задачи

Целью работы является создание метода автоматического end-to-end тестирования Miminet.

В ходе работы были поставлены следующие **задачи**:

1. Обзор технологий.
2. Сбор требований.
3. Создание первых тестов на ключевые функции Miminet.
4. Добавление возможности локального запуска тестов.
5. Интеграция с CI.

Обзор

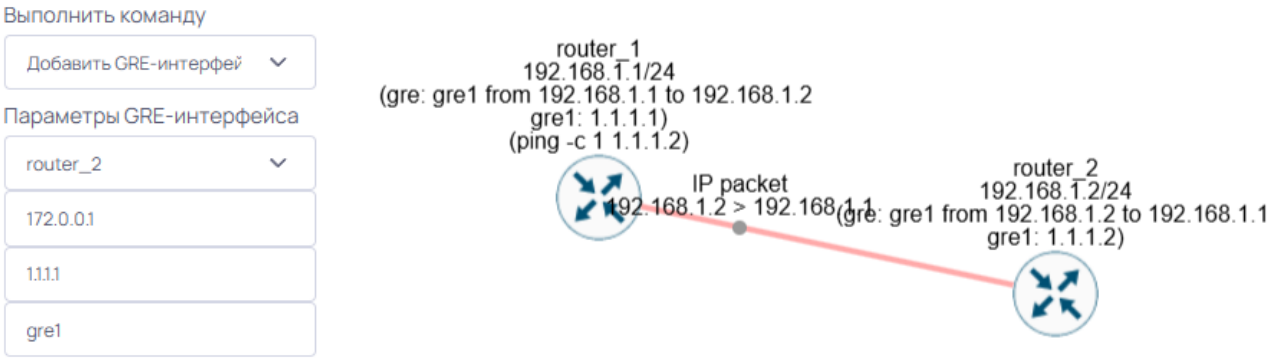
2.1 Добавление GRE

В связи с обновлением структуры курса «Компьютерные сети» и добавлением в него раздела «Туннелирование», появилась необходимость добавить в проект поддержку GRE-туннелей.

GRE [4] (*Generic Routing Encapsulation*, *Общая Инкапсуляция Маршрутов*) — это протокол туннелирования, обеспечивающий инкапсуляцию пакетов сетевого уровня (например, IPv4, IPv6, IPX, AppleTalk и так далее) в IP-пакеты для передачи через IP-сети. Данный протокол решает основную проблему IPIP (IPIP туннель может инкапсулировать только IPv4 пакеты) и чаще используется на практике для построения туннелей.

Чтобы добавить поддержку GRE-туннелирования в Miminet, необходимо было создать форму для конфигурации роутера и добавить для его эмулятора необходимые настройки.

Результат выполненной работы представлен ниже:



(a) Панель настройки GRE.

(b) Пример сети.

Рис. 1: Реализация GRE.

2.2 Обновление архитектуры

В процессе реализации описанной выше функциональности был обнаружен небезопасный фрагмент с избыточным использованием повторяющегося (дублированного) кода. Эта часть проекта была предназначена для конфигурации сетевых устройств и проверки передаваемых им параметров на стороне фронтенда. Основными выявленными проблемами были: отсутствие целостной структуры, неравномерность проверок для входных данных и несогласованность сообщений об ошибках.

В ходе работы была проанализирована старая структура кода, были выявлены её слабые места, а также разработана полностью новая архитектура, позволяющая добавлять новую

функциональность (описывать методы конфигурации сетевых устройств, добавлять проверки на них) максимально просто, при этом не допуская прежних уязвимостей.

Ниже представлена диаграмма классов. На ней хорошо видно, как с помощью наследования получилось решить проблему дублирующегося кода и повысить связность.

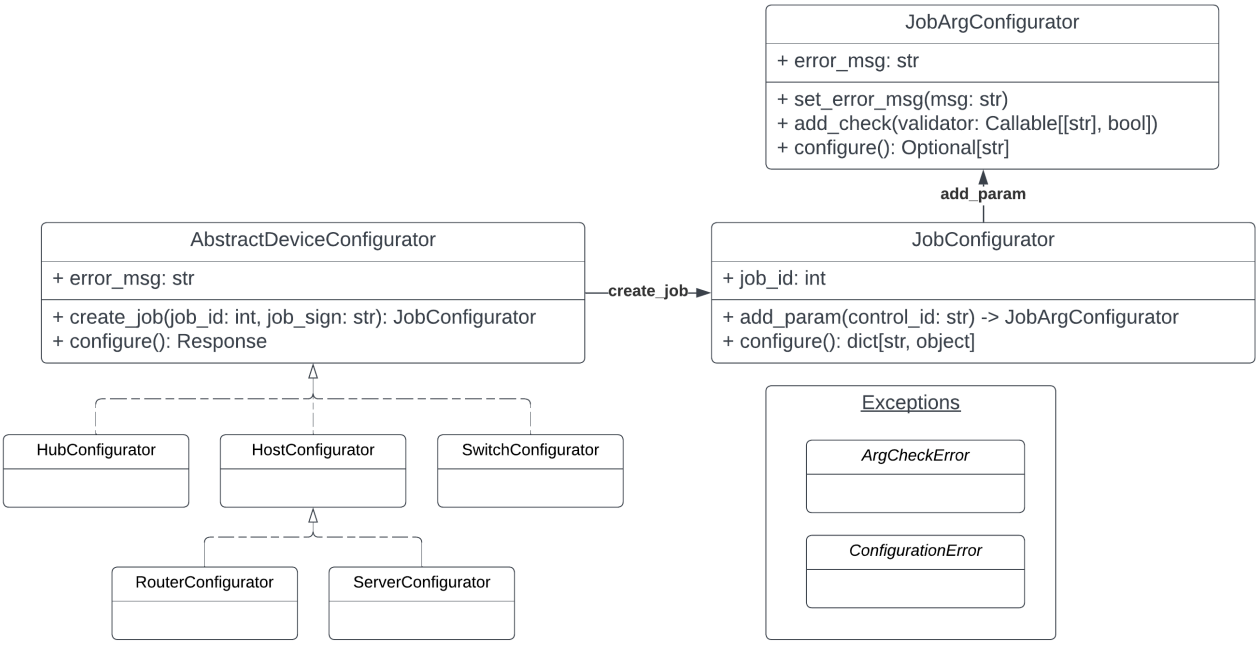


Рис. 2: Обновлённая архитектура.

Также удалось упростить процесс добавления новых параметров конфигурации. До обновления архитектуры для выполнения тех же действий приходилось вручную разбираться в структуре всего файла (около двух тысяч строк кода), а затем копировать подходящий участок и вставлять его, внося некоторые изменения под свой конкретный случай. Результатом таких преобразований становились: плохая читаемость кода, сложности в его обновлении и поддержке, а также некорректная обработка неправильно введённых пользователем данных.

Сейчас для добавления новой функциональности достаточно ознакомиться с примером ниже:

Алгоритм 1 Добавление новых параметров конфигурации устройства.

```
# ping -c 1 (1 param)
host_ping_job = host.create_job(1, "ping -c 1 [0]")
host_ping_job.add_param("config_host_ping_c_1_ip")\
    .add_check(IPv4_check)\
    .set_error_msg(build_error(ErrorType.ip, "ping"))
```

На нём можно понять, как переиспользуются методы проверки корректности введённых данных, а также как происходит генерация сообщений об ошибках. Кроме того, общее количество строк в файле уменьшилось почти в два раза, что значительно облегчило процесс добавления новых элементов и проверок.

2.3 Подробное описание проблемы

Чтобы обосновать необходимость в системе для автоматического тестирования, приведу несколько примеров из своей собственной практики.

Через неделю после обновления архитектуры пользователи начали находить ошибки в работе Miminet. Например, из-за пропущенного оператора *break* кнопка «Добавить сабинтерфейс с VLAN [5]» перестала показывать панель конфигурации. Точно так же нашли ещё одну проблему: проверка корректности введённых TCP [6] и UDP [7] портов давала сбой при числах, больше тридцати двух. Причина была в том, что при переносе настроек в новую версию проекта один из параметров был перепутан, и вместо проверки порта выполнялась проверка маски подсети.

Хотя устранение неполадок заняло не более получаса, всё то время, в течение которого сервис функционировал некорректно, причинило пользователям значительные неудобства. Кроме того, выявление этих проблем могло затянуться ещё сильнее, если бы пользователи сами не уведомили о них.

Подобных ситуаций можно было бы легко избежать, если бы в Miminet была полноценная система для end-to-end тестирования.

2.4 Обзор технологий

Архитектура Miminet состоит из фронтенда (веб-интерфейс и серверная часть для работы с базой данных) и бэкенда (эмулятор сетей Mininet). Фронтенд взаимодействует с бэкендом через брокер сообщений RabbitMQ [8]. Каждая отдельная часть проекта запускается в своём docker-контейнере, что позволяет избежать проблем с зависимостями, а также обеспечивает необходимый уровень изоляции и защиты компонентов приложения.

Для реализации системы автоматического тестирования необходимо было найти подходящую технологию для имитации работы пользователя с интерфейсом приложения. В результате анализа был выбран **Selenium** [9] [10], инструмент для автоматизации действий веб-браузера. Он поддерживает большинство современных браузеров: Chrome, Firefox, Safari, Edge и имеет реализацию в таких языках, как Java, Python (на котором написана большая часть Miminet), C#, JavaScript, Ruby. Кроме того, Selenium получает регулярные обновления и поддерживается уже двадцать лет, а также имеет исчерпывающую документацию и активное сообщество разработчиков. Среди аналогов (таких как **Cypress** [11]) инструмент выделяется своей простотой, стабильностью, поддержкой в популярных языках, а также возможностью параллельного запуска тестов и лёгкой интеграцией в CI/CD.

Для реализации возможности локального запуска тестов были использованы **Docker** [12] (система контейнеризации, позволяющая упаковывать приложения и их зависимости в изолированные контейнеры) и **Docker Compose** [13] (инструмент для определения и управления многоконтейнерными приложениями Docker). Также был применён **Selenium Grid** [14] – расширение Selenium, позволяющее распределять запросы по нескольким машинам (или контейнерам) для параллельного запуска тестов и одновременного тестирования приложения на разных версиях веб-браузеров.

Требования к тестирующей системе

Перед разработкой тестирующей системы был определён список требований. Система для тестирования Miminet должна:

- уметь выполнять автоматические тесты, имитирующие реальные сценарии использования проекта,
- обеспечивать возможность локального запуска тестов без ручной настройки среды и управления зависимостями,
- работать только в специальном режиме приложения для тестирования,
- быть легко адаптируемой к изменениям проекта (простое добавление и обновление тестов),
- интегрироваться с CI/CD, автоматически запускаясь при изменении кода и предоставляя наглядные результаты его проверки.

Реализация

4.1 Архитектура

Для реализации возможности локального запуска тестов в проект были добавлены два новых docker-контейнера: Selenium-hub и Chrome. Первый контейнер [15] – это центральный узел в Selenium Grid, который управляет всеми узлами (nodes) и распределяет задачи между ними. С его помощью можно параллельно запускать тесты или выполнять проверку Miminet сразу на разных версиях веб-браузеров. Второй контейнер [15] – узел в Selenium Grid, который запускает веб-браузер (Chrome) и выполняет в нём тесты, имитируя действия пользователя. Новые контейнеры объединены в сеть miminet_network, в которой также находится фронтенд приложения, к которому отправляются все запросы.

На данный момент все тесты запускаются только на одной версии браузера и параллельность не используется, однако архитектура допускает простое масштабирование в случае необходимости.

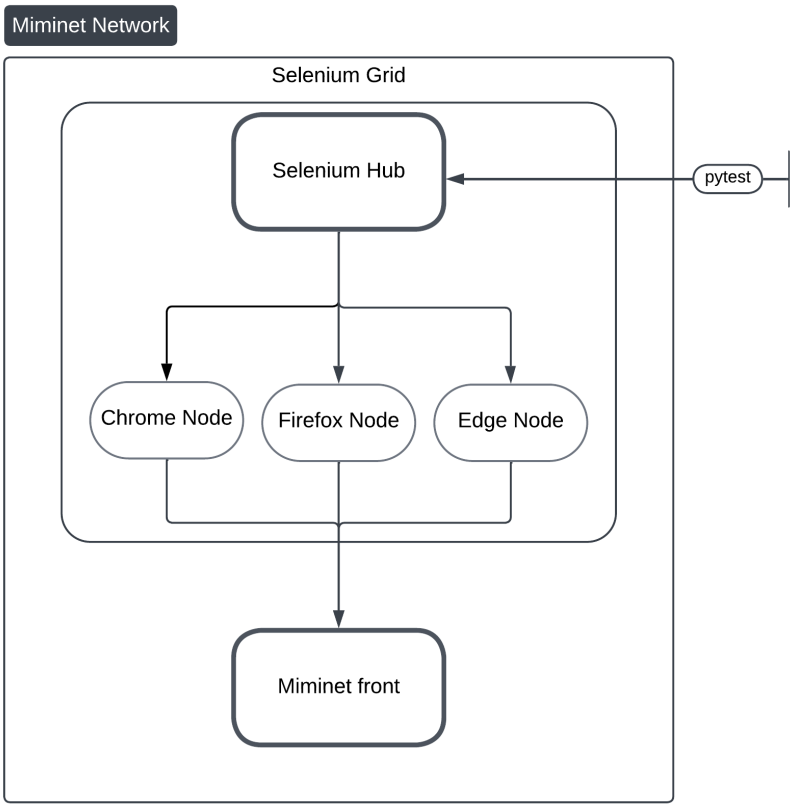


Рис. 1: Взаимодействие между прототитестирующей системой и приложением.

Специально для тестирования, в Miminet был добавлен «режим разработки» и «рабочий режим». Их главное отличие в том, что в «режиме разработки» тестирующая система может

свободно пользоваться Miminet, проходя авторизацию через специально внесённые в базу данных параметры. При этом в «рабочем режиме», в котором приложение запускается по умолчанию, база данных не содержит таких параметров для входа. Сделано это было с целью обеспечения безопасности рабочей среды и предотвращения несанкционированного доступа к системе.

4.1.1 Прототип

Как было заявлено в требованиях, процесс создания новых тестов и изменения старых должен быть достаточно удобным, чтобы новые разработчики не тратили на это лишнее время. С этой целью было принято несколько архитектурных решений, благодаря которым появилась возможность писать тесты на конфигурацию сетей на более высоком уровне абстракций. Схема того, как это было реализовано в первой версии тестирующей системы, приведена ниже:

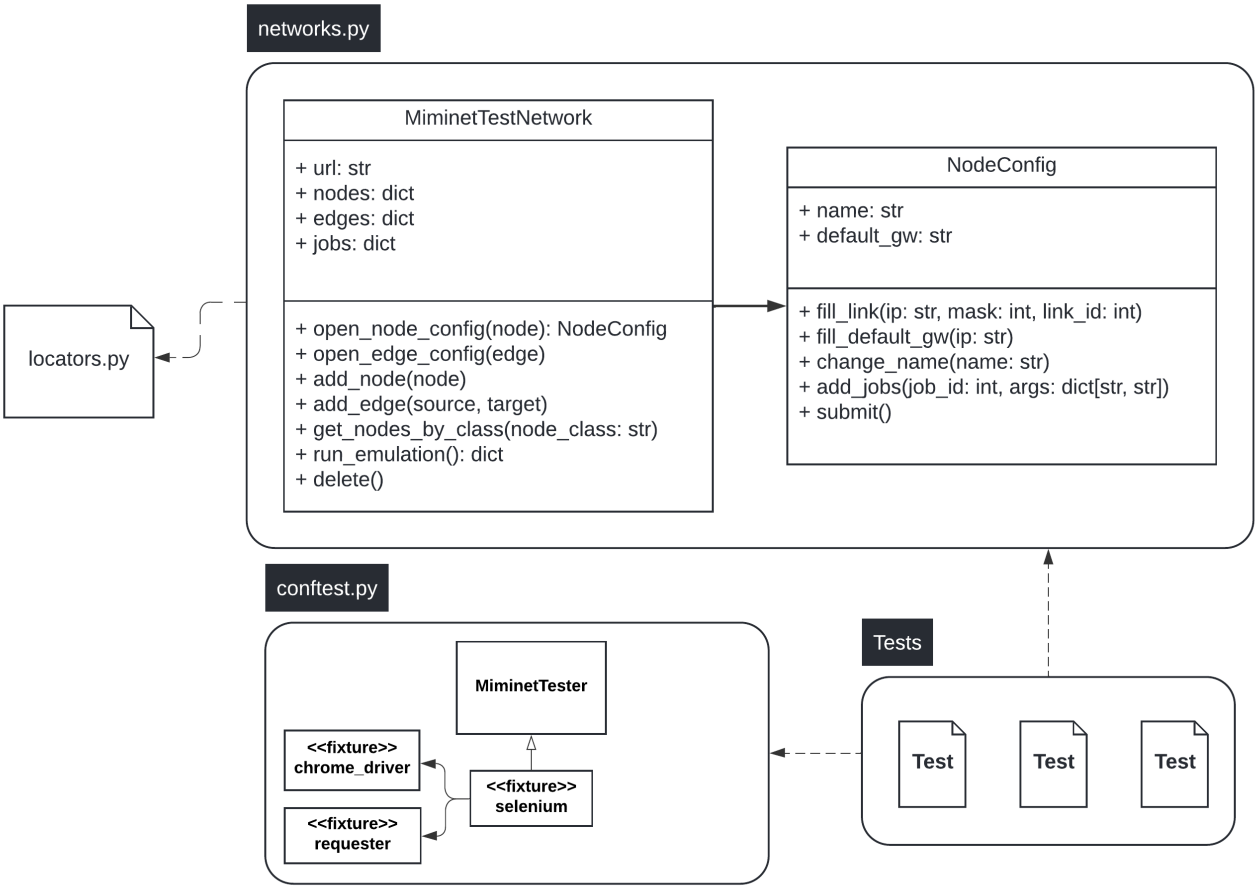


Рис. 2: Архитектура тестирующей системы.

В файле networks.py описаны абстракции, предназначенные для простой конфигурации сетей. Благодаря их использованию, код тестов не зависит от изменений в Miminet. Кроме того, время, необходимое на написание одного теста, сокращается до пяти-десяти минут за счёт эффективного переиспользования кода.

Фикстуры, определённые в conftest.py, используются в каждом тесте, они позволяют взаимодействовать с необходимыми функциями selenium, а также использовать специализированные методы для работы с Miminet (определённые в MiminetTester).

Файл locators.py представляет собой удобно структурированный набор констант, к которому обращаются элементы тестирующей системы при необходимости найти что-либо на

странице. В случае изменений в структуре Miminet, константы можно будет легко заменить на другие, не меняя сам код тестов.

4.1.2 Итоговая версия

После того, как была получена первая версия тестирующей системы, а также написаны тесты для самых базовых случаев применения Miminet, появилась потребность в покрытии более сложных случаев. Чтобы упростить дальнейшее написание тестов, был проведён небольшой рефакторинг. Схема того, как выглядит финальная версия тестирующей системы, приведена ниже:

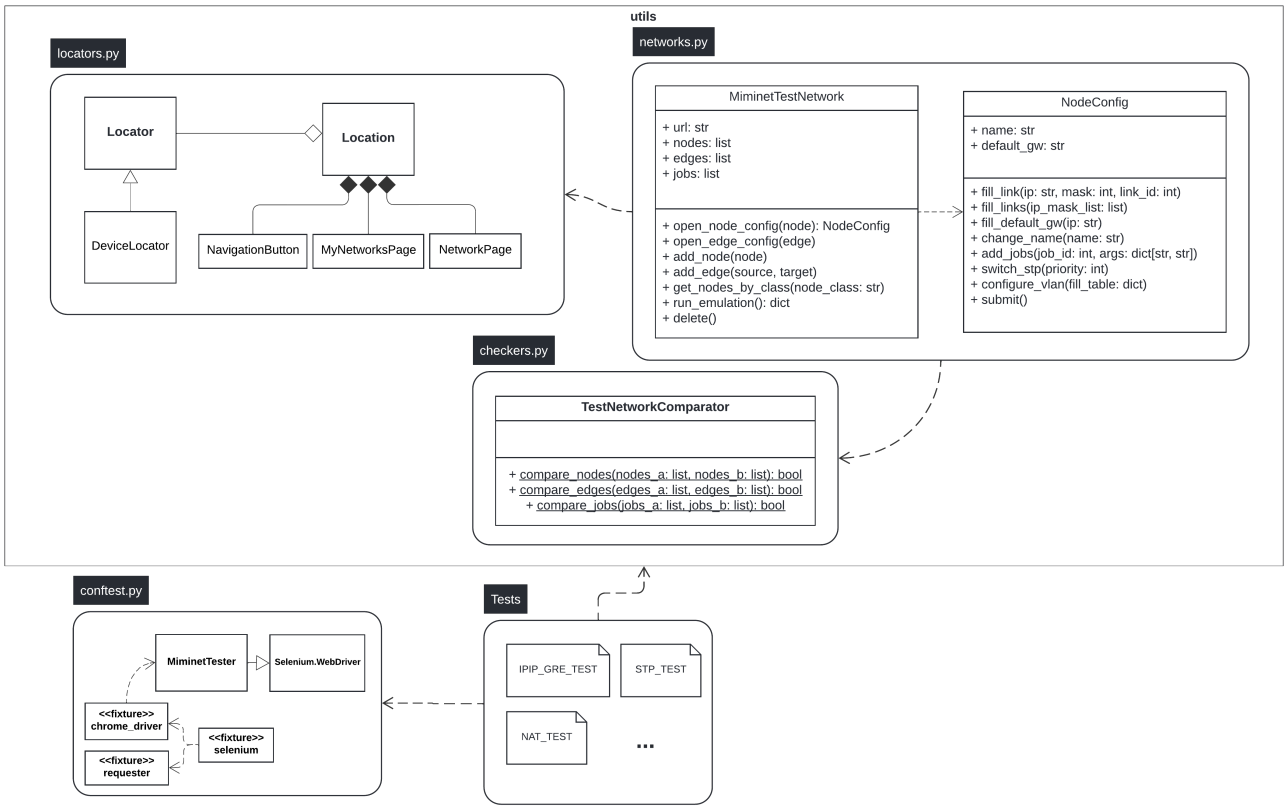


Рис. 3: Улучшенная версия архитектуры тестирующей системы (детально).

В ходе рефакторинга было внесено несколько ключевых изменений.

Во-первых, логика сравнения компонентов построенных сетей была инкапсулирована в отдельный класс и вынесена в самостоятельный модуль. Это позволило сократить объём кода в основном классе MininetTestNetwork и изолировать проверки сетей от их построения в тестах.

Во-вторых, серьёзные изменения были внесены в модуль locators, который теперь представляет собой иерархию классов, а не набор констант-словарей. Такая структура обеспечивает более удобный и наглядный доступ к идентификаторам элементов на страницах.

Кроме того, появились новые функции, упростившие написание тестов — например, поддержка контекста модальной формы или возможность получить идентификаторы устройств при их создании.

Наконец, класс NodeConfig был дополнен новыми методами для работы с VLAN и (R)STP. Это позволило избежать переноса соответствующей логики в тесты, повысив читаемость и переиспользуемость кода.

4.2 Тесты

На данный момент реализованы следующие тесты:

1. Проверка доступности ключевых страниц сайта.
2. Создание пустой сети.
3. Проверка навигации в меню выбора сетей.
4. Размещение устройств в сети.
5. Изменение имён устройств.
6. Создание связей между устройствами.
7. Конфигурация простой сети, использующей задачу «ping».
8. Конфигурация сетей, использующих протоколы туннелирования «IPIP»/«GRE».
9. Конфигурация сети с «NAT».
10. Конфигурация сетей, использующих протоколы «TCP»/«UDP».
11. Создание кольца из 3-х маршрутизаторов.
12. Настройка «STP» и «RSTP».
13. Конфигурация сети с «VLAN».

При создании и настройке сетей тестирующая система имитирует действия пользователя. При этом результат сверяется с заданным заранее шаблоном. Таким образом, гарантируется корректность работы приложения (возможность создавать аналогичные сети). Ниже приведён пример такой сети, а также описан процесс её тестирования и конфигурации.

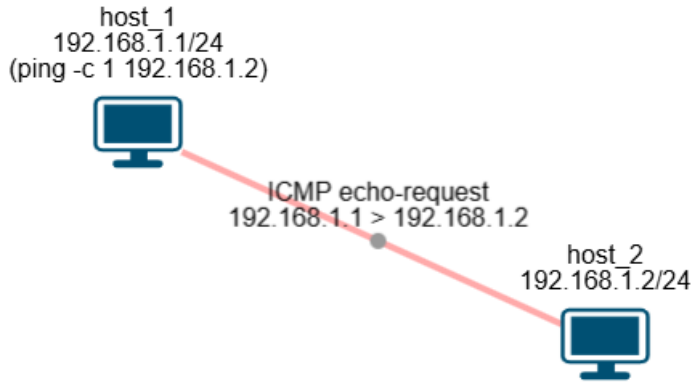


Рис. 4: Сеть, построенная тестирующей системой.

Алгоритм 2 Конфигурация и тестирование простой сети.

```
@pytest.fixture(scope="class")
    def network(self, selenium: MiminetTester):
        # Create a new network
        network = MiminetTestNetwork(selenium)

        # Add network devices
        network.add_node(NodeType.Host)
        network.add_node(NodeType.Host)

        # Add edges between them
        network.add_edge(source_id=0, target_id=1)

        # Configure first host
        config0 = network.open_node_config(node_id=0)
        config0.fill_link("192.168.1.1", 24)
        config0.add_jobs(1, {host_job_ip_selector: "192.168.1.2"})
        config0.submit()

        # Configure second host
        config1 = network.open_node_config(1)
        config1.fill_link("192.168.1.2", 24)
        config1.submit()

        # Return configured network
        yield network

        network.delete()

    def test_ping(self, selenium: MiminetTester, network: MiminetTestNetwork)
        # Compare network's data with templates
        assert compare_nodes(network.nodes, self.JSON_NODES)
        assert compare_edges(network.edges, self.JSON_EDGES)
        assert compare_jobs(network.jobs, self.JOBS)
```

Масштабирование и улучшение кода Miminet

5.1 Переход на PostgreSQL

Первоначально в Miminet в качестве СУБД использовалась SQLite [16] — легковесное и простое в использовании решение, хорошо подходящее для прототипов и небольших проектов. Однако с ростом числа пользователей и количества сетей, объём данных возрос до ∼1 ГБ. Производительность запросов начала снижаться, а также возникла потребность переноса приложения на новый диск из-за ограничений в объёме старого. Это стало удобным моментом для модернизации архитектуры и основной причиной перехода на более быстрое и масштабируемое решение PostgreSQL [17].

Прежняя реализация доступа к базе данных на SQLite была завязана на использовании Volume, подключённого непосредственно к основному контейнеру приложения, что ограничивало гибкость и мешало масштабируемости. Поэтому было принято решение развернуть PostgreSQL в отдельном контейнере, чтобы изолировать базу данных от логики приложения, а в перспективе упростить бэкап, восстановление данных и подготовить систему к горизонтальному масштабированию.

5.1.1 Отделение компонента БД

В рамках улучшения архитектуры проекта было принято решение отделить компонент базы данных от основного контейнера приложения и перенести его в отдельный сервис внутри `docker-compose`. Для работы был взят готовый официальный образ PostgreSQL [18], который был добавлен в конфигурационный файл `docker-compose.yml`. Основные параметры конфигурации контейнера были вынесены в отдельный файл `.env`, чтобы упростить настройку и повысить читаемость файлов конфигурации. Также для контейнера был настроен `healthcheck`, обеспечивающий автоматическую проверку готовности сервиса к работе.

На уровне приложения была изменена логика подключения к базе данных: теперь оно подключается к контейнеру PostgreSQL через внутреннюю сеть docker-compose, объединяющую все сервисы. Кроме того, была упрощена инициализация базы данных: если ранее для этого требовалось явное указание параметра `init` при старте приложения, то теперь оно автоматически определяет, нужно ли создать новую схему или использовать уже существующую. Это сделало процесс запуска miminet чуть более понятным.

5.1.2 Подготовка модели

Переход с SQLite на PostgreSQL потребовал ряда изменений в ORM-модели. Несмотря на то, что SQLAlchemy предоставляет дополнительный уровень абстракции над БД, на практике разные СУБД реализуют типы данных и своё поведение немного по-разному. В связи с этим необходимо было адаптировать модель под особенности PostgreSQL, чтобы обеспечить стабильную и предсказуемую работу приложения.

Во-первых, были уточнены типы данных: `db.Integer` заменён на `BigInteger`; строковые поля, реализованные через `db.String(255)` или `db.UnicodeText`, были заменены на `Text`,

так как в PostgreSQL нет необходимости ограничивать длину текстовых полей без явной причины; тип `db.DateTime(timezone=True)` был заменён на `TIMESTAMP(timezone=True)`, соответствующий нативному типу PostgreSQL с поддержкой временных зон.

Во-вторых, для некоторых колонок были явно заданы параметры `autoincrement` и `unique`, так как в PostgreSQL это является обязательным, в отличие от SQLite.

5.1.3 Перенос данных

После настройки новой архитектуры и адаптации ORM-модели возникла задача переноса существующих данных из SQLite в PostgreSQL. Основной целью было максимально сохранить актуальные данные, при этом избавившись от устаревшей или некорректной информации.

На первом этапе был развёрнут контейнер PostgreSQL, после чего приложение было сконфигурировано таким образом, чтобы одновременно подключаться и к старой базе на SQLite, и к новой на PostgreSQL. Перед началом переноса, из SQLite вручную были удалены данные, не подходящие под новую схему: сети привязанные к несуществующим пользователям, а также эмуляции отсутствующих сетей.

Для самой миграции использовалась специализированная утилита `pgloader [19]`, предназначенная для переноса данных между разными СУБД. Прямо внутри контейнера с Postgres, был создан конфигурационный файл `.load` с настройками утилиты, а также описанием правил переноса типов между SQLite и PostgreSQL, чтобы избежать возможных конфликтов и неточностей.

После запуска `pgloader`, данные были успешно перенесены и никаких конфликтов не возникло, благодаря заблаговременному тестированию и подготовке к переносу.

5.2 Рефакторинг бэкенда

5.2.1 Улучшения в эмуляторе

Эмулятор Miminet является его ключевой частью, которая стабильно обновляется студентами каждый месяц на протяжении всего учебного года. В связи с тем, что в нём накопилось достаточное количество громоздкого и дублирующегося кода, а также устаревших наименований и комментариев, было принято решение провести рефакторинг с целью улучшить ситуацию.

Прежде всего, были устранены неточности в именовании классов, файлов и переменных. Например, часто вместо «эмуляции» использовалось слово «симуляция», что вводило в заблуждение. Также были переписаны некоторые комментарии, которые ранее не всегда соответствовали реальной логике кода и могли искажать смысл происходящего.

Основной файл с функцией для эмуляции сетей, ранее представлявший собой блок из почти пятисот строк кода, был декомпозирован: ключевые части логики были вынесены в отдельные функции с осмысленными именами и подробным описанием, а логика, относящаяся к конфигурации сетей, была перенесена в новые классы. В результате файл, ранее содержащий достаточно сложный для понимания (и рефакторинга) код, сократился до менее чем 140 строчек, сохранив при этом всю необходимую функциональность.

Как уже говорилось выше, значительная часть логики построения и конфигурации сети была инкапсулирована в специальные классы, унаследованные от стандартных классов `Mininet`. Например, класс `MiminetTopology` теперь позволяет описывать и настраивать топологии в явной форме, управлять интерфейсами, настраивать сетевые устройства и линки, не загромождая основной код эмулятора.

В результате, эмулятор приложения был переработан, исправлены ошибки с терминологией, улучшилась читаемость кода.

5.2.2 Переработка тестирования бэкенда

Тестовая система бэкенда Miminet была переработана с целью повышения читаемости, удобства сопровождения и масштабируемости.

Главным улучшением стало упрощение архитектуры самих тестов: вместо множества разрозненных проверок на каждый тип сетевого пакета, теперь используются обобщённые вспомогательные функции и параметризация тестов. После рефакторинга удалось выразить всё множество тест-кейсов через одну параметризованную функцию. Благодаря избавлению от дублирующегося кода, общий объём файла для запуска тестов уменьшился почти на 50%, а добавление тестов на новые протоколы теперь не требует понимания всей структуры файла и деталей тестирования других протоколов.

Также были внесены и более мелкие изменения:

- Тесты были полностью вынесены из основной кодовой базы (`src/`) в отдельную директорию `tests/`. Использование таких устоявшихся практик разработки, как отделение тестовой инфраструктуры от бизнес-логики, поможет новым разработчикам быстрее разбираться в структуре приложения.

- Было реализовано логирование в процессе выполнения тестов и возможность настройки через конфигурационный файл. Это изменение упростило отладку и позволило лично мне быстрее находить ошибки во время рефакторинга бэкенда.

- Весь код был снабжён поясняющими комментариями, добавлено краткое описание работы тестов и инструкция для их запуска в README.

Помощь в подготовке к YADRO ИМПУЛЬС 2025

«YADRO ИМПУЛЬС» —- это программа летних стажировок для студентов и молодых специалистов, реализуемая компанией «YADRO». Она предоставляет участникам возможность погрузиться в работу над реальными инженерными задачами в таких направлениях, как программная и аппаратная разработка, системное проектирование, радиотехника и другие. Программа включает образовательные мероприятия, наставничество со стороны опытных инженеров компании, а также возможность дальнейшего трудоустройства.

Отбор на стажировку осуществляется в несколько этапов, включая как теоретическое тестирование, так и практические задания, направленные на проверку реальных знаний студентов в конкретной области. В предыдущие годы для проведения практического этапа уже использовался проект Miminet, однако до 2025 года проверка выполнялась вручную. В рамках отбора на стажировку «YADRO ИМПУЛЬС 2025» было принято решение внедрить систему автоматической проверки заданий по сетям. Эта система должна была обеспечить возможность массового проведения практического этапа отбора, позволяя его участникам в удобной форме выполнять и сдавать задания, а команде Miminet экономить часы на проверке этих заданий.

6.1 Описание проблемы

За 1–1.5 недели до начала отборочного этапа, поступили условия задач и требования от компании «YADRO». Одним из заданий предусматривалось моделирование отказоустойчивости сетевой архитектуры: «при выходе из строя одной из сетей N23 или N32 связь не нарушается». Это требование оказалось несовместимым с текущей архитектурой системы автоматической проверки заданий.

Проблема возникла в том, что цикл обработки каждого задания был следующим: приём полностью сконфигурированной сети, передача её на эмуляцию, получение результатов эмуляции и последующая проверка корректности. Такая схема позволяла оценивать задания, которые не требовали моделирования каких-либо ситуаций поверх уже сконфигурированной сети, но не предусматривала возможности (в частности) избирательного отключения соединений прямо в процессе тестирования. Это сделало невозможным реализацию условия об устойчивости к отказу. Учитывая ограниченные сроки и необходимость оперативного внесения архитектурных изменений, было принято решение привлечь дополнительную помощь, в том числе с моей стороны.

Кроме этого, была проблема с отсутствием этапа предобработки сети пользователя, при котором тестирующая система могла самостоятельно менять уже настроенную сеть под текущее задание. Например, без данного этапа, пользователь должен был самостоятельно выполнять команду «ping» у нужного устройства, чтобы проверка сети прошла успешно. В связи с новыми требованиями, этот этап получил ещё большую важность, ведь необходимо было выразить требование на проверку отказоустойчивости сети.

6.2 Требования и ограничения при проектировании

Проектирование новой архитектуры началось с анализа требований и ограничений, наложенных как временными рамками, так и существующим устройством системы.

1. Нежелательно было вносить изменения в эмулятор Miminet. Эта часть изначально разрабатывалась как независимый модуль и попытка внедрить дополнительную логику нарушила бы абстракцию, а также могла бы повлечь за собой нежелательные побочные эффекты — в том числе повлиять на другие функции приложения.

2. Этап *предобработки* пользовательских сетей обязательно должен был быть добавлен. В этот этап входила бы очистка лишних команд, например, ICMP-запросов, а также дополнительная модификация сети, например, отключение одного из линков между узлами, чтобы моделировать отказ.

3. Проверка одного задания никак не могла быть выполнена за одну эмуляцию. Все логические проверки выполнялись на стороне фронтенда, так как сам эмулятор является независимой частью и не должен был содержать этой логики. Таким образом, каждая новая топология (например, с отключенным линком) требовала отдельного запуска эмуляции. При этом между отдельными эмуляциями требовалось сохранять контекст: например, баллы за уже пройденные этапы задания или информация о пользователе.

4. Пользователь не должен был ожидать результата проверки в реальном времени. Проверка должна была запускаться асинхронно и не блокировать возможность отправки новых заданий или эмуляции обычных сетей.

5. Ключевым требованием было соблюдение принципа простоты и читаемости архитектуры. Так как система должна была использоваться другими разработчиками после окончания нашей работы, избыточная сложность или неочевидные решения могли бы затруднить её поддержку и дальнейшее расширение.

6.3 Новая архитектура системы проверки практических заданий

После сбора требований и проведения серии обсуждений с командой, была разработана и реализована обновлённая архитектура системы автоматической проверки:

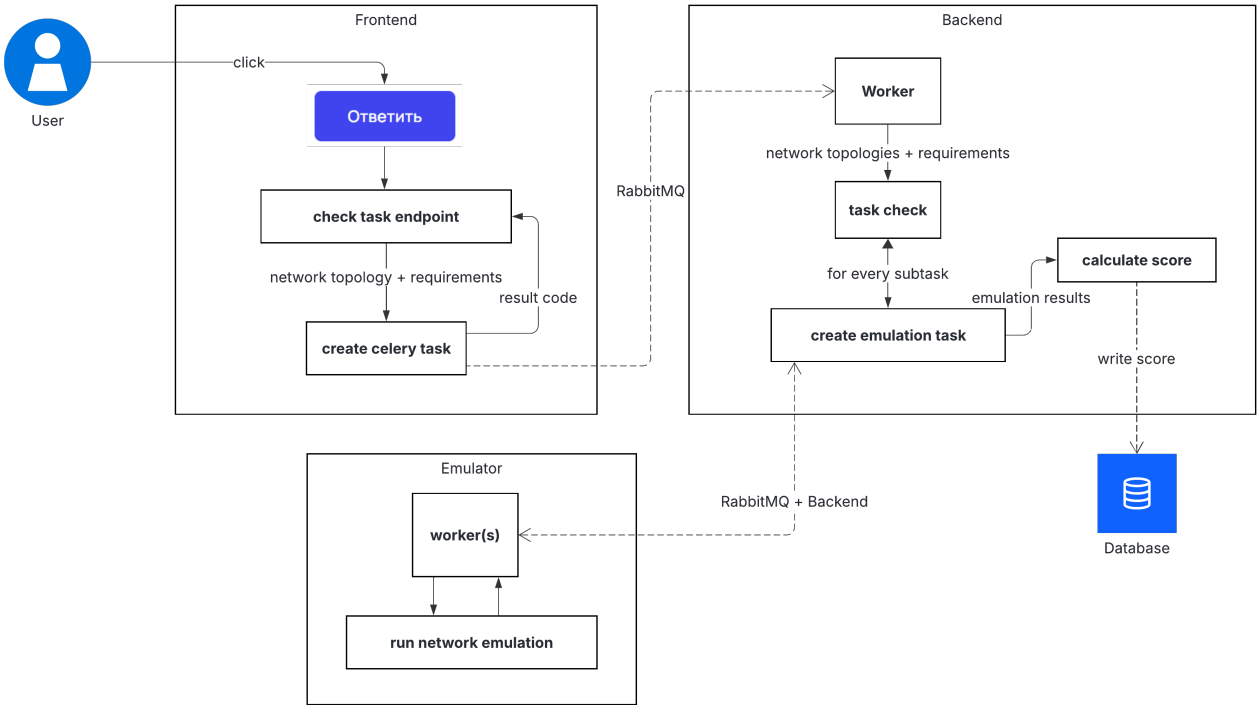


Рис. 1: Схема проверки задания пользователя в обновлённой архитектуре.

Новая архитектура построена на взаимодействии нескольких автономных, но скоординированных между собой компонентов, обеспечивающих асинхронную обработку пользовательских запросов. В обобщённом виде систему можно разделить на четыре ключевых модуля:

1. Пользовательский интерфейс, в котором пользователь строит и настраивает сеть, а затем отправляет её на проверку. Здесь же происходит предобработка сети.

2. Серверную часть, где происходит управление задачами эмуляции и используется модуль для проверки результатов.

3. Эмулятор сетей, который дополнительно никак изменён не был.

4. База данных, где хранится информация о выполненном задании и итоговые баллы за него.

Связь между компонентами осуществляется через очередь сообщений *RabbitMQ*, а для прямого обмена данными между эмулятором и бэкендом используется механизм (*RPC*).

Процесс проверки задания запускается пользователем через веб-интерфейс после нажатия на кнопку «Ответить». После этого фронтенд отправляет запрос на специализированный *endpoint* проверки, передавая текущую конфигурацию сети и условия задания. На следующем этапе выполняется предобработка: удаляются избыточные команды (например, лишние ICMP-запросы), добавляются необходимые команды в зависимости от условий конкретного задания. Если задание, например, предполагает проверку устойчивости к отказу, исходная топология автоматически клонируется с удалением одного из линков, в результате чего формируется несколько независимых конфигураций сетей, подлежащих проверке.

Полученные модифицированные конфигурации сетей отправляются в бэкенд через очередь сообщений. Обработчик выполняет эмуляцию каждой из сетей с помощью RPC-интерфейса к эмулятору и сохраняет полученные результаты в список. Завершив эмуляцию всех сетей, относящихся к заданию, система переходит к финальной фазе — модуль проверки анализирует поведение сети в каждом сценарии, выставляет итоговые баллы и записывает результаты в базу данных.

Таким образом, удалось покрыть все имеющиеся требования, не слишком переусложняя уже имеющуюся архитектуру.

Результаты

За год в рамках производственной практики были решены следующие задачи:

1. Создание метода автоматического end-to-end тестирования Miminet
    - выполнен обзор технологий,
    - собраны требования для тестирующей системы,
    - созданы тесты на основные функции Miminet,
    - реализована возможность локального запуска тестов,
    - выполнена интеграция с CI.
2. Масштабирование и улучшение кода Miminet
    - проект перенесён на новую СУБД PostgreSQL,
    - выполнен рефакторинг бэкенда.
3. Помощь в подготовке к «YADRO ИМПУЛЬС 2025».

Литература

1. *Зеленчук Илья.* Веб-эмулятор компьютерных сетей Miminet. — URL: https://miminet.ru/.
2. *Github.* Репозиторий проекта Miminet. — URL: https://github.com/mimi-net/miminet.
3. *Зеленчук Илья.* Курс «Основы компьютерных сетей». — URL: https://stepik.org/course/208904/info.
4. *D. Farinacci T. Li S. Hanks D. Meyer P. Traina.* Generic Routing Encapsulation specification. — 2000. — URL: https://datatracker.ietf.org/doc/html/rfc2784.
5. *IEEE.* Standard for Local and Metropolitan Area Networks–Bridges and Bridged Networks. — 2018. — URL: https://standards.ieee.org/ieee/802.1Q/6844/.
6. *Information Sciences Institute.* Transmission Control Protocol Specification. — 1981. — URL: https://datatracker.ietf.org/doc/html/rfc793.
7. *Postel J.* User Datagram Protocol specification. — 1980. — URL: https://www.rfc-editor.org/rfc/rfc768.
8. *VMware.* RabbitMQ main page. — URL: https://www.rabbitmq.com/.
9. *Selenium.* Selenium Documentation. — URL: https://www.selenium.dev/documentation/.
10. *Selenium.* Selenium github page. — URL: https://github.com/SeleniumHQ/selenium.
11. *Cypress.* Cypress Documentation. — URL: https://docs.cypress.io/app/get-started/why-cypress.
12. *Inc. Docker.* Docker main page. — URL: https://www.docker.com/.
13. *Inc. Docker.* Docker Compose Documentation. — URL: https://docs.docker.com/compose/.
14. *Selenium.* Selenium Grid Documentation. — URL: https://www.selenium.dev/documentation/grid/.
15. *Selenium.* Hub and Node in Selenium Grid. — URL: https://www.selenium.dev/documentation/grid/getting_started/#hub-and-node.
16. *Hipp D. Richard.* SQLite home page. — URL: https://sqlite.org/.
17. *Group PostgreSQL Global Development.* PostgreSQL main page. — URL: https://www.postgresql.org/.
18. *Hub Docker.* PostgreSQL official docker image. — URL: https://hub.docker.com/_/postgres.
19. *Fontaine Dimitri.* pgloader main page. — URL: https://pgloader.io/.