



Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет
имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ Информатики и систем управления

КАФЕДРА Теоретической информатики и компьютерных технологий

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА К КУРСОВОЙ РАБОТЕ

НА ТЕМУ:

Консольное приложение под операционную
систему Linux/Unix для визуализации
поточковых данных

Студент ИУ9-51Б
(Группа)

(Подпись, дата) В.А. Локшин
(И.О.Фамилия)

Руководитель курсовой работы

(Подпись, дата) Д.П. Посевин
(И.О.Фамилия)

Консультант

(Подпись, дата) _____
(И.О.Фамилия)

2023 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ	5
1 Обзор предметной области	6
1.1 Анализ конкурентов	6
1.2 Технические обзор графических окон	7
1.3 Технический обзор терминала.....	9
1.4 Производительность визуализатора	10
1.5 Устройства без графических окон	11
2 Разработка приложения.....	13
2.1 Клиент-серверная архитектура приложения.....	13
2.2 Выбор языков программирования	14
2.3 Визуализатор потоковых данных.....	15
2.4 SSH-сервер	16
2.5 Клиент для ssh-сервера и генератор тестовых данных	17
3 Реализация приложения	19
3.1 Постановка задачи реализации приложения.....	19
3.2 Особенности реализации SSH-сервера.....	19
3.3 Особенности реализации визуализатора потоковых данных.....	20
3.4 Клиент и генерация данных для тестирования приложения	23
3.5 Описание работы приложения	24
4 Тестирование.....	28
ЗАКЛЮЧЕНИЕ.....	31
СПИСОК ЛИТЕРАТУРЫ	32

ПРИЛОЖЕНИЕ.....	34
-----------------	----

ВВЕДЕНИЕ

В нашу эпоху объем данных растет экспоненциально и важно иметь эффективные инструменты для их визуализации и анализа. Это становится особенно важным в случаях, когда доступ к графическим интерфейсам ограничен, например, при работе в консольном интерпретаторе на удаленных серверных средах или системах без графических окон.

Одним из ключевых элементов требуемого решения является возможность удаленного выполнения, что делает его удобным инструментом для мониторинга данных на удаленных серверах, в распределенных системах и микропроцессорных устройствах, где доступ к графическим интерфейсам ограничен.

Таким образом, разрабатываемое приложение открывает новые перспективы для визуализации данных в условиях, где не всегда доступны традиционные средства визуализации с использованием графических окон.

Целью данной курсовой работы является разработка программы под операционные системы UNIX, визуализирующей потоковые данные в терминале без возможностей запуска X Windows System [1].

1 Обзор предметной области

1.1 Анализ конкурентов

Существующие инструменты в области визуализации данных, такие как Matplotlib и GNU Plot, прекрасно служат своей цели в графических интерфейсах, однако они ограничены в предоставлении полноценной консольной визуализации числовых данных. Эти библиотеки в основном ориентированы на создание графиков в графических окнах и не обеспечивают удовлетворительного опыта в терминале.

Например, Matplotlib предоставляет богатые возможности визуализации, но его использование в терминале ограничивается из-за зависимости от графических библиотек, что делает его менее пригодным для консольной среды. GNU Plot, в свою очередь, хотя и является мощным инструментом для создания графиков, также ориентирован на визуализацию в графических интерфейсах.

Терминальные инструменты, такие как Termplot, стремятся предоставить консольные графики, но часто ограничиваются в функциональности. Они могут предоставлять базовые графические возможности, но часто не обеспечивают необходимой гибкости и возможности адаптации для сложных задач визуализации в терминале.

Таким образом, ни Matplotlib, ни GNU Plot, ни Termplot не до конца удовлетворяют требованиям для эффективной консольной визуализации потоковой информации в реальном времени. Они либо зависят от графических интерфейсов, либо предоставляют ограниченные возможности в терминале, что подчеркивает необходимость разработки специализированного инструмента, учитывающего требования удаленных серверных сред и систем без графических окон.

1.2 Технические обзор графических окон

Графические окна — компонент современных операционных систем, предоставляющий пользователю удобный интерфейс для взаимодействия с приложениями. Они представляют собой области на экране, в которых отображаются графические элементы.

Далее рассмотрим структуру графического окна [2]. Оно состоит из следующих элементов:

1. Графический контекст (Graphics Context) — определяет параметры отображения, такие как цвет, шрифт, стиль линий и другие атрибуты. Он предоставляет приложению средства для рисования.
2. Буфер экрана (Screen Buffer) — область памяти, в которой хранится изображение, отображаемое в окне. Для обеспечения плавного обновления, часто используется двойная буферизация.
3. Менеджер окон (Window Manager) — отвечает за управление созданием, отображением и перемещением окон на экране. Он также обрабатывает взаимодействие с окнами, такие как изменение размеров и закрытие.

Рассмотрим, как взаимодействие с окнами происходит в операционной системе, внедряясь в процессы создания, управления и отображения. Когда приложение запускается в операционной системе, оно инициирует запрос на создание графического окна. Этот запрос вызывает активацию операционной системы, которая выделяет необходимые ресурсы для создания окна и связывает его с соответствующим процессом.

Процесс инициализации не ограничивается простым выделением ресурсов. После создания окна приложение инициализирует графический контекст, устанавливая параметры рисования, такие как цвета, шрифты и стили. Это позволяет приложению контролировать внешний вид графических элементов, создаваемых в окне. Используя установленные параметры,

приложение приступает к рисованию графических элементов и помещает изображение в буфер экрана.

Важно отметить, что буфер экрана представляет собой виртуальную область памяти, где хранится изображение, которое должно быть отображено в окне. Для обеспечения плавности отображения и избежания мерцания, часто используется метод двойной буферизации. Этот подход предполагает наличие двух буферов: один, называемый передним буфером, отображается на экране, в то время как другой, задний буфер, активно подготавливается приложением. Периодически оконный менеджер обновляет содержимое экрана, перемещая изображение из заднего буфера в передний, что обеспечивает плавное и непрерывное отображение изменений.

Оконный менеджер [3] следит за событиями, такими как перемещение мыши или нажатия клавиш, и передает их соответствующим окнам для обработки.

В UNIX-подобных системах X Window System (X11) выступает в роли основного инструмента для работы с графикой и окнами. Этот протокол обеспечивает эффективное взаимодействие с графическими элементами и управление окнами. Совместно с библиотекой Xlib, предоставляющей API, X11 становится мощным средством для создания, управления и взаимодействия с окнами в UNIX-системах.

Xlib предоставляет высокоуровневый доступ к функциональности X11, позволяя разработчикам эффективно управлять графическими ресурсами. С его помощью можно легко настроить графический контекст, работать с цветами, шрифтами и другими атрибутами, что открывает дверь для создания разнообразных и сложных пользовательских интерфейсов.

Этот протокол также служит основным интерфейсом для многих графических приложений и библиотек в UNIX-среде. Например, популярные библиотеки GTK и Qt используют X11 для рендеринга графики и управления окнами. Именно через X11 эти библиотеки обеспечивают интуитивное взаимодействие пользователя с приложениями на UNIX-платформах.

Использование X11 в разработке графических интерфейсов подчеркивает его значимость и признание в сообществе разработчиков. Его гибкость и расширяемость являются ключевыми факторами, содействующими созданию современных и эффективных пользовательских интерфейсов в UNIX-системах.

1.3 Технический обзор терминала

Терминал [4] — это текстовый интерфейс, предоставляющий пользователю средство взаимодействия с операционной системой. В отличие от графических окон, терминал работает в текстовом режиме, где вывод и ввод осуществляются через символы и команды.

Структура терминала включает в себя следующие элементы:

1. **Виртуальный экран (Virtual Screen):** терминал создает виртуальный экран, представляющий собой область, на которой отображается текст. Этот текст может включать в себя вывод команд, результаты выполнения программ, и другие сообщения.
2. **Текстовый буфер (Text Buffer):** буфер текста служит для хранения содержимого виртуального экрана. Он может быть использован для прокрутки и перехода к предыдущим выводам.

Процесс работы с терминалом состоит из шагов:

1. **Отображение текста:** при выполнении команды система выводит текст на виртуальный экран терминала. Это может быть как результат выполнения команд, так и обычный текстовый вывод.
2. **Курсор и позиционирование:** терминал поддерживает курсор, который указывает на текущую позицию в тексте. При вводе текста или выполнении команд курсор перемещается по виртуальному экрану.

3. Цветовая схема (Color Scheme): некоторые терминалы поддерживают цветовую схему для выделения синтаксиса или подчеркивания важных элементов текста.

Терминал в UNIX-подобных системах взаимодействует с программами через стандартные потоки ввода (stdin) и вывода (stdout) [5]. Этот механизм предоставляет прямой канал связи между пользователем и выполнением команд, где результаты отображаются непосредственно в терминале. Это взаимодействие основано на принципе текстового режима, что делает терминал эффективным средством для работы с операционной системой.

Одним из ключевых элементов функциональности терминала является обработка сигналов, направляемых процессам. Это дает пользователям возможность активно взаимодействовать с выполняемыми задачами, контролируя их выполнение. Например, пользователь может отправить сигнал процессу для завершения его работы или изменения приоритета выполнения.

Таким образом, терминал предоставляет эффективное средство взаимодействия с операционной системой в текстовом режиме, предоставляя пользователю контроль и гибкость в использовании команд и выполнении задач.

1.4 Производительность визуализатора

Текстовые терминалы могут поддерживать ASCII-графику, что представляет собой простые изображения, созданные с использованием символов. Это требует минимальной обработки и обычно не создает значительной нагрузки на процессор, но терминалы ограничены в графических возможностях, и их способности ограничиваются простыми графическими элементами, которые можно представить с использованием текстовых символов.

Графические окна могут использовать библиотеки для отрисовки полноценных графиков с использованием векторной или растровой графики. Это может включать линии, кривые, заполненные области, и другие сложные

элементы. При использовании библиотек, таких как Matplotlib в Python, процесс отрисовки графиков может включать в себя выполнение сложных алгоритмов и создание высококачественных изображений.

При сравнении текстовых терминалов и графических окон можно отметить различия. Терминалы обладают кратно более низкой нагрузкой на процессор по сравнению с графическими окнами. Это обобщенное суждение, и конечные результаты могут варьироваться в зависимости от конкретных сценариев использования и программного обеспечения.

1.5 Устройства без графических окон

Устройства, подобные терминалу и не имеющие графических окон, включают в себя множество встраиваемых систем и одноплатных компьютеров. Примером таких устройств является Raspberry Pi и подобные одноплатные компьютеры.

Raspberry Pi - это одноплатный компьютер [6] размером с кредитную карту, который работает на различных версиях ARM-процессоров. В отличие от обычных персональных компьютеров, Raspberry Pi не имеет графического интерфейса по умолчанию, и взаимодействие с ним часто происходит через терминал.

Raspberry Pi подобным терминалу делают такие особенности, как:

1. Raspberry Pi, при использовании операционной системы Raspbian (основанной на Debian), может поставляться с текстовым интерфейсом по умолчанию.
2. Raspberry Pi часто используется в удаленных или встраиваемых сценариях, где доступ к графическому интерфейсу может быть неэффективным или невозможным. В таких случаях взаимодействие осуществляется через SSH или другие терминальные протоколы.

Raspberry Pi широко используется в образовательных целях, где учащиеся могут учиться основам программирования и системного администрирования через терминал. Также Raspberry Pi может служить в качестве легких серверов, таких как веб-серверы, базы данных или серверы для домашней автоматизации. Одноплатные компьютеры, такие как Raspberry Pi, активно используются в проектах IoT для управления и мониторинга устройств и сенсоров.

2 Разработка приложения

2.1 Клиент-серверная архитектура приложения

В основе приложения лежит клиент-серверная архитектура, где эффективное взаимодействие между компонентами осуществляется через сетевое соединение. Клиентский компонент генерирует и передает данные на сервер, используя протокол SSH [7] для обеспечения безопасного удаленного доступа. SSH-сервер, в свою очередь, обеспечивает возможность удаленного выполнения программы, обрабатывающей поступившие данные и визуализирующей их в удобном формате.

Компилируемый бинарный файл, размещенный на сервере, принимает данные от клиента, проводит их обработку и затем создает графики в соответствии с предоставленными входными данными. Эта архитектура позволяет эффективно использовать ресурсы сервера для выполнения вычислительных задач, в том числе и для генерации графиков.

Генератор данных, как дополнительный компонент системы, отвечает за предоставление тестовых данных, которые передаются клиенту. Этот процесс воспроизведения тестовых данных дает клиенту возможность тестирования и проверки функциональности приложения, а также поддерживает его работоспособность.

Мною используется SSH-протокол для обеспечения безопасного удаленного взаимодействия. Безопасность обеспечивается шифрованием данных и аутентификацией через SSH.

Исходный код программы компилируется в бинарный файл, который далее размещается на SSH-сервере. Этот процесс развертывания программы и связанных файлов на сервере обеспечивает доступ клиента к функционалу приложения. Важно отметить, что использование SSH в данном контексте не только обеспечивает безопасность, но также позволяет эффективно управлять и

выполнять приложение на удаленном сервере. Генератор данных создает тестовый поток данных для клиента и программы. Схема приложения представлена на Рисунке 1.

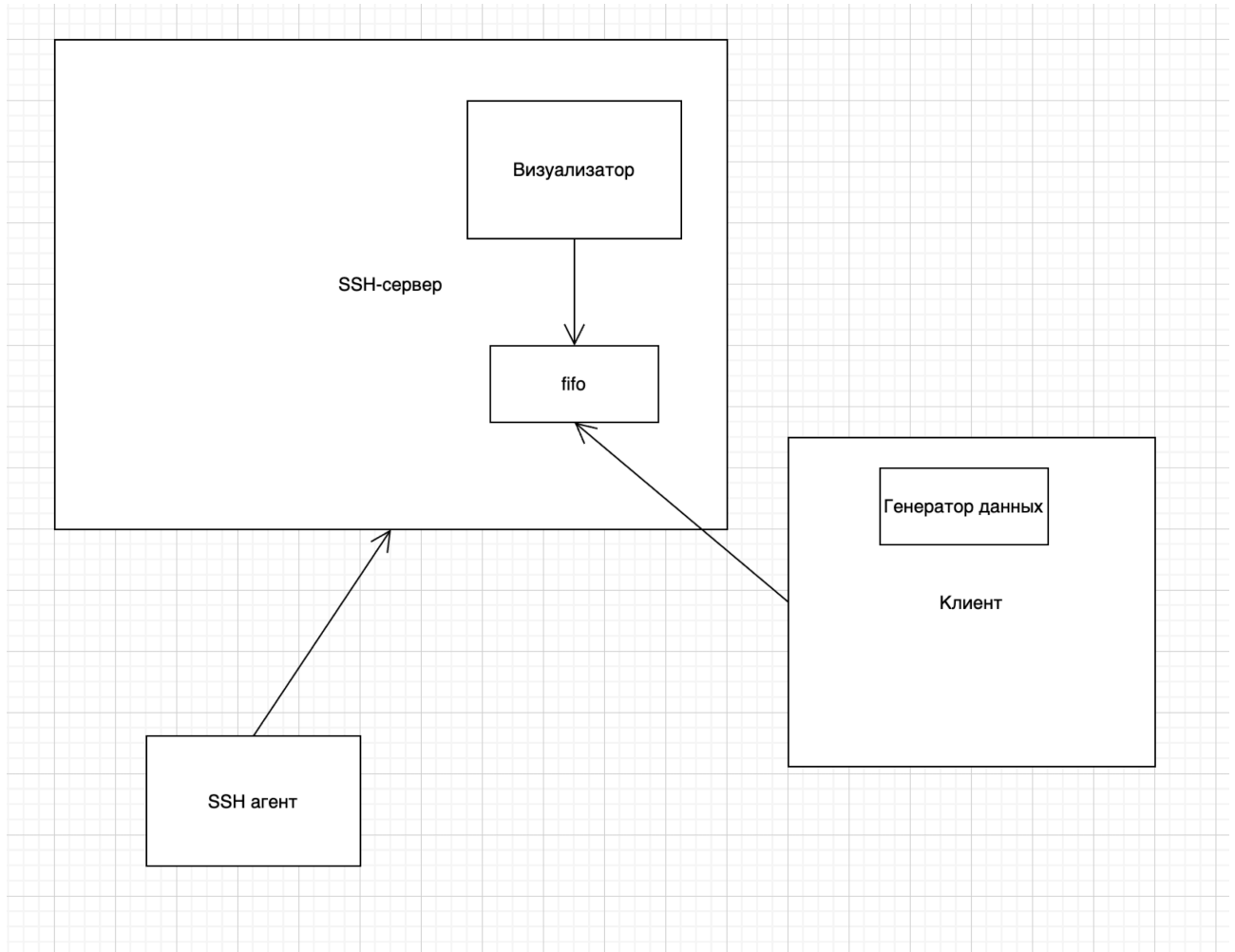


Рисунок 1 — Схема приложения

2.2 Выбор языков программирования

Для разработки компонентов приложения были выбраны два языка программирования, исходя из их специфических характеристик и требований к проекту.

Для визуализатора был выбран язык программирования C++ для разработки бинарной программы визуализации данных. Этот выбор обусловлен

целью использования программы в контексте Интернета вещей (IoT) [8], где C++ широко распространен и находит применение в реальных проектах.

Язык программирования Go был выбран для разработки серверной, клиентской и генераторной частей приложения. Go обеспечивает простоту и эффективность в написании кода, что особенно важно для серверных приложений. Благодаря своей производительности, Go является идеальным выбором для реализации взаимодействия между компонентами приложения.

Преимущество выбора заключается в том, что оба языка, C++ и Go, компилируемые, что обеспечивает высокую производительность и эффективность в различных сценариях использования.

C++ поддерживает широкий спектр применений, что делает его подходящим для реального мира и потенциальных проектов в области Интернета вещей.

Go является идеальным выбором для разработки серверов, обеспечивая простоту в написании и высокую производительность при взаимодействии с компонентами на различных платформах.

2.3 Визуализатор потоковых данных

Библиотека Ncurses [9] представляет собой библиотеку для создания текстовых пользовательских интерфейсов в терминале. Ее обширный набор функций по управлению вводом и выводом в терминале делает ее превосходным выбором для разработки текстовых графических интерфейсов в приложениях, работающих в терминальной среде.

Поддержка цветов в библиотеке Ncurses открывает дополнительные возможности для создания наглядных и красочных интерфейсов в терминале. Это особенно важно для приложений, где визуальная яркость и разнообразие цветов способствуют лучшему восприятию данных и графических элементов.

Ncurses также предоставляет широкие возможности по манипуляции положением курсора и обработке нажатий клавиш. Это обеспечивает плотное взаимодействие с пользователем и создает возможность реализации динамических и отзывчивых интерфейсов в терминале.

Важным компонентом библиотеки является возможность вывода и форматирования текста в терминале. Это особенно ценно при отображении графиков и данных, где правильное форматирование текста способствует более четкому и информативному представлению информации.

Таким образом, библиотека ncurses является неотъемлемым инструментом для разработки высокофункциональных и удобных текстовых пользовательских интерфейсов в терминале.

Визуализатор включает в себя следующие модули:

1. Модуль приема данных — принимает данные от клиента или других источников через входной поток.
2. Модуль визуализации водопада — использует функционал Ncurses для создания динамического эффекта водопада, отображающего изменения данных по мере их поступления.
3. Модуль отрисовки двумерного графика — использует Ncurses для создания окна и вывода двумерных графиков в терминале.

Эффективное использование библиотеки Ncurses в визуализаторе обеспечивает создание удобного и информативного текстового пользовательского интерфейса, подходящего для взаимодействия в терминальной среде.

2.4 SSH-сервер

Протокол SSH обеспечивает безопасное удаленное взаимодействие с приложением и позволяет пользователям подключаться к серверу, загружать и запускать бинарный файл в терминале.

Для создания терминала на сервере был выбран терминальный мультиплексор Tmux, который позволяет организовать несколько виртуальных окон в одном терминале. Пользователи могут подключаться к серверу посредством SSH-протокола, соответственно, могут использовать различные сторонние SSH-клиенты, также пользователи могут загружать бинарный файл на сервер и запускать его в терминале для обработки данных.

Tmux обеспечивает виртуальные окна и панели, что позволяет пользователям эффективно организовывать свою работу в терминале. Сеансы Tmux сохраняют состояние работы, что полезно при длительных вычислениях или обработке данных.

Он использовался как псевдотерминал (PTY) [10] — это пара виртуальных устройств, обычно представленных как мастер и рабочий. PTY предоставляет интерфейс для программного взаимодействия с терминалом. Мастер-терминал служит в качестве точки ввода и вывода для приложения, в то время как рабочий-терминал ассоциирован с физическим терминалом или окном терминала.

Процессы могут взаимодействовать через PTY так, как если бы они взаимодействовали с физическим терминалом, что позволяет программам, ожидающим ввода с терминала или вывода на терминал, работать в условиях, когда реальный терминал не доступен или используется удаленное соединение.

2.5 Клиент для ssh-сервера и генератор тестовых данных

Клиентское приложение обеспечивает взаимодействие пользователя с сервером и контроль генератора данных. Клиент использует SSH-протокол для безопасного подключения к удаленному серверу.

Пользователь передает параметры генерации данных (задержка t , длина массива, амплитуда чисел) на сервер. Клиент отправляет запрос на сервер с параметрами генерации. Получив подтверждение от сервера о валидности

конфигурации, клиент запускает генератор и получает сгенерированные числа в момент времени t .

Генератор данных ответственен за генерацию случайных чисел с учетом переданных параметров. Генератор генерирует случайные числа с учетом заданных параметров. Генератор получает от сервера конфигурацию для генерации данных и отправляет сгенерированные числа клиенту по мере генерации.

3 Реализация приложения

3.1 Постановка задачи реализации приложения

Для выполнения поставленной задачи было необходимо реализовать три модуля приложения: SSH-сервер, визуализатор и клиент для генерации и поставки данных.

В SSh-сервере должна быть поддержка подключения различных SSH-агентов, сам сервер должен поддерживать внутри себя запуск терминала без ограничений, так же нужно иметь возможность загружать на него бинарный файл и запускать его.

Визуализатор отвечает за отрисовку в консоли графиков и обработку поступающей в него информации, он должен асинхронно читать потоковую информацию и сразу отображать ее на экране.

Клиент должен поставлять для визуализатора потоковую информацию согласно схеме, данные для клиента производятся генератором.

Разработка проекта осуществлялась в средах разработки GoLand и CLion. Эти среды разработки предоставляют умное автодополнение, анализ кода и надежный рефакторинг — возможности, необходимые для быстрой и эффективной разработки.

3.2 Особенности реализации SSH-сервера

Разработка проекта началась с создания SSH-сервера, и выбор языка программирования Go в данном контексте был обоснован стремлением к эффективности в сетевых приложениях. Язык Go известен своей высокой производительностью и удобством в разработке подобных приложений. Этот

выбор был подкреплён обширной библиотекой, предоставляемой языком Go для работы с протоколом SSH.

Основным инструментом реализации SSH-сервера в проекте стала стандартная библиотека `x/crypto/ssh`, которая предоставляет все необходимые инструменты для эффективного создания и управления SSH-соединениями. Эта библиотека включает в себя множество функций, позволяющих обеспечить безопасное и стабильное взаимодействие между клиентом и сервером по протоколу SSH. (Листинг 1)

Для настройки SSH-сервера использовалась команда `ssh-keygen` для генерации приватных и публичных ключей. Конфигурация сервера определялась с использованием транспортного протокола, порта и типа авторизации. Разработка включала использование встроенной функции для прослушивания запросов к серверу и написание собственной функции обработки этих запросов. Особый акцент был сделан на обеспечение поддержки параллельных подключений множества клиентов. Это обеспечивается через встроенные механизмы, что делает сервер таким, что он может обрабатывать нескольких клиентов одновременно. В процессе разработки ключевым аспектом стала поддержка терминала на сервере. Для этого использовались библиотеки `x/term` (внутренняя) и `creack/pty` (внешняя). (Листинг 2)

`x/term` предоставляет инструменты для работы с терминалом в контексте программы, включая управление вводом/выводом, цветами и стилями.

`creack/pty` позволяет создавать псевдотерминалы (PTY) для взаимодействия с программами, работающими в терминальной среде, обеспечивая поток данных между процессами. (Листинг 3)

3.3 Особенности реализации визуализатора потоковых данных

Модуль визуализатора структурирован на три ключевые функции:

1. Функция, определяющая необходимый тип графика в зависимости от установленных флагов при запуске.
2. Функция, отвечающая за визуализацию графика "водопад".
3. Функция, отвечающая за визуализацию мерцающего графика.

Для взаимодействия с библиотекой `ncurses`, необходимой для работы с терминалом, требуется ее импорт в проект. Сборка на языке C++ осуществлялась с использованием технологии `CMake`, которая эффективно связывает библиотеки и устанавливает правильные зависимости для построения проекта. Этот подход обеспечивает корректную компиляцию и сборку, учитывая все необходимые библиотеки и зависимости.

При построении графиков было две ключевые логические части: обработка данных и визуализация. Основная проблема заключалась в необходимости создания механизма, который бы позволил программе одновременно принимать и отрисовывать данные. В данном контексте выбор между использованием многопоточности и работы с FIFO-файлами [11] был решен в пользу последних, как более быстрого и простого варианта.

FIFO (First In, First Out) представляют собой специальные файлы, обеспечивающие механизм межпроцессного взаимодействия через файловую систему. Создаваемые с использованием команды `mkfifo`, они служат именованным каналам, по которым процессы могут обмениваться данными. FIFO работает по принципу очереди данных, обеспечивая синхронизацию и порядок передачи информации между процессами. Это решение обеспечивает эффективное и быстрое взаимодействие без необходимости использования многопоточности, что особенно важно при передаче данных в режиме реального времени.

Таким образом, была решена проблема ввода информации, поскольку программа теперь извлекает данные в процессе выполнения из FIFO-файлов. Обработка данных осуществляется следующим образом: сначала происходит сканирование схемы данных, включая размеры графика, максимальную амплитуду и временной интервал t , с которым данные могут поступать. Затем

начинается прием потока данных. Однако важной частью этого процесса является блокирование на чтение данных, так как FIFO-файлы по умолчанию не обладают встроенным механизмом блокировки.

Блокирование на чтение представляет собой механизм, который приостанавливает выполнение программы до тех пор, пока не станет возможным считать данные из определенного источника. В случае FIFO-файлов блокирование на чтение гарантирует ожидание данных и обеспечивает синхронизацию, предотвращая чтение несуществующих данных и обеспечивая правильный порядок информации в реальном времени.

Таким образом данные помещались в буфер для последующей отрисовки на экран.

Для отображения информации из буфера в ncurses необходимо выполнить несколько шагов. Первым шагом является создание окна с помощью функции `initscr`. Затем, для использования цветов, используется `start_color`, который инициализирует цветовую пару. Далее, `attron` и `attroff` используются для включения и выключения атрибутов окна, таких как цвет. Функция `mvprintw` используется для размещения текста в определенной позиции окна.

Важные функции для работы с цветами в ncurses. `init_pair`: используется для инициализации цветовой пары, задавая соответствие между номером цвета и фоновым/текстовым цветом. `init_color`: позволяет настраивать конкретные цвета в палитре.

Относительно настройки цветов в водопаде, терминал может поддерживать разное количество цветов. Цвета в терминале представлены в виде ANSI-цветов, который является стандартом для управления цветами в текстовых терминалах. Этот стандарт включает в себя 8 основных цветов (черный, красный, зеленый, желтый, синий, маджента, циан, белый) и их яркие варианты. Позже было добавлено расширение с 256 цветами. Команды `tput colors` или `echo $TERM_PROGRAM` позволяют узнать количество цветов в терминале.

Ручной отбор 18 оттенков синего и красного в палитре позволяет управлять цветами в водопаде в зависимости от значений чисел, что создает эффективную визуализацию данных в терминале. (Листинг 4)

3.4 Клиент и генерация данных для тестирования приложения

Клиентская часть приложения ответственна за передачу данных между генератором и визуализатором. Она обеспечивает поточную передачу числовых данных, полученных от генератора, в `fifo`-файл, используемый визуализатором для визуализации графиков. Для генерации данных используется встроенный пакет для генерации случайных чисел в языке Go, а также механизм каналов (`channels`), предоставляемый Go. (Листинг 5)

Каналы в Go представляют собой механизм для обеспечения взаимодействия и синхронизации между горутинами (`concurrent goroutines`).

Горутины — это легковесные потоки выполнения в языке Go. Главная идея заключается в том, чтобы использовать каналы для безопасной передачи данных между горутинами.

Горутины и каналы позволяют эффективно управлять параллельными задачами, таким образом, клиент может создать горутину для генерации данных и передачи их через канал в `fifo`-файл для визуализатора.

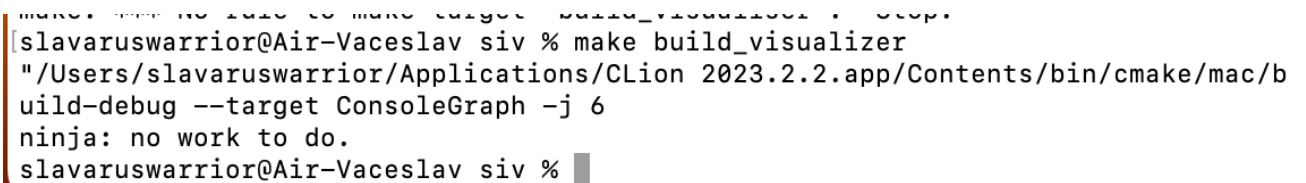
Промежуток времени `t` между генерацией чисел обеспечивается использованием функции ожидания времени в языке Go. Этот механизм позволяет управлять частотой генерации данных для создания рядов чисел по времени. (Листинг 6)

3.5 Описание работы приложения

Разработанное приложение работает следующим образом.

Первый шаг заключается в том, что пользователь собирает у себя на компьютере исходные файлы визуализатора командой “make build_visualiser”,

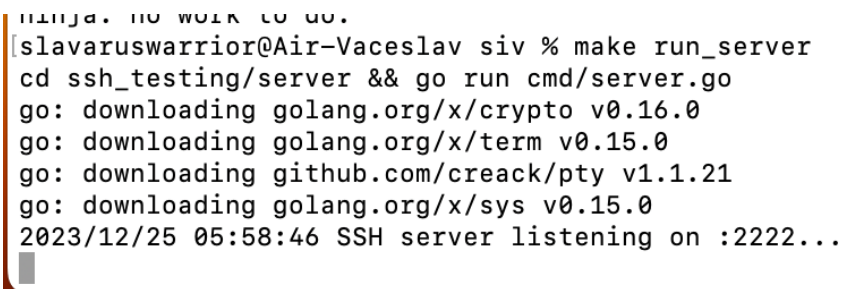
что показано на Рисунке 2.



```
make: *** No rule to make target 'build_visualizer'.  Stop.
slavaruswarrior@Air-Vaceslav siv % make build_visualizer
"/Users/slavaruswarrior/Applications/CLion 2023.2.2.app/Contents/bin/cmake/mac/b
uild-debug --target ConsoleGraph -j 6
ninja: no work to do.
slavaruswarrior@Air-Vaceslav siv %
```

Рисунок 2 — Сборка визуализатора данных

Вторым шагом пользователь запускает сервер командой “make run_server” (Рисунок 3).



```
ninja: no work to do.
slavaruswarrior@Air-Vaceslav siv % make run_server
cd ssh_testing/server && go run cmd/server.go
go: downloading golang.org/x/crypto v0.16.0
go: downloading golang.org/x/term v0.15.0
go: downloading github.com/creack/pty v1.1.21
go: downloading golang.org/x/sys v0.15.0
2023/12/25 05:58:46 SSH server listening on :2222...
```

Рисунок 3 — Запуск сервера

Третьим шагом пользователь подключается к серверу через любой агент, например, классический терминал по “ssh user@127.0.0.1 -p 2222” к порту 2222 (Рисунок 4).



```
slavaruswarrior — ssh user@127.0.0.1 -p 2222 — 80x24
Last login: Mon Dec 25 04:41:17 on ttys002
slavaruswarrior@Air-Vaceslav ~ % ssh user@127.0.0.1 -p 2222
user@127.0.0.1's password:
Welcome to vis SSH server!
>
```

Рисунок 4 — Вход на сервер

Четвертым шагом пользователь прописывает на сервере команду tmux и попадает в PTY.

Пятым шагом пользователь запускает бинарный файл визуализатора командой “make run_visualiser” с флагом выбора графика, что показано на Рисунке 5.

```
slavaruswarrior@Air-Vaceslav siv % make run_visualizer
./visualizer/cmake-build-debug/ConsoleGraph < ./visualizer/visfifo
```

[6] 0:make*

"Air-Vaceslav.Dlink" 06:05 25-Dec-23

Рисунок 5 — Запуск визуализатора

Шестым шагом пользователь запускает клиента для генерации данных с нужными ему параметрами схемы: тип графика, количество целочисленных данных в посылке, максимальное возможное значение, период времени, число посылок (Рисунок 6).

```
slavaruswarrior@Air-Vaceslav siv % make run_client
cd ssh_testing/client && go run cmd/client.go
choose schema: waterfall | flashing
waterfall
Scan: flowSize | waterfallSize | maxAmplitude | period ms | batchCount
50 20 100 250 100
```

Рисунок 6 — Запуск клиента

Седьмым шагом начинают генерироваться данные, которые клиент передает в fifo визуализатора (Рисунок 7).

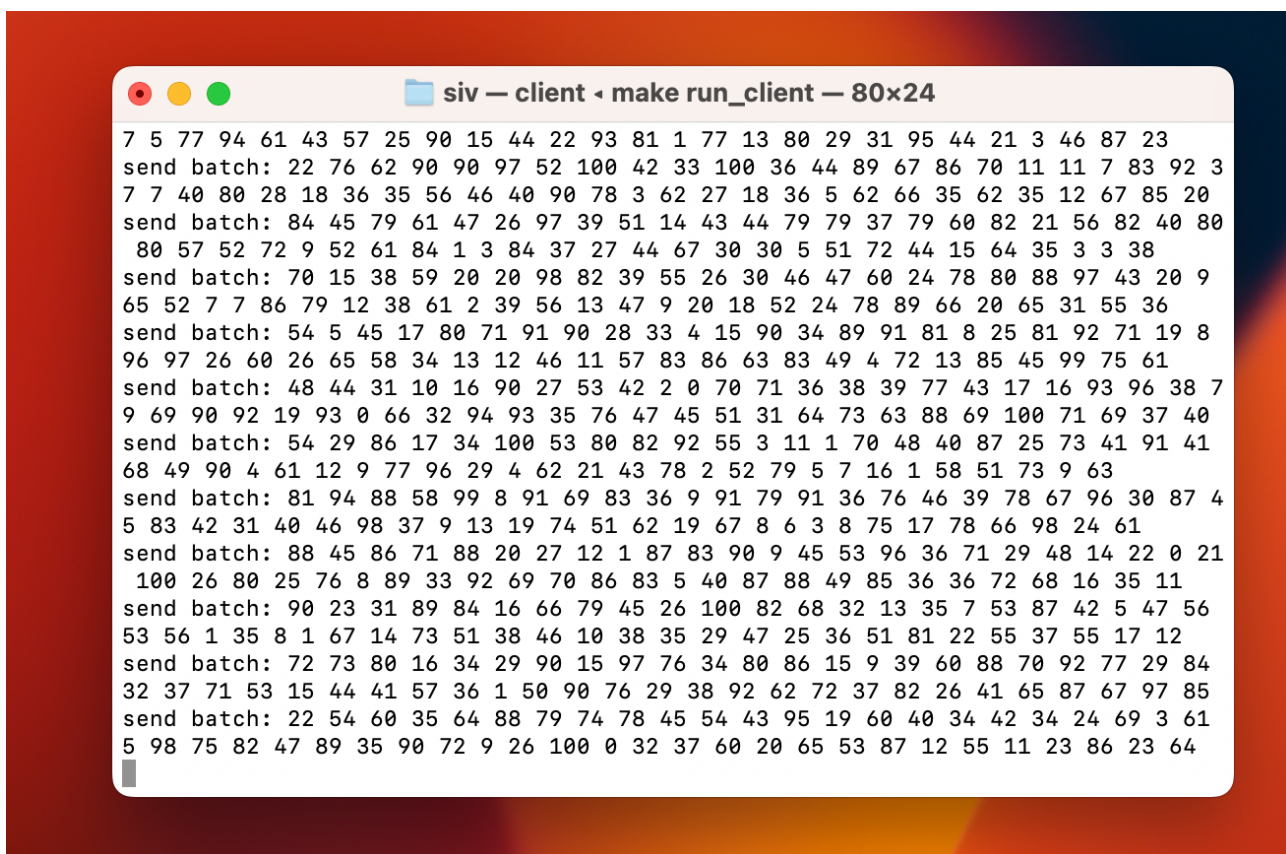


Рисунок 7 — Генерация данных

Восьмым шагом визуализатор принимает и обрабатывает данные, генерирует окно в терминале и начинает выводить на экран информацию (Рисунок 8).

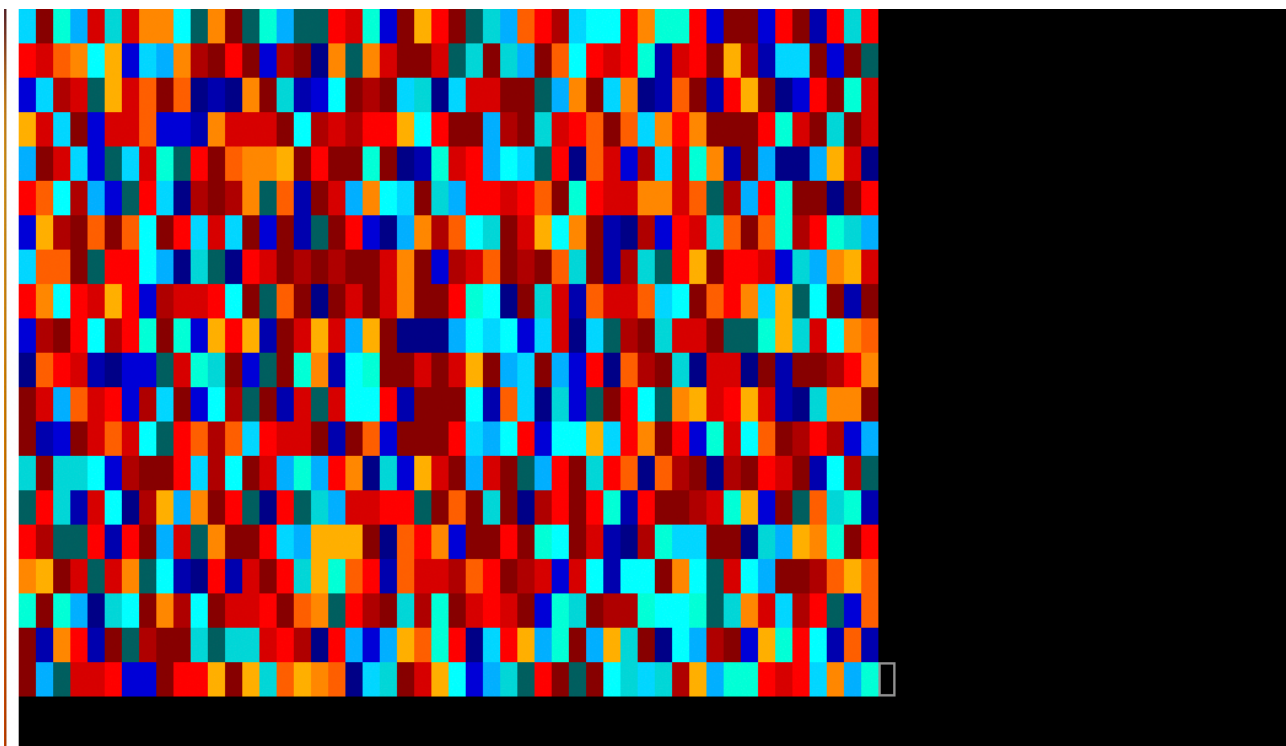


Рисунок 8 — Работа визуализатора

Девятым шагом, после того как число посылок в клиенте закончится, он пошлет массив из отрицательных чисел, что означает окончание ввода (по условию программы мы визуализируем только положительные значения чисел).

Десятым шагом визуализатор очистит буфер, закроет окно, и пользователь окажется на сервере. Для выхода с сервера нужно дважды прописать `exit`.

4 Тестирование

Для тестирования работоспособности программы нужно выполнить шаги из пункта 3.5. Для достоверности отображения графика заведём канонический файл данных и рассмотрим его работу локально и на сервере. Для этого был написан файл с длиной массива 50 и числом строк 30. Пусть высота водопада будет 20, а максимальное число будет 70. Для этого я определил функцию считывания этих данных.

Результат N-ой секунды локально представлены на Рисунке 9.



Рисунок 9 — Тест локально

Результат N-ой секунды на сервере представлены на Рисунке 10.

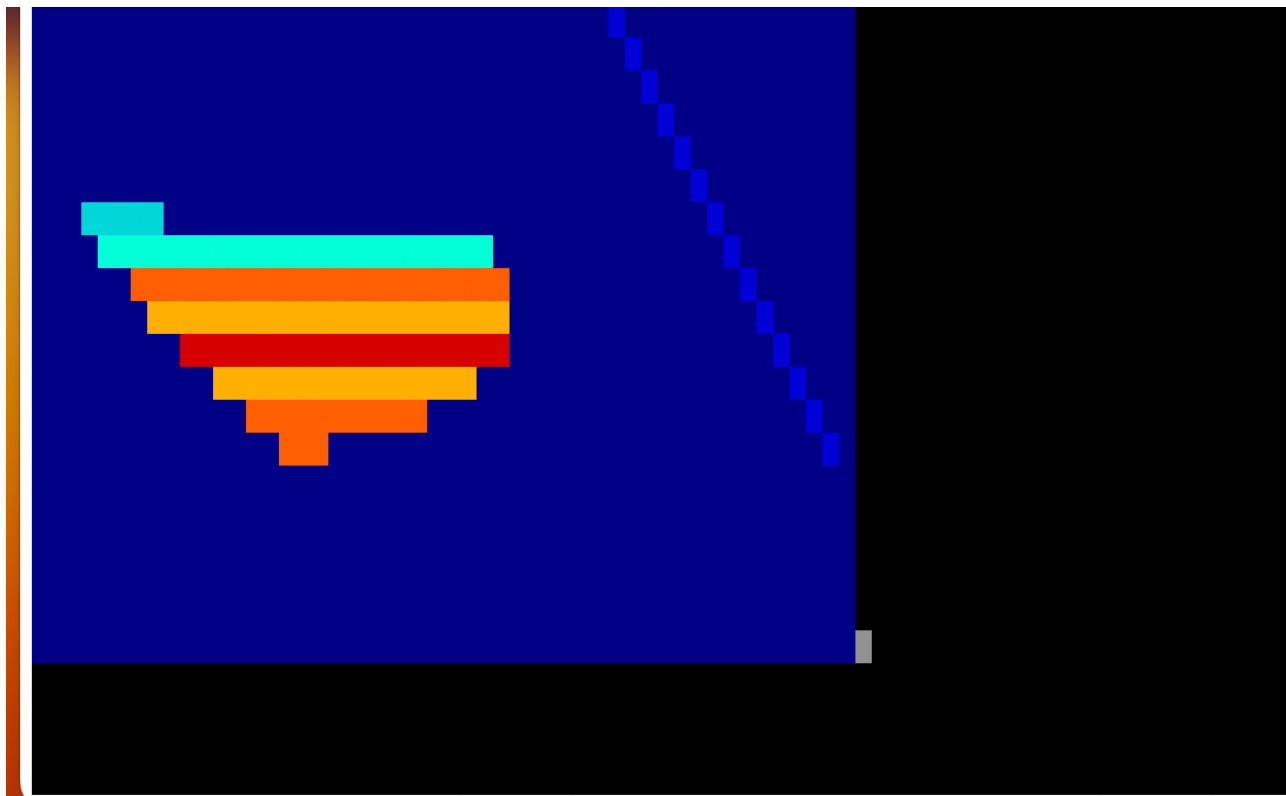


Рисунок 10 — Тест на сервере

По итогу тестирования из рисунков мы видим, что поведение локально и на сервере совпадает и это поведение корректно. Тем самым было доказано условие корректного управления программы на сервере.

Кроме того, визуализатор демонстрирует плавную работу при различных периодах отправления данных, включая интервалы до 200 миллисекунд. Этот результат является весьма обнадеживающим, поскольку плавная и эффективная работа визуализатора при таких временных интервалах представляет собой важный аспект пользовательского опыта.

Этот результат свидетельствует о надежности механизмов взаимодействия между клиентом и сервером, а также об эффективной реализации протокола SSH. Такой подход в тестировании укрепляет уверенность в том, что приложение работает стабильно и предсказуемо в различных условиях.

Приведем так же пример того, как выглядит совместный запуск сервера и клиента на Рисунке 11.

Рисунок 11 — Совместный запуск

Таким образом, тестирование не только не выявило возможных проблем, но и подтвердило то, что приложение готово к эффективному использованию в реальных условиях.

ЗАКЛЮЧЕНИЕ

В ходе выполнения данной курсовой работы были рассмотрены актуальные проблемы в области визуализации и анализа данных, особенно в условиях ограниченного доступа к графическим интерфейсам. С учетом экспоненциального роста объема данных становится критически важным иметь эффективные инструменты для работы с данными в различных сценариях, включая удаленные серверные среды и системы без графических окон.

Целью данной курсовой работы было разработать программу, предназначенную для операционных систем UNIX, способную визуализировать потоковые данные в терминале, не требуя использования X Windows System. Решение этой задачи предоставило новые возможности для мониторинга данных на удаленных серверах, в распределенных системах и микропроцессорных устройствах, где доступ к графическим интерфейсам ограничен или отсутствует.

В процессе разработки нами были использованы эффективные методы удаленного выполнения, что делает наше приложение удобным инструментом для анализа данных в различных сценариях использования. Полученный результат открывает новые перспективы в области визуализации данных в условиях, где не всегда доступны традиционные средства визуализации с использованием графических окон.

В итоге, разработанное приложение успешно решает поставленную задачу и представляет собой ценный инструмент для работы с данными в ограниченных по ресурсам и удаленных средах операционных систем UNIX.

У работы еще есть точки роста в виде лучшей обработки и анализа данных, подключения какой-либо базы данных для хранения информации, улучшенные генераторы для тестов и многое.

СПИСОК ЛИТЕРАТУРЫ

[1] Zhadchenko, A. V. Porting X windows system to operating system compliant with portable operating system interface / A. V. Zhadchenko, K. A. Mamrosenko, A. M. Giatsintov // International Journal of Advanced Computer Science and Applications. – 2020. – Vol. 11, No. 7. – P. 17-22.

[2] Исследование эффективности переключения окон в современных графических оболочках / Д. А. Костюк, К. Л. Костюк, С. С. Дереченник [и др.] // Вестник Брестского государственного технического университета. Физика, математика, информатика. – 2011. – № 5(71). – С. 45-48.

[3] Садовников, В. Ю. Оконный менеджер для создания интерактивной графической среды под управлением операционной системы с открытым исходным кодом / В. Ю. Садовников, С. А. Баскаков, Д. М. Шефер // Современное образование: содержание, технологии, качество. – 2014. – Т. 1. – С. 168.

[4] Катунин, Г. П. Основы инфокоммуникационных технологий : Учебник / Г. П. Катунин. – Саратов : Ай Пи Эр Медиа, 2018. – 797 с. – ISBN 978-5-4486-0335-8.

[5] Дмитриев, В. Л. Перенаправление стандартных потоков ввода-вывода в C++ / В. Л. Дмитриев, С. П. Суханов // Теория и практика современной науки. – 2016. – № 12-1(18). – С. 383-386.

[6] Исследование способов взаимодействия сетевых устройств на базе микрокомпьютеров / И. Н. Бабков, К. А. Пудов, В. В. Коновалова, Г. М. Дибиров // Научные известия. – 2022. – № 26. – С. 35-38.

[7] Яковенко, П. Н. Обеспечение конфиденциальности информации, обрабатываемой на компьютере с сетевым подключением / П. Н. Яковенко // Проблемы информационной безопасности. Компьютерные системы. – 2009. – № 4. – С. 23-41.

[8] Internet of things as a complement to increase safety / E. Larsson, E. Bratt, J. Palmqvist [et al.] // Журнал Белорусского государственного университета. Международные отношения. – 2020. – No. 1. – P. 88-93.

[9] Артамонов, Ю. Н. Модульное проектирование программных приложений : Учебное пособие / Ю. Н. Артамонов, В. А. Докучаев, С. В. Шевелев. – Москва : Ай Пи Ар Медиа, 2023. – 172 с. – ISBN 978-5-4497-2116-7.

[10] Средства отладки программного обеспечения многоядерных процессоров / С. Г. Елизаров, Г. А. Лукьянченко, Д. С. Марков, В. А. Роганов // Программная инженерия. – 2017. – Т. 8, № 12. – С. 531-542. – DOI 10.17587/prin.8.531-542.

[11] Бречка, Д. М. Операционные системы : учебно-методическое пособие в 3 частях / Д. М. Бречка. Том Часть 3. – Омск : Омский государственный университет им. Ф.М. Достоевского, 2014. – 84 с. – ISBN 978-5-7779-1793-5.

ПРИЛОЖЕНИЕ

Листинг 1 – SSH-сервер – функция запуска

```
package internal

import (
    "log"
    "net"

    "github.com/VyacheslavIsWorkingNow/siv/ssh_testing/se
rver/internal/handlers"
)

func RunServer() {
    pk, errPK := parsePrivateKey(privateKeyPath)
    if errPK != nil {
        log.Fatal("Failed to parse private key:", errPK)
    }

    config := NewConfig()

    config.AddHostKey(pk)

    listener, errL := net.Listen("tcp", "0.0.0.0:2222")
    if errL != nil {
        log.Fatal("Failed to listen:", errL)
    }
    defer func(listener net.Listener) {
        _ = listener.Close()
    }(listener)
```

```

log.Println("SSH server listening on :2222...")

for {
    conn, err := listener.Accept()
    if err != nil {
        log.Printf("Failed to accept connection: %v",
err)
        continue
    }

    go handlers.HandleSSHConnection(conn, config)
}
}

```

Листинг 2 – Функции обработки на SSH-сервере

```

package handlers

import (
    "fmt"
    "github.com/creack/pty"
    "golang.org/x/crypto/ssh"
    "golang.org/x/term"
    "io"
    "log"
    "net"
    "os"
    "os/exec"
)

```

```

func HandleSSHConnection(conn net.Conn, config
*ssh.ServerConfig) {

    sshConn, chans, reqs, err := ssh.NewServerConn(conn,
config)
    if err != nil {
        log.Printf("Failed to handshake: %v", err)
        return
    }
    defer func(sshConn *ssh.ServerConn) {
        _ = sshConn.Close()
    }(sshConn)

    log.Printf("New SSH connection from %s",
sshConn.RemoteAddr())

    go ssh.DiscardRequests(reqs)
    for newChannel := range chans {
        if newChannel.ChannelType() != "session" {
            err =
newChannel.Reject(ssh.UnknownChannelType, "unknown
channel type")
            if err != nil {
                return
            }
            continue
        }

        channel, _, errA := newChannel.Accept()
        if errA != nil {

```

```

        log.Printf("Failed to accept channel: %v",
errA)

        return
    }

    go handleSSHSession(channel)
}

func handleSSHSession(channel ssh.Channel) {
defer func(channel ssh.Channel) {
    _ = channel.Close()
}(channel)

    _, _ = fmt.Fprintf(channel, "Welcome to vis SSH
server!\n\r")

    terminal := term.NewTerminal(channel, "> ")

    for {
        line, err := terminal.ReadLine()
        if err != nil {
            _, _ = fmt.Fprintf(channel, "error read line
%e\n", err)
            break
        }

        if errH := handler(channel, line); errH != nil {
            log.Printf("error in handler %+v", errH)
        } else {

```

```

        _, _ = fmt.Fprintf(channel, "handle %s\n\r", line)
    }
}

_, _ = fmt.Fprintf(channel, "closing session")

}

func handler(channel ssh.Channel, command string) error {
    switch command {
    case "tmux":
        return handleTmux(channel)
    case "exit":
        return handleExit(channel)
    default:
        _, _ = fmt.Fprintf(channel, "Unknown command: %s\n\r", command)
    }

    return nil
}

func handleTmux(channel ssh.Channel) error {
    _, _ = fmt.Fprintf(channel, "Upload binary to remoute server\n\r")

    if err := startTmux(channel); err != nil {
        return fmt.Errorf("failed to upload bin %w", err)
    }

    return nil
}

```

```

    }

    func startTmux(channel ssh.Channel) error {
        cmd := exec.Command("tmux")

        ptmx, errS := pty.Start(cmd)
        if errS != nil {
            return fmt.Errorf("failed to start tmux: %v",
errS)
        }
        defer func(ptmx *os.File) {
            _ = ptmx.Close()
        }(ptmx)

        go func() {
            _, _ = io.Copy(channel, ptmx)
        }()

        go func() {
            _, _ = io.Copy(ptmx, channel)
        }()

        if errW := cmd.Wait(); errW != nil {
            return fmt.Errorf("tmux failed: %v", errW)
        }

        return nil
    }

    func handleExit(channel ssh.Channel) error {

```

```
_ , _ = fmt.Fprintf(channel, "Stoping server\n\r")
os.Exit(0)
return nil
}
```

Листинг 3 – Обработка ключей для SSH-сервера

```
package internal

import (
    "fmt"
    "os"

    "golang.org/x/crypto/ssh"
)

func parsePrivateKey(keyPath string) (ssh.Signer,
error) {
    privateBytes, errR := os.ReadFile(keyPath)
    if errR != nil {
        return nil, fmt.Errorf("failed read ssh key %w",
errR)
    }

    pk, errPPK := ssh.ParsePrivateKey(privateBytes)
    if errPPK != nil {
        return nil, fmt.Errorf("failed parse ssh key %w",
errPPK)
    }

    return pk, nil
}
```

```
}
```

Листинг 4 – Функция визуализации графика waterfall

```
#include <ncurses.h>
#include <cmath>
#include <iostream>
#include <fstream>

const int num_colors = 18;

int* scan_flow(int size) {
    int* array = new int[size];

    for (int i = 0; i < size; ++i) {
        std::cin >> array[i];
    }

    return array;
}

bool is_end_of_flow(int a) {
    return a < 0;
}

void shift_and_insert(int** array2D, int* newValues,
int M, int N) {

    delete(array2D[M-1]);

    for (int i = M - 1; i > 0; --i) {
        array2D[i] = array2D[i - 1];
```



```

    }

    array2D[0] = newValues;
}

void print_buffer(int* buffer[], int M, int N) {
    for (int i = 0; i < M; ++i) {
        for (int j = 0; j < N; ++j) {
            std::cout << buffer[i][j] << " ";
        }
        std::cout << "\n";
    }
}

//Оттенки синего: Темный синий: 18 | Средний синий: 19
| Светлый синий: 20 | Голубой: 23 | Светло-голубой: 39
// Зеленовато-голубой: 44 | Бирюзовый: 45 | Бирюзово-
зеленый: 50 | Зеленовато-бирюзовый: 51

//Оттенки красного: Светлый красный: 196 | Оранжево-
красный: 202 | Оранжевый: 208 | Темный оранжевый: 214
// Красновато-коричневый: 160 | Темно-красный: 124 |
Красный: 160 | Темный красный: 196 | Красно-коричневый: 88

void init_color_pair() {

    int pair_number = 1;
    int blueShades[] = {18, 19, 20, 23, 39, 44, 45,
50, 51};

```

```

        int redShades[] = {196, 202, 208, 214, 160, 124,
160, 196, 88};

        for (const auto& bc: blueShades) {
            init_pair(pair_number, bc, bc);
            pair_number++;
        }

        for (const auto& rs: redShades) {
            init_pair(pair_number, rs, rs);
            pair_number++;
        }
    }

    int get_color_index(int value, int max_value) {
        int step = max_value / (num_colors - 1);

        int color_index = value / step + 1;

        if (color_index < 1) {
            color_index = 1;
        } else if (color_index > num_colors) {
            color_index = num_colors;
        }

        return color_index;
    }

    void draw_buffer(int* waterfall_buffer[], int
waterfall_size, int flow_size, int max_amplitude) {

```

```

        clear();

        for (int i = 0; i < waterfall_size; ++i) {
            for (int j = 0; j < flow_size; ++j) {
                int value = waterfall_buffer[i][j];

                int colorIndex = get_color_index(value,
max_amplitude);

                attron(COLOR_PAIR(colorIndex));
                mvprintw(i, j, "#");
                attroff(COLOR_PAIR(colorIndex));
            }
        }

        refresh();
    }

    void waterfall_buffer(int flow_size, int
waterfall_size, int max_amplitude, int period, WINDOW*
win) {
        int* waterfall_buffer[waterfall_size];
        for (int i = 1; i < waterfall_size; i++) {
            waterfall_buffer[i] = new int[flow_size]{};
        }

        std::fstream fifoStream("./visfifo",
std::ios::in);

        waterfall_buffer[0] = scan_flow(flow_size);
    }

```

```

        while (true) {
            int* flow = scan_flow(flow_size);
            if (is_end_of_flow(flow[0])) {
                break;
            }
            shift_and_insert(waterfall_buffer, flow,
waterfall_size, flow_size);

            // real buffer
            //          print_buffer(waterfall_buffer,
waterfall_size, flow_size);

            // ncurses buffer
            draw_buffer(waterfall_buffer, waterfall_size,
flow_size, max_amplitude);

            fifoStream.close();
            fifoStream.open("./visfifo");
        }

        endwin();

        for (int i = 0; i < waterfall_size; i++) {
            delete(waterfall_buffer[i]);
        }
    }

    void waterfall() {

```

```

        int    flow_size,    waterfall_size,    max_amplitude,
period;

        std::cout << "waterfall start" << std::endl <<
"write schema: " << std::endl;
        std::cout << "flow size | " << "waterfall size |
" << "max amplitude | " << "period | " << std::endl;
        std::fstream                                fifoStream("./visfifo",
std::ios::in);
        std::cin  >>  flow_size  >>  waterfall_size  >>
max_amplitude >> period;
        fifoStream.close();
        fifoStream.open("./visfifo");

        std::cout << "your schema:" << "\nfs: " <<
flow_size << "\nws: " << waterfall_size <<
        "\nma: " << max_amplitude << "\np: " << period <<
std::endl;

        initscr();
        start_color();

        init_color_pair();

        WINDOW* win = newwin(flow_size, waterfall_size, 0,
0);

        waterfall_buffer(flow_size,                waterfall_size,
max_amplitude, period, win);

```

```
        endwin();  
    }
```

Листинг 5 – Генератор тестовых данных

```
package datagen  
  
import (  
    "fmt"  
    "math/rand"  
    "time"  
)  
  
type Batch struct {  
    Time    time.Time  
    Length  int  
    Data    []int  
}  
  
type GeneratorMode int  
  
const (  
    RandomMode GeneratorMode = iota  
    AmplitudeK1Mode  
    AmplitudeK2Mode  
)  
  
func DataGenerator(t time.Duration, l, N, K1, K2 int,  
mode GeneratorMode) <-chan Batch {  
    dataStream := make(chan Batch)  
  
    go func() {
```

```

defer close(dataStream)

rand.NewSource(time.Now().UnixNano())

for {
    time.Sleep(t)
    batch := Batch{
        Time:    time.Now(),
        Length: 1,
    }

    for i := 0; i < 1; i++ {
        var value int

        switch mode {
        case RandomMode:
            value = rand.Intn(N + 1)
        case AmplitudeK1Mode:
            value = rand.Intn(K1 + 1)
        case AmplitudeK2Mode:
            value = rand.Intn(K2 + 1)
        }

        batch.Data = append(batch.Data, value)
    }

    dataStream <- batch
}

}()

return dataStream

```

```
}
```

Листинг 6 – Клиент для доставки тестовых данных

```
package main

import (
    "fmt"
    "github.com/VyacheslavIsWorkingNow/siv/ssh_testing/client/internal"
    "github.com/VyacheslavIsWorkingNow/siv/ssh_testing/client/internal/datagen"
    "log"
    "os"
    "strings"
    "time"
)

const (
    waterfall      = "waterfall"
    flashing        = "flashing"
    fifoFileName    =
"/Users/slavaruswarrior/Documents/GitHub/siv/visualizer/visfifo"
)

func main() {

    var userSchema string

    fmt.Printf("choose schema: waterfall | flashing\n")
```



```

_, _ = fmt.Scanf("%s\n", &userSchema)

if userSchema == waterfall {
    w := scanWaterfallSchema()
    errW := writeDataToFifo([]int{w.FlowSize,
w.WaterfallSize, w.MaxAmplitude, w.Period})
    if errW != nil {
        return
    }
    generator := datagen.DataGenerator(
        time.Duration(w.Period)*time.Millisecond,
w.FlowSize, w.MaxAmplitude, 0, 0, datagen.RandomMode)
    for i := 0; i < w.BatchCount; i++ {
        batch := <-generator
        err := writeDataToFifo(batch.Data)
        if err != nil {
            log.Fatalf("failed to send data to fifo
in waterfall %e", err)
        }
    }
    errW =
writeDataToFifo(generateEndData(w.FlowSize))
    if errW != nil {
        log.Fatalf("can't write end data")
    }
} else if userSchema == flashing {
    f := scanFlashingSchema()
    errW := writeDataToFifo([]int{f.FlowSize,
f.MaxAmplitude, f.Period})
    if errW != nil {

```

```

        log.Fatalf("can't write conf information")
    }
    generator := datagen.DataGenerator(
        time.Duration(f.Period)*time.Millisecond,
f.FlowSize, f.MaxAmplitude, 0, 0, datagen.RandomMode)
    for i := 0; i < f.BatchCount; i++ {
        batch := <-generator
        err := writeDataToFifo(batch.Data)
        if err != nil {
            log.Fatalf("failed to send data to fifo
in waterfall %e", err)
        }
    }
    errW =
writeDataToFifo(generateEndData(f.FlowSize))
    if errW != nil {
        log.Fatalf("can't write end data")
    }
} else {
    log.Fatalf("wrong graph type")
}

canonicTest()

canonicNormalTest()

}

func scanWaterfallSchema() *internal.Waterfall {

```

```

    var flowSize, waterfallSize, maxAmplitude, period,
batchCount int
    fmt.Println("Scan:  flowSize  |  waterfallSize  |
maxAmplitude | period ms | batchCount")
    _, _ = fmt.Scanf("%d %d %d %d %d", &flowSize,
&waterfallSize, &maxAmplitude, &period, &batchCount)
    w := &internal.Waterfall{
        FlowSize:      flowSize,
        WaterfallSize: waterfallSize,
        MaxAmplitude:  maxAmplitude,
        Period:        period,
        BatchCount:    batchCount,
    }

    return w
}

func scanFlashingSchema() *internal.Flashing {
    var flowSize, maxAmplitude, batchCount int
    var period float64
    fmt.Println("Scan: flowSize | maxAmplitude | period |
batchCount")
    _, _ = fmt.Scanf("%d %d %d %f %d", &flowSize,
&maxAmplitude, &period, &batchCount)
    return nil
}

func writeDataToFifo(data []int) error {
    resultString                                     :=
strings.Join(strings.Fields(fmt.Sprint(data)), " ")

```

```

    resultString = resultString[1 : len(resultString)-1]

    fmt.Println("send batch:", resultString)
    fifoFile, err := os.OpenFile(fifoFileName,
os.O_WRONLY|os.O_CREATE, os.ModeNamedPipe)
    if err != nil {
        return fmt.Errorf("failed to open fifo file %w",
err)
    }
    defer func(fifoFile *os.File) {
        _ = fifoFile.Close()
    }(fifoFile)

    _, err = fifoFile.WriteString(resultString)
    if err != nil {
        return fmt.Errorf("failed to write fifo file %w",
err)
    }
    return nil
}

func generateEndData(N int) []int {
    arr := make([]int, N)
    arr[0] = -1
    return arr
}

func canonicTest() {
    w := &internal.Waterfall{
        FlowSize: 50,

```

```

        WaterfallSize: 20,
        MaxAmplitude: 70,
        Period:        250,
        BatchCount:    30,
    }

    errW      :=      writeDataToFifo([]int{w.FlowSize,
w.WaterfallSize, w.MaxAmplitude, w.Period})
    if errW != nil {
        return
    }
    for i := 0; i < w.BatchCount; i++ {
        err := writeDataToFifo(datagen.CanonicData[i])
        if err != nil {
            log.Fatalf("failed to send data to fifo in
waterfall %e", err)
        }
        time.Sleep(time.Millisecond *
time.Duration(w.Period))
    }
    errW = writeDataToFifo(generateEndData(w.FlowSize))
    if errW != nil {
        log.Fatalf("can't write end data")
    }
}

func canonicNormalTest() {
w := &internal.Waterfall{
    FlowSize:        30,
    WaterfallSize: 15,
    MaxAmplitude:    100,

```

```

        Period:          250,
        BatchCount:      15,
    }
    errW := writeDataToFifo([]int{w.FlowSize,
w.WaterfallSize, w.MaxAmplitude, w.Period})
    if errW != nil {
        return
    }
    for i := 0; i < w.BatchCount; i++ {
        err := writeDataToFifo(datagen.CanonicData[i])
        if err != nil {
            log.Fatalf("failed to send data to fifo in
waterfall %e", err)
        }
        time.Sleep(time.Millisecond *
time.Duration(w.Period))
    }
    errW = writeDataToFifo(generateEndData(w.FlowSize))
    if errW != nil {
        log.Fatalf("can't write end data")
    }
}

```