Оглавление

Введение

Удаленное подключение к терминалу позволяет пользователю управлять работой удаленного сервера или рабочей станции по сети. Такой подход может быть полезен, например, системным администраторам для администрирования выделенных серверов и технической поддержки пользователей. На основе удаленного подключения реализуется возможность делиться сессией терминала, то есть использовать терминал несколькими пользователями из разных сетей одновременно. С помощью этой возможности можно совместно редактировать файлы, отлаживать какой-либо проект командой разработчиков и так далее.

Целью данной работой является разработка программного обеспечения, позволяющего удаленно подключения терминалу нескольких пользователей и его редактирования. Для достижения поставленной цели необходимо решить следующие задачи:

— провести анализ существующих решений;

— изучить существующие протоколы прикладного уровня;

— разработать свой протокол прикладного уровня для поставленной задачи;

— реализовать программное обеспечение (сервер и клиент) с использованием разработанного протокола для удаленного подключения к терминалу.

1 Аналитическая часть

В данном разделе проводится сравнительный анализ методов решения поставленной задачи, делается обоснованный выбор метода.

1.1 Постановка задачи

В соответствии с заданием необходимо разработать программное обеспечение для удаленного подключения к терминалу: серверное и клиентское приложение. Сервер должен поддерживать связь с множеством клиентов одновременно, обрабатывая запросы от каждого и рассылая изменения всем остальным. По необходимости сервер должен запустить команду терминала, высылая результат всем подключенным клиентам. Для решения этой задачи необходимо изучить предметную область и проанализировать существующие решения.

1.2 Анализ существующих решений

1.2.1 tty-share

`tty-share` [1] - инструмент, используемый для совместного редактирования терминала. Сеансом можно поделиться как в локальной сети, так и в удаленной. При совместном использовании терминала через сеть Internet, tty-share подключается к прокси-серверу, который является посредником при обмене данными между участниками. Экземпляр этого сервера может быть размещен как на вашей машине, так и на `tty-share.com`. Сквозное шифрование при таком подключении отсутствует.

К достоинствам tty-share можно отнести:

- возможность подключения к сессии из браузера;
- отсутствие зависимостей. tty-share представляет из себя статический кроссплатформенный двоичный файл, который не содержит зависимостей;

1.2.2 tmate

`tmate` [2] – ответвление терминального мультиплексора tmux [3], который обеспечивает безопасное, мгновенное и простое в использовании решение для совместного использования терминалов с помощью ssh-соединения [4]. На рисунке 1.1 представлена схема архитектуры tmate с различными компонентами.

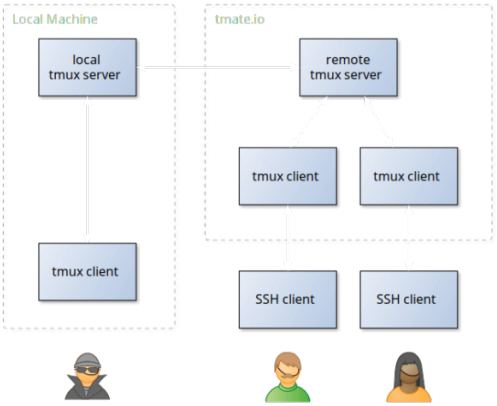

Рис. 1.1: Схема архитектуры tmate. Слева представлена архитектура при запуске в локальной сети, справа – в удаленной сети

Локальный, как и удаленный сервер tmux взаимодействует с протоколом поверх стерилизатора данных msgpack [5], который сжимается через ssh-соединение для повышения эффективности пропускной способности сети [2].

К достоинствам tmate можно отнести:

- возможность подключения как в локальной сети, так и в удаленной;
- использование ssh-сессий;
- возможность разместить tmate-сервер как на своей машине, так и на серверах `tmate.io`.

1.3 Модель клиент-сервер

В модели клиент-сервер роли определены: сервер предоставляет ресурсы и службы одному или нескольким клиентам, которые обращаются

к серверу за обслуживанием. В качестве примеров серверов можно привести веб-серверы, почтовые серверы и файловые серверы . Каждый из этих серверов предоставляет ресурсы для клиентских устройств, таких как настольные компьютеры, ноутбуки, планшеты и смартфоны . Большинство серверов могут устанавливать отношение «один ко многим» с клиентами, что означает, что один сервер может предоставлять ресурсы нескольким клиентам одновременно. Когда клиент запрашивает соединение с сервером, сервер может либо принять, либо отклонить это соединение. Если соединение принято, сервер устанавливает и поддерживает соединение с клиентом по определенному протоколу. Например, почтовый клиент может запросить SMTP-соединение с почтовым сервером для отправки сообщения. Затем приложение SMTP на почтовом сервере запросит проверку подлинности у клиента, например адрес электронной почты и пароль. Если эти учетные данные совпадают с учетной записью на почтовом сервере, сервер отправит электронное письмо целевому получателю. Часто клиенты и серверы взаимодействуют через компьютерную сеть на разных аппаратных средствах, но и клиент и сервер могут находиться в одной и той же системе. Хост сервера запускает одну или несколько серверных программ, которые совместно используют свои ресурсы с клиентами. Клиент не предоставляет общий доступ ни к одному из своих ресурсов, но запрашивает данные или службу у сервера. Поэтому клиенты инициируют сеансы связи с серверами, которые ожидают входящих запросов. Клиенту не знает о том, как работает сервер при выполнении запроса и доставке ответа. Клиент должен только понимать ответ, основанный на хорошо известном прикладном протоколе, т.е. содержание и форматирование данных для запрашиваемой службы. Клиенты и серверы обмениваются сообщениями в шаблоне обмена сообщениями запрос-ответ. Клиент отправляет запрос, а сервер возвращает ответ.

1.4 Сокеты

Сокет (англ. socket [6])— конечный пункт передачи данных, абстракция, обозначающая одно из окончаний сетевого соединения. Каждая из программ, устанавливаю щих соединение, должна иметь собственный сокет.

Сокеты – это интерфейс прикладного программирования для сетевых приложений TCP/IP. Интерфейс сокетов был создан в восьмидесятых годах для операционной системы UNIX. Позднее интерфейс сокетов был перенесен в Microsoft Windows. Сокеты до сих пор используются в приложениях для сетей TCP/IP. Сетевые приложения используют сокеты, как виртуальные разъемы для обмена данными между собой. Сокеты бывают трех видов: клиентские, слушающие и серверные.

1.4.1 Принципы сокетов

Каждый процесс может создать слушающий сокет (серверный сокет) и привязать его к какому-нибудь порту операционной системы (в UNIX непривилегированные процессы не могут использовать порты меньше 1024). Слушающий процесс обычно находится в цикле ожидания, то есть просыпается при появлении нового соединения. При этом сохраняется возможность проверить наличие соединений на данный момент, установить таймаут для операции и т.д. Каждый сокет имеет свой адрес. ОС семейства UNIX могут поддерживать много типов адресов, но обязательными являются INET-адрес и UNIX адрес. Если привязать сокет к UNIX-адресу, то будет создан специальный файл (файл сокета) по заданному пути, через который смогут сообщаться любые локальные процессы путём чтения/записи из него. Сокеты типа INET доступны из сети и требуют выделения номера порта [7]. Обычно клиент явно подсоединяется к слушателю, после чего любое чтение или запись через его файловый дескриптор будут передавать данные между ним и сервером.

1.4.2 Основные функции сокетов

В таблицах 1.1 - 1.3 представлены общие функции, функции сервера и клиента соотвественно.

Функция	Описание
`socket`	Создать новый сокет и вернуть файловый дескриптор
`send`	Отправить данные по сети
`receive`	Получить данные по сети
`close`	Закрыть соединение

Таблица 1.1: Общие функции

Функция	Описание
`bind`	Связать сокет с IP-адресом и портом
`listen`	Слушает порт и ждет когда будет установлено соединение
`accept`	Принять запрос на установку соединения

Таблица 1.2: Функции сервера

1.4.3 Типы сокетов

Различают два типа сокетов:

— сокеты с предварительным установлением соединения, когда до начала передачи данных устанавливаются адреса сокетов отправителя и получателя данных – такие сокеты соединяются друг с другом и остаются соединенными до окончания обмена данными;

— сокеты без установления соединения, когда соединение до начала передачи данных не устанавливается, а адреса сокетов отправителя и получателя передаются с каждым сообщением.

Если тип сокета – виртуальный канал, то сокет должен устанавливать соединение, если же тип сокета – датаграмма, то, как правило, это сокет без установления соединения, хотя последнее не является требованием.

1.5 Обработка запросов клиентов

При приходе клиентского запроса у сервера имеется несколько вариантов действий:

Функция	Описание
`connect`	Установить соединение

Таблица 1.3: Функции клиента

— обрабатывать все запросы в одном потоке;

— обрабатывать каждый запрос в отдельном потоке;

— организовать пул потоков.

1.5.1 Обработка всех запросов в одном потоке

Это решение подходит только для ограниченного числа случаев, в которых количество клиентов невелико, и обращаются они к серверу не часто. Эта самая простая схема работы: минимум потоков, минимум ресурсов, нет синхронизации. Необходимо построить очередь входящих запросов, чтобы они не терялись при последовательной обработке. Но при большом количестве клиентов, клиенты будут долго ждать ответа от сервера.

1.5.2 Обработка каждого запроса в отдельном потоке

Для каждого клиентского запроса создается отдельный поток. Такая схема работает следующим образом: первичный поток приложения прослушивает клиентские запросы и при поступлении каждого создает новый поток, передавая ему клиентский пакет (данные или команду). Созданный поток выполняет соответствующую обработку, передает результаты обратно клиенту, или же помещает их в БД, и завершает свое существование.

Основные недостатки такой модели:

— частое создание и завершение потоков;

— малое время работы потока;

— нерегулируемое количество потоков;

— в большинстве случаев отсутствие очереди клиентских запросов;

— большое количество переключений контекстов рабочих потоков.

Для решения этих проблем и предназначен пул потоков.

1.5.3 Организация пула потоков

Имеется главный поток приложения, прослушивающий клиентские запросы. Пул потоков создается заранее или при поступлении первого запроса. При поступлении запроса главный поток выбирает поток из пула и передает ему запрос. Если все потоки занимаются обработкой, то есть активны, пакет ставится в очередь и ждет освобождения одного из потоков. Алгоритмы добавления потоков в пул и определения оптимального размера пула сильно зависят от решаемой задачи [8].

1.6 Протоколы

Протокол – набор правил, определяющих взаимодействие двух одноименных уровней модели взаимодействия открытых систем в различных абонентских ЭВМ. Стек протоколов — набор взаимодействующих сетевых протоколов.

Наиболее популярные стеки протоколов: TCP/IP, IPX/SPX, NetBIOS/SMB, DECnet, SNA и OSI. Большинство протоколов (все из перечисленных, кроме SNA) одинаковы на физическом и канальном уровне, но на других уровнях как правило используют разные протоколы.

1.6.1 Протоколы транспортного уровня

Протоколы транспортного уровня предназначены для обеспечения непосредственного информационного обмена между двумя пользовательскими процессами. Существует два типа протоколов транспортного уровня

– сегментирующие протоколы и не сегментирующие протоколы доставки дейтаграмм [9].

Протоколы доставки дейтаграмм просты для реализации, однако, не обеспечивают гарантированной и достоверной доставки сообщений.

В качестве протоколов транспортного уровня в сети Internet могут быть использованы два протокола:

— UDP – User Datagram Protocol;

— TCP – Transmission Control Protocol.

Описание принципов построения протокола UDP приведено в RFC 768. Для передачи сообщений UDP используются пакеты IP. Сообщения UDP в данном случае размещаются в поле данных переносящего их пакета.

Протокол TCP используется для обеспечения надежного информационного обмена на транспортном уровне в сетях Internet.

Существует достаточно много причин, которые могут помешать пакету, который передается в сети, успешно достичь станции назначения. Таким образом, если не будут использованы специальные методы для обеспечения гарантированной доставки, принятое сообщение может существенным образом отличаться от того сообщения, которое было передано.

Надежный информационный обмен предполагает следующие возможности:

— потоковый обмен;

— использование виртуальных соединений;

— буферизированная передача данных;

— неструктурированный поток;

— обмен в режиме полного дуплекса.

1.6.2 Протоколы прикладного уровня

Протоколы прикладного уровня описывают взаимодействие между клиентской и серверной частями программы. Прикладные протоколы работа-

ют на верхнем уровне модели OSI. Они обеспечивают взаимодействие приложений и обмен данными между ними.

Далее будет подробнее рассмотрен протокол HTTP.

«Классическая» схема HTTP-сеанса выглядит так:

— установление TCP-соединения;

— запрос клиента;

— ответ сервера;

— разрыв TCP-соединения.

Запрос клиента содержит в себе заголовок и тело запроса. Заголовок содержит в себе строку состояния, которая имеет следующий формат: `метод запроса, URL ресурса, версия протокола HTTP`. Метод, указанный в строке состояния, определяет способ взаимодействия на ресурс, URL которого задан в той же строке. Методе может принимать значения GET, POST, HEAD, PUT, DELETE и т.д.

Ответ сервера клиенту начинается со строки состояния, которая имеет следующую форму: `версия протокола, код ответа, сообщение`. Версия протокола задается в том же формате, что и в запросе клиента. Код ответа – трехзначное десятичное число, представляющее в закодированном виде результат обслуживания запроса сервером. Пояснительное сообщение дублируют код ответа в символьном виде.

Вывод

В поставленной необходимо чтобы пакет, отправленный клиентом, гарантировано доходил до сервера и данные не терялись при передаче. Таким образом, в качестве протокола транспортного уровня был выбран протокол TCP. Были проанализированы аналоги и рассмотрены особенности существующих протоколов прикладного уровня для дальнейшего проектирования собственного протокола прикладного уровня. В качестве механизма обработки клиентов был выбрана обработка запросов в одном потоке.

2 Конструкторская часть

В данном разделе рассматривается процесс проектирования структуры программного обеспечения.

2.1 Состав программного обеспечения

Программное обеспечение состоит из клиент-серверного приложения. Структура разрабатываемого проекта:

— сервер – программа, обрабатывающая запросы клиентов и являющаяся посредником для передачи информации от одного клиента всем остальным;

— клиент – программа, которая является инициатором соединения и способная генерировать события.

2.2 Функциональная модель

На рисунках 2.1 и 2.2 представлена функциональная модель IDEF0 нулевого и первого уровня соответственно.

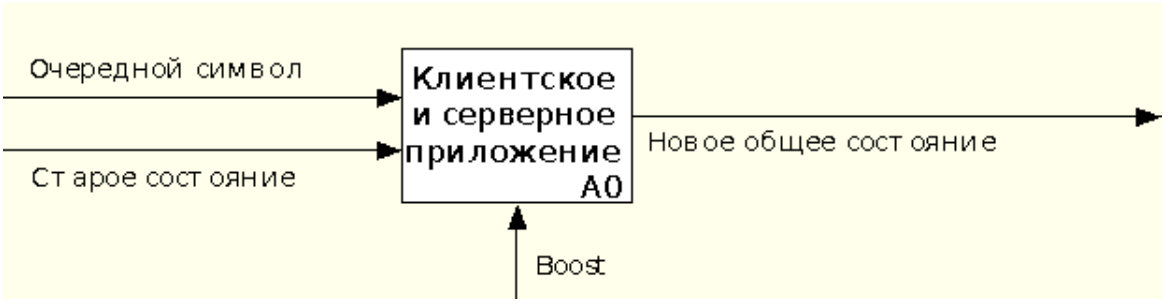

Рис. 2.1: Функциональная модель IDEF0 нулевого уровня

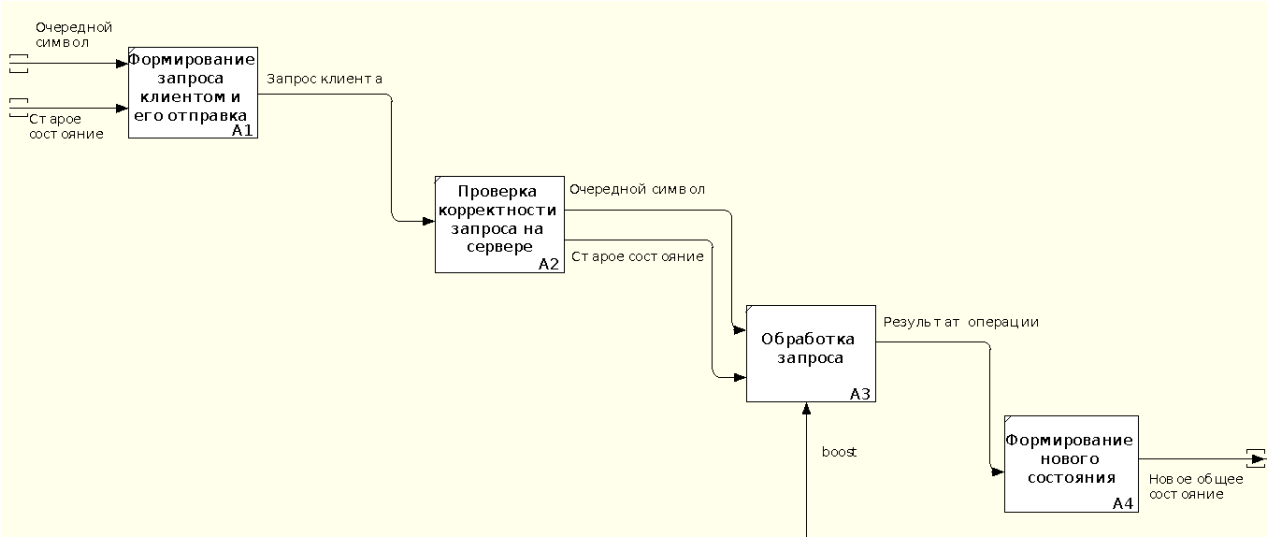

Рис. 2.2: Функциональная модель IDEF0 первого уровня

2.3 Сценарий использования

На рисунке 2.3 представлены действия доступные клиенту.

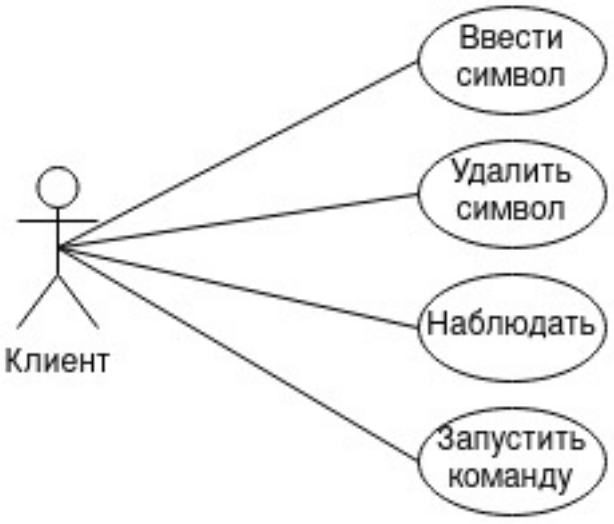

Рис. 2.3: Use-case диаграмма

2.4 Проектирование протокола прикладного уровня

Передаваемый пакет от клиента к серверу имеет следующую структуру: `пакет = (размер, сообщение)`, где `сообщение = (тип, данные)`. В свою очередь, тип может быть равен значению `getAll` либо `addChar`.

- тип запроса `getAll` сообщает серверу о необходимости вернуть клиенту изменяемое информацию о состоянии системы. Поле данных в таком случае в отправляемом пакете не заполняется;
- тип запроса `addChar` говорит о необходимости добавить передаваемый символ на экран и сообщить о добавлении этого символа всем подключенным клиентам. В таком случае поле данных содержат добавляемый символ и его позицию на экране.

Если клиент хочет добавить символ, он отправляет эту информацию серверу (запрос с типом `addChar`), сервер проверяет такой запрос на валидность, добавляет символ и сообщает всем подключенным к нему клиентам – то есть передает новое состояние системы. Сервер может передавать клиентам лишь состояние системы.

Коды ошибок не предусмотрены – в случае какой-либо ошибки клиент об этом узнать по средствам протокола не может.

Вывод

В данном разделе был спроектирована структура разрабатываемого программного обеспечения.

3 Технологическая часть

В данном разделе рассматривается выбор технологий для реализации поставленной задачи, листинги реализации разработанного программного обеспечения и приведены результаты работы ПО.

3.1 Язык программирования

Для реализации данного проекта выл выбран язык программирования C++ [10] по нескольким причинам:

— наличие для языка C++ необходимых для реализации задачи библиотек;

— скорость исполнения кода;

— хорошее знание языка C++;

3.2 ncurses

С помощью библиотеки `ncurses` [11], предназначенной для управления ввода-вывода на терминал и позволяющей задавать экранные координаты и цвет выводимых символов, реализуется окно терминала на стороне клиента. Библиотека `ncurses` была выбрана в связи с отсутствием хорошо задокументированных аналогов.

3.3 boost

С помощью набора библиотек `boost` [12] реализуется сериализация сообщений которые находятся в пакетах, а так же с запуск команд терминала. Данная библиотека была выбрана из-за большого опыта работы с ней.

3.4 Детали реализации

В листингах 3.1 - 3.5 представлены детали реализации разработанного программного обеспечения.

3.4.1 Обработчик сообщений на стороне сервера

В листинге 3.1 представлена реализация обработчика сообщений приходящих серверу.

```
template <typename Executer>
class MessageProcessor {
  ImmutableState imstate;
  MutableState mstate;
  CursesChProcessor processor;
  Executer executer;

  public:
  MessageProcessor() = default;
  bool operator()(const std::vector<char>& data, const int fd, const std::
   unordered_set<int>& all_fds);

  private:
  bool processAddChar(const Message& msg, const int fd, std::unordered_set<int
   > all_fds);
  bool processGetAll(const int fd);
};

template <typename Executer>
bool MessageProcessor<Executer>::operator()(const std::vector<char>& data,
    const int fd, const std::unordered_set<int>& all_fds)
{
  const std::string s(data.begin(), data.end());
  auto message = Message(s);
  switch (message.type) {
    case Message::Type::addChar:
      return processAddChar(message, fd, all_fds);
    case Message::Type::getAll:
      return processGetAll(fd);
    default:
      return false;
  }
  return false;
```

```
}

template <typename Executer>
bool MessageProcessor<Executer>::processAddChar(const Message& msg, const int
    fd, std::unordered_set<int> all_fds)
{
  auto m = AddCharMessage(msg.data);
  mstate.setCurpos(m.pos);
  auto res = processor.process(imstate, mstate, m.data);

  std::string packet;
  if (res == CursesChProcessor::ActionType::Exec) {
    auto command = mstate.getData();
    auto res = executer.execute(command);

    res.insert(res.begin(), "> " + command);
    for (auto& s : res)
    imstate.insertLine(s);

    auto diffMessage = DiffMessage(res);
    auto message = diffMessage.buildMessage();
    packet = message.buildPacket();
  } else {
    packet = msg.buildPacket();
  }

  if (res != CursesChProcessor::ActionType::BadKey)
    broadcastPacket(packet, all_fds);

  return true;
}

template <typename Executer>
bool MessageProcessor<Executer>::processGetAll(const int fd)
{
  auto message = GetAllMessage(imstate, mstate);
  auto msg = message.buildMessage();
  auto packet = msg.buildPacket();
  if (send(fd, packet.data(), packet.size(), 0) == -1) {
    std::cout << "Can't send data to client\n";
    return false;
  }
  return true;
}
```

Листинг 3.1: Обработчик сообщений (сервер)

3.4.2 Обработчик сообщений на стороне клиента

В листинге 3.2 представлена реализация обработчика сообщений на стороне клиента.

```
class MessageProcessor {
  public:
  void operator()(ImmutableState& imstate, MutableState& mstate, Message msg);

  private:
  void movePos(MutableState& msate, const std::size_t old_pos, const std::
    size_t pos, const int data);
};

void MessageProcessor::operator()(ImmutableState& imstate, MutableState&
    mstate, Message msg)
{
  if (msg.type == Message::Type::addChar) {
    const auto message = AddCharMessage(msg.data);
    const auto old_pos = mstate.getCurpos();

    mstate.setCurpos(message.pos);
    if (is_backspace(message.data))
      mstate.removeChar();
    else
      mstate.addChar(static_cast<char>(message.data));
      movePos(mstate, old_pos, message.pos, message.data);

  } else if (msg.type == Message::Type::diff) {
    auto message = DiffMessage(msg.data);
    mstate.getData();
    imstate.insertLines(std::move(message.diff));
  }
}

void MessageProcessor::movePos(MutableState& mstate, const std::size_t old_pos
    , const std::size_t pos, const int data)
{
  std::size_t new_pos;
  if (data == KEY_BACKSPACE || data == 127 || data == KEY_DC)
    new_pos = (old_pos >= pos && old_pos > 0) ? old_pos - 1 : old_pos;
  else
    new_pos = (old_pos < pos) ? old_pos : old_pos + 1;
  mstate.setCurpos(new_pos);
}
```

Листинг 3.2: Обработчик сообщений (клиент)

3.4.3 Базовые структуры сообщения

Сообщение, передаваемое в теле пакета, описывается некоторой структурой. Эти структуры и их методы описаны в листинге 3.3.

```
struct Message {
  enum class Type : std::uint8_t {
    addChar,
    getAll,
    diff,
  };

  Type type;
  std::string data;

  Message() = default;
  Message(const std::string& s);
  Message(Message::Type type, const std::string&& s);
  std::string buildPacket() const;
  template <typename Archive>
  void serialize(Archive& ar, const unsigned int version);
};

struct AddCharMessage {
  std::size_t pos;
  int data;

  template <typename Archive>
  void serialize(Archive& ar, const unsigned int version);
  AddCharMessage(const std::string& s);
  AddCharMessage(const int data, const std::size_t pos);
  Message buildMessage() const;
};

struct DiffMessage {
  std::vector<std::string> diff;

  template <typename Archive>
  void serialize(Archive& ar, const unsigned int version);
  DiffMessage(const std::string& s);
  DiffMessage(const std::vector<std::string>& diff);
```

```
37    Message buildMessage() const;
38  };
39
40  struct GetAllMessage {
41    ImmutableState imstate;
42    MutableState mstate;
43
44    template <typename Archive>
45    void serialize(Archive& ar, const unsigned int version);
46    GetAllMessage(const std::string& s);
47    GetAllMessage(const ImmutableState& imstate, const MutableState& mstate);
48    Message buildMessage() const;
49  };
50
51  template <typename Archive>
52  void Message::serialize(Archive& ar, const unsigned int version)
53  {
54    ar& type;
55    ar& data;
56  }
57
58  Message::Message(const std::string& s)
59  {
60    std::istringstream ins(s);
61    boost::archive::binary_iarchive in(ins);
62    in >> *this;
63  }
64
65  Message::Message(Message::Type type, const std::string&& s)
66  : type(type)
67  , data(std::move(s))
68  {
69  }
70
71  std::string Message::buildPacket() const
72  {
73    std::ostringstream outs;
74
75    boost::archive::binary_oarchive out(outs);
76    out << *this;
77    auto res = outs.str();
78
79    char buf[4];
80    size_to_char(res.size(), buf);
81    std::string prefix = { buf, 4 };
82    return prefix + res;
83  }
84
```

```
85  template <typename Archive>
86  void AddCharMessage::serialize(Archive& ar, const unsigned int version)
87  {
88    ar& pos;
89    ar& data;
90  }
91
92  AddCharMessage::AddCharMessage(const std::string& s)
93  {
94    std::istringstream ins(s);
95    boost::archive::binary_iarchive in(ins);
96    in >> *this;
97  }
98
99  AddCharMessage::AddCharMessage(const int data, const std::size_t pos)
100 : pos(pos)
101 , data(data)
102 {
103 }
104
105 Message AddCharMessage::buildMessage() const
106 {
107   std::ostringstream outs;
108   boost::archive::binary_oarchive out(outs);
109   out << *this;
110   return Message { Message::Type::addChar, outs.str() };
111 }
112
113 template <typename Archive>
114 void DiffMessage::serialize(Archive& ar, const unsigned int version)
115 {
116   ar& diff;
117 }
118
119 DiffMessage::DiffMessage(const std::vector<std::string>& diff)
120 : diff(diff)
121 {
122 }
123
124 DiffMessage::DiffMessage(const std::string& s)
125 {
126   std::istringstream ins(s);
127   boost::archive::binary_iarchive in(ins);
128   in >> *this;
129 }
130
131 Message DiffMessage::buildMessage() const
132 {
```

```
133   std::ostringstream outs;
134   boost::archive::binary_oarchive out(outs);
135   out << *this;
136   return Message { Message::Type::diff, outs.str() };
137 }
138
139 template <typename Archive>
140 void GetAllMessage::serialize(Archive& ar, const unsigned int version)
141 {
142   ar& imstate;
143   ar& mstate;
144 }
145
146 GetAllMessage::GetAllMessage(const std::string& s)
147 {
148   std::istringstream ins(s);
149   boost::archive::binary_iarchive in(ins);
150   in >> *this;
151 }
152
153 Message GetAllMessage::buildMessage() const
154 {
155   std::ostringstream outs;
156   boost::archive::binary_oarchive out(outs);
157   out << *this;
158   return Message { Message::Type::getAll, outs.str() };
159 }
160
161 GetAllMessage::GetAllMessage(const ImmutableState& imstate, const MutableState
      & mstate)
162 : imstate(imstate)
163 , mstate(mstate)
164 {
165 }
```

Листинг 3.3: Базовые структуры описывающие сообщение

3.4.4 Состояния системы

Система может находиться в двух различных состояниях: изменяемом и неизменяемом. В неизменяемом состоянии система находится в тот момент, когда исполняется какая-либо команда, введенная пользователем, при этом все введенные пользователем символы записываются в специальный буфер и обрабатываются после исполнения команды.

В листингах 3.4 и 3.5 представлена реализация неизменяемого и изменяемого состояния.

```
class ImmutableState {
  std::size_t startline = 0;
  std::vector<std::string> data;

  public:
  std::size_t printLines(
  const std::size_t max_lines = std::numeric_limits<std::size_t>::max(),
  const std::size_t offset = 0) const;
  void insertLine(const std::string& s);
  void insertLines(std::vector<std::string>&& vs);
  void moveUp(const std::size_t diff = 1);
  void moveDown(const std::size_t diff = 1);
  std::size_t linesToShow();
  void setLinesToShow(const std::size_t lines);

  private:
  friend class boost::serialization::access;
  template <typename Archive>
  void serialize(Archive& ar, const unsigned int version)
  {
    ar& data;
  }
};

std::size_t ImmutableState::printLines(
const std::size_t max_lines,
const std::size_t offset) const
{
  auto limit = std::min(max_lines, data.size() - startline);
  for (std::size_t i = 0; i < limit; i++) {
    move(i + offset, 0);
    clrtoeol();
    printw("%s", data[i + startline].c_str());
  }
  return limit;
}

void ImmutableState::insertLine(const std::string& s)
{
  data.push_back(s);
}

void ImmutableState::insertLines(std::vector<std::string>&& vs)
{
```

```
45    for (std::size_t i = 0; i < vs.size(); i++)
46    data.push_back(std::move(vs[i]));
47  }
48
49  void ImmutableState::moveUp(const std::size_t diff)
50  {
51    startline = diff > startline ? 0 : startline - diff;
52  }
53
54  void ImmutableState::moveDown(const std::size_t diff)
55  {
56    startline = std::min(data.size(), startline + diff);
57  }
58
59  std::size_t ImmutableState::linesToShow()
60  {
61    return data.size() - startline;
62  }
63
64  void ImmutableState::setLinesToShow(const std::size_t lines)
65  {
66    startline = data.size() > lines ? data.size() - lines : 0;
67  }
```

Листинг 3.4: Класс описывающий неизменяемое состояние системы

```
1  #define PROMPT "> "
2  #define PROMPT_SIZE 2
3
4  class MutableState {
5    std::string data;
6    std::size_t curpos = 0;
7
8    public:
9    std::size_t getCurpos() const;
10   void setCurpos(const std::size_t new_curpos);
11
12   void printLine(const std::size_t yoffset = 0) const;
13   void addChar(const char c);
14   void removeChar();
15   void moveLeft(const std::size_t diff = 1);
16   void moveRight(const std::size_t diff = 1);
17   const std::string& getText() const;
18   void clearData();
19   std::string getData();
20
21   private:
22   friend class boost::serialization::access;
23   template <typename Archive>
```

```
24    void serialize(Archive& ar, const unsigned int version)
25    {
26      ar& data;
27    }
28  };
29
30  std::size_t MutableState::getCurpos() const { return curpos; }
31
32  void MutableState::setCurpos(const std::size_t new_curpos)
33  {
34    curpos = std::min(new_curpos, data.size());
35  }
36
37  void MutableState::printLine(const std::size_t yoffset) const
38  {
39    move(yoffset, 0);
40    clrtoeol();
41    printw(PROMPT "%s", data.c_str());
42    move(yoffset, PROMPT_SIZE + curpos);
43  }
44
45  void MutableState::addChar(const char c)
46  {
47    data.insert(data.begin() + curpos++, c);
48  }
49
50  void MutableState::removeChar()
51  {
52    if (curpos > 0)
53    data.erase(--curpos, 1);
54  }
55
56  void MutableState::moveLeft(const std::size_t diff)
57  {
58    curpos = diff > curpos ? 0 : curpos - diff;
59  }
60
61  void MutableState::moveRight(const std::size_t diff)
62  {
63    curpos = std::min(data.size(), curpos + diff);
64  }
65
66  const std::string& MutableState::getText() const { return data; }
67
68  void MutableState::clearData()
69  {
70    curpos = 0;
71    data.clear();
```

```
72 }
73
74 std::string MutableState::getData()
75 {
76   curpos = 0;
77   auto old_data = std::move(data);
78   data.clear();
79   return old_data;
80 }
```

Листинг 3.5: Класс описывающий изменяемое состояние системы

3.5 Примеры работы разработанного ПО

Вывод

В данном разделе был обоснован выбор технологий для решения поставленной задачи, рассмотрены листинги реализованного программного обеспечения и приведены результаты работы ПО.

Заключение

В ходе выполнения данной работы:

— был проведен анализ существующих решений;

— был изучены протокол прикладного уровня HTTP;

— был разработан свой протокол прикладного уровня для поставленной задачи;

— было реализовано программное обеспечение (сервер и клиент) с использованием разработанного протокола для удаленного подключения к терминалу.

Таким образом, цель курсовой работы была достигнута – было разработано ПО в составе клиента и сервера для удаленного подключения к терминалу нескольких пользователей одновременно.

Литература

[1] tty-share terminal sharing [Электронный ресурс]. Режим доступа: https://tty-share.com/ (дата обращения: 08.12.2021).

[2] tmate – Instant terminal sharing [Электронный ресурс]. Режим доступа: https://tmate.io/ (дата обращения: 08.12.2021).

[3] tmux source code - GitHub [Электронный ресурс]. Режим доступа: https://github.com/tmux/tmux (дата обращения: 08.12.2021).

[4] SSH Communications Security [Электронный ресурс]. Режим доступа: https://www.ssh.com/ (дата обращения: 08.12.2021).

[5] MessagePack: It's like JSON. But fast and small. [Электронный ресурс]. Режим доступа: https://msgpack.org/index.html (дата обращения: 08.12.2021).

[6] What Is a Socket? (The Java™ Tutorials) [Электронный ресурс]. Режим доступа: https://docs.oracle.com/javase/tutorial/networking/sockets/definition.html (дата обращения: 08.12.2021).

[7] Сокеты - сетевое программирование [Электронный ресурс]. Режим доступа: https://lecturesnet.readthedocs.io/net/low-level/ipc/socket/intro.html (дата обращения: 08.12.2021).

[8] Многопоточность [Электронный ресурс]. Режим доступа: https://se.ifmo.ru/documents/10180/1422934/prog-concurrency.pdf/da25f4c8-02fc-506d-79da-7ea9a2186dac (дата обращения: 08.12.2021).

[9] Протоколы транспортного уровня [Электронный ресурс]. Режим доступа: http://opds.spbsut.ru/data/_uploaded/mu/itm_psu/vlss_present/psu-09_tcp_udp.pdf (дата обращения: 08.12.2021).

[10] Standard C++ [Электронный ресурс]. Режим доступа: https://isocpp.org/ (дата обращения: 08.12.2021).

[11] NCURSES Programming [Электронный ресурс]. Режим доступа: https://tldp.org/HOWTO/NCURSES-Programming-HOWTO/ (дата обращения: 08.12.2021).

[12] Boost C++ Libraries [Электронный ресурс]. Режим доступа: https://www.boost.org/ (дата обращения: 08.12.2021).