ТЕМА 7
Сетевое программирование

Цель лабораторной работы

Изучить способы организации сетевого взаимодействия в Java. Научиться создавать сетевые приложения на основе сокетов.

1 Основы сетевого взаимодействия

Обмен информацией в компьютерных сетях основывается на передаче между узлами сообщений, называемых пакетами, каждый из которых представляет собой набор байт и содержит служебную информацию и полезную нагрузку.

Для адресации пакетов в сетях, работающих на основе протокола IP, используется IP-адрес, который позволяет с некоторыми оговорками однозначно идентифицировать любой узел в сети. IP-адрес состоит из 4 чисел в диапазоне от 0 до 255 каждое, при записи разделяемых точкой; иными словами, IP-адрес состоит из 4 байт. К примеру, 217.21.43.2 – адрес web-сайта БГУ. Ряд диапазонов IP-адресов имеет специальное назначение, так адреса вида 127.*xx*.*xx*.*xx* используются для тестирования сетевого программного обеспечения методом обратной передачи. Среди них чаще всего используется адрес 127.0.0.1, позволяющий отправить пакет самому себе. Другим примером специализированных адресов служит диапазон 224.0.0.0-239.255.255.255, адреса из которого применяются для массовой рассылки пакетов группе узлов.

IP-адрес, вообще говоря, идентифицирует не компьютер, а интерфейс сетевой карты, поэтому узел, подключенный к двум сетям, будет иметь одновременно два различных IP-адреса. Назначение IP-адреса для узла может выполняться вручную через настройки операционной системы либо автоматически сервером динамической настройки DHCP. Для того чтобы определить IP-адрес компьютера, в MS Windows можно воспользоваться утилитой командной строки `ipconfig`, а в UNIX-системах – `ifconfig`.

Поскольку один и тот же компьютер должен иметь возможность участвовать сразу в нескольких соединениях (например, одновременно загружать страницу «ВКонтакте» и тестировать приложение из лабораторной работы №7), необходимо иметь возможность идентифицировать отдельные приложения, работающие на одном физическом узле. Для этого вводится понятие порта – конечной точки сетевого взаимодействия в пределах одного компьютера. Порт представляет собой логическую абстракцию операционной системы и системных библиотек и никак не обнаруживает

себя физически при осмотре компьютера. На любом современном компьютере доступны 65536 портов, нумеруемых начиная с 0, из которых первые 1024 считаются системными – некоторые операционные системы требуют прав администратора для работы с портами из данного диапазона.

Совокупность IP-адреса и порта с теми же оговорками, что и в случае просто IP-адреса, однозначно идентифицирует *сетевое приложение* в Интернет.

Для организации взаимодействия между приложениями источник и получатель пакетов должны договориться о способе кодирования информации, понятном обоим – такая договоренность носит название протокола. Конечно, изобретение нового протокола при каждом новом соединении неразумно, поэтому в глобальных сетях выработано и стандартизировано множество универсальных и специализированных протоколов, которые могут использоваться участниками взаимодействия.

Наибольшее распространение в глобальных сетях на сегодняшний день получил протокол маршрутизации IP (Internet Protocol), структура пакета которого приведена на рисунке 1.1. Формат протокола IP понятен большинству маршрутизаторов в локальных сетях и Интернет, которые на основе информации в заголовке (в первую очередь, по адресу получателя) принимают решение о дальнейшей судьбе пакета.

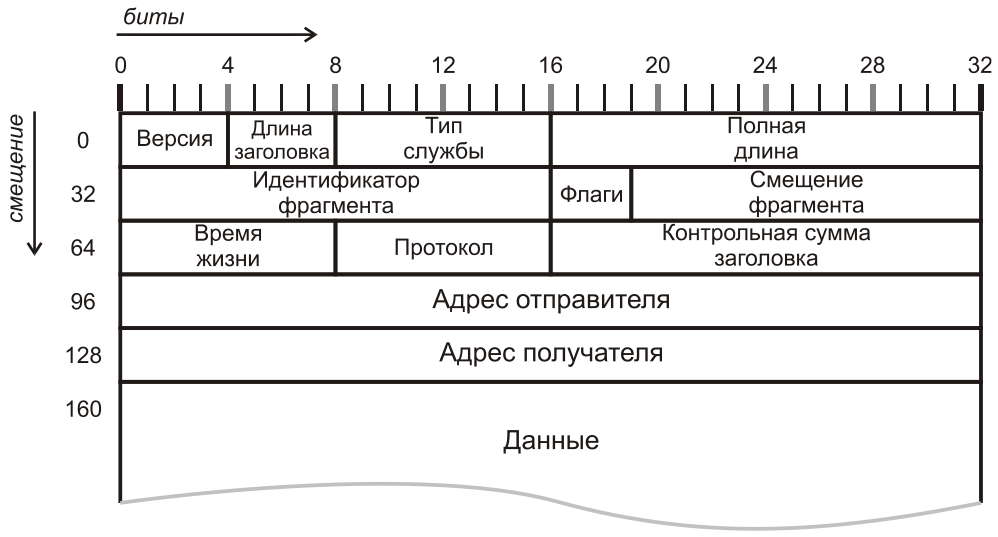

Рисунок 1.1 – Структура пакета IP

Данные, содержащиеся в пакете IP начиная со 160-ого бита, имеют двоичный формат. Обе взаимодействующие стороны должны договориться о способе кодирования и декодирования этих данных, что порождает потребность в протоколе более высокого уровня. Таких протоколов два: TCP (Transport Control Protocol) и UDP (User Datagram Protocol).

Протокол TCP используется для организации взаимодействия, ориентированного на соединения, при этом гарантируется доставка каждого из отправленных пакетов и порядок их прихода. Для гарантированной доставки пакетов протокол TCP использует механизм подтверждений: узел-получатель обязан подтвердить получение очередного пакета, отправив короткий служебный пакет-уведомление узлу-источнику – если же подтверждение не получено, через некоторый промежуток времени узел-источник повторит попытку отправки пакета. Порядок прихода гарантирован благодаря нумерации пакетов TCP: при получении пакета с неожиданным номером узел-адресат просто игнорирует такой пакет.

Протокол UDP, напротив, обходится с пакетами по принципу «отправил и забыл». При этом ни доставка конкретного пакета, ни порядок прихода пакетов не контролируется. Использование UDP полезно в таких приложениях, где потеря одиночных пакетов не играет существенной роли (приложения потокового мультимедиа, IP-телефонии, разного рода регулярные массовые уведомления).

Формат пакетов TCP и UDP мы не приводим, однако принципиально структура этих пакетов аналогична структуре пакетов IP: присутствует заголовок и полезная нагрузка (данные). Как и в случае с IP, необходимость расшифровывать записанные в пакете данные обусловливает потребность в протоколе еще более высокого уровня. Развивая эти рассуждения далее легко прийти к многоуровневой схеме организации (т.н. «стеку», от англ. stack – стопка) протоколов Интернет (рисунок 1.2).

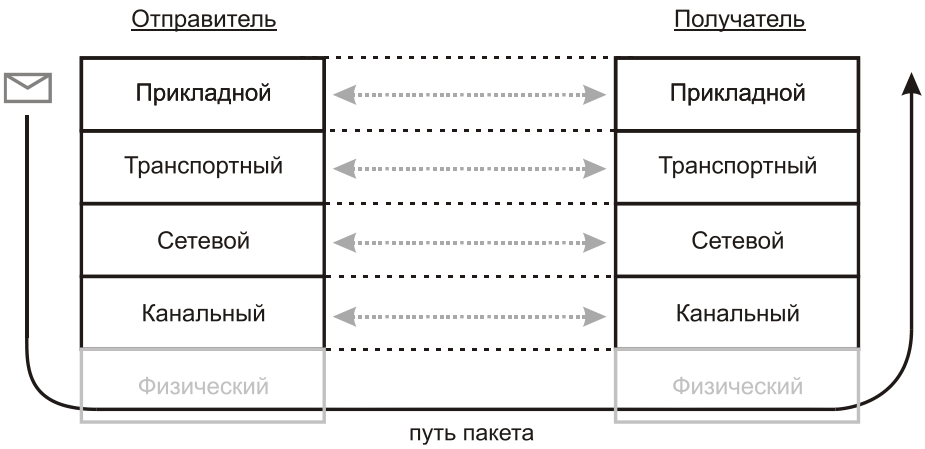

Рисунок 1.2 – Стек протоколов TCP/IP

На канальном уровне приведенной схемы в локальных сетях функционирует протокол Ethernet. Протокол IP располагается на сетевом уровне и отвечает за маршрутизацию пакетов от узла к узлу. Протоколы TCP и UDP занимают транспортный уровень, определяя способы

транспортировки информации между приложениями. Способ обработки данных приложениями задается протоколом прикладного уровня.

Для понимания стековой организации протоколов может быть полезна аналогия с работой традиционной почтовой службы. При необходимости отправить письмо своему деловому партнеру за рубежом вы вкладываете лист в конверт, адресуете его в соответствии с почтовыми требованиями и опускаете в ящик сбора почты. Почтальон извлекает конверты из ящика, сбрасывает их в мешок и транспортирует в ближайшее почтовое отделение. Почтовое отделение сортирует письма в зависимости от страны и города назначения, ориентируясь на адреса на конвертах (содержание писем не учитывается). Письма, адресованные в конкретное иностранное государство, помещаются в один контейнер и доставляются к грузовому терминалу аэропорта, где дожидаются своего рейса. По прибытии самолета в страну назначения процедура повторяется в обратном порядке. Заметим, что на каждом этапе контейнеры, способы их адресации и транспортировки различны. При этом:

- вы и ваш коллега избавлены от необходимости вникать в процедуры транспортировки сообщений, работа почтовой службы для вас проявляется только в задержке между моментом, когда вы опускаете конверт в ящик сбора писем, и моментом, когда адресат извлекает письмо из собственного почтового ящика;
- работа почтовой службы на каждом из этапов никак не зависит от содержания транспортируемых писем.

Аналогичные наблюдения относительно компьютерных сетей позволяют, рассматривая взаимодействие на одном из уровней стека протоколов, абстрагироваться от особенностей выше- и нижестоящих протоколов, полагая, что общение осуществляется горизонтально в пределах одного уровня.

1.1 Логические модели сетевых приложений

Логическая модель приложения определяет способ взаимодействия на прикладном уровне рассмотренной выше схемы. На рисунках 1.3 и 1.4 представлены две противоположные архитектуры сетевого приложения: полностью централизованная клиент-серверная модель и полностью децентрализованная одноранговая модель.

Клиент-серверная модель подразумевает наличие выделенного узла – сервера, к которому клиенты обращаются с запросами. При этом сообщения клиентов между собой не происходит.

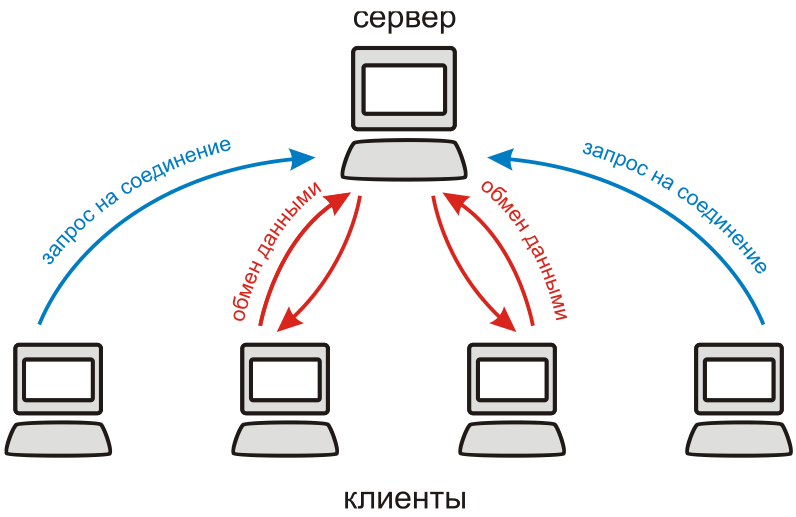

Рисунок 1.3 – Клиент-серверная модель приложения

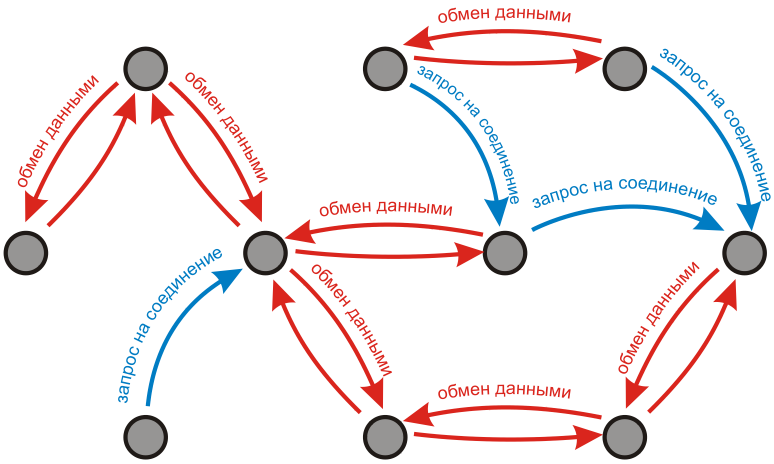

Рисунок 1.4 – Одноранговая модель приложения

Одноранговая модель, называемая также пиринговой (peer-to-peer, P2P) предполагает равноправие узлов сети. При этом каждый узел может инициировать соединение и обмениваться данными с любым другим узлом. В чистом виде одноранговая модель практически не используется – реальные пиринговые приложения (наиболее известными из которых являются различные варианты BitTorrent) в той или иной мере включают элементы централизации.

Ввиду распространенности клиент-серверной модели основные примитивы сетевого взаимодействия в Java (как и во многих других языках) изначально ориентированы на работу по этой схеме. В частности, серверный и клиентский сокеты представлены в стандартной библиотеке разными классами. Для организации одноранговового взаимодействия средствами стандартной библиотеки каждый из узлов должен выполнять функции как

клиента, так и сервера, иными словами иметь активными сокеты обоих типов.

Последовательность действий в рамках клиент-серверной модели изображена на рисунке 1.5.

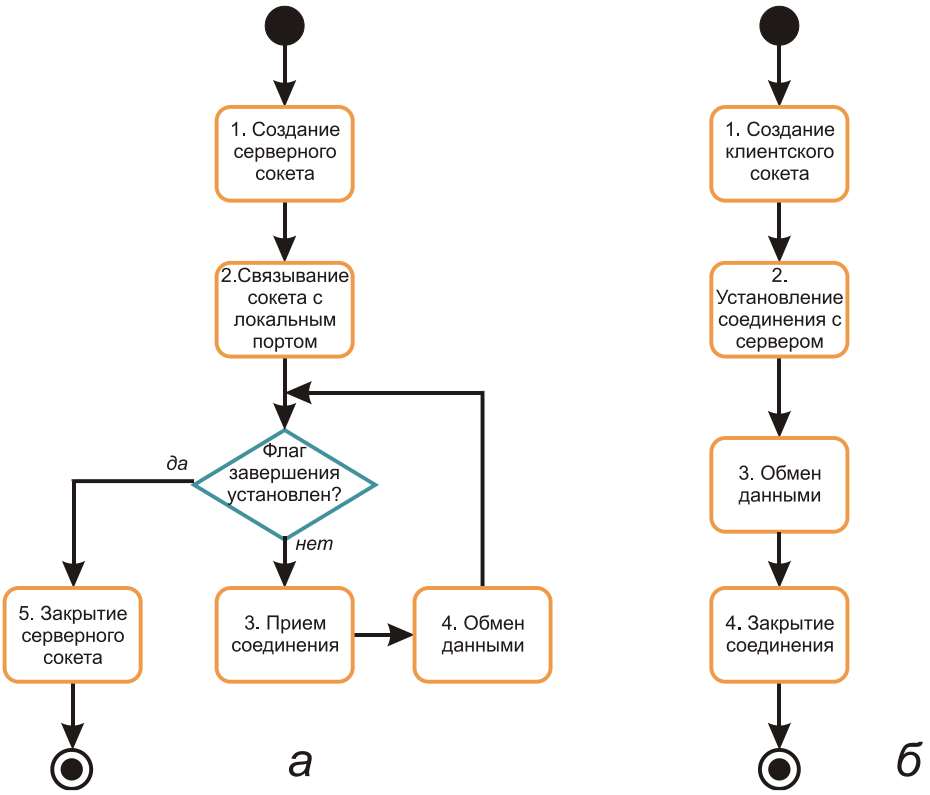

Рисунок 1.5 – Последовательность действия на стороне сервера (а) и клиента (б).

Работа на стороне сервера начинается с создания объекта серверного сокета (шаг 1). Далее серверный сокет должен быть привязан к некоторому порту (шаг 2), номер которого известен клиентам и куда клиенты будут адресовать запросы на подключение. Привязанный к определенному порту серверный сокет «прослушивает» данный порт и циклически принимает соединения от клиентов (шаг 3). В рамках клиентского подключения выполняется необходимый обмен сообщениями (шаг 4), после чего сервер принимает и обрабатывает очередное соединение. Заметим, что обмен сообщениями может осуществляться в отдельном потоке или нескольких потоках, при этом сервер имеет возможность сразу по установлении соединения с одним клиентом приступить к обслуживанию следующего, не дожидаясь завершения обмена. При выполнении некоторых условий работа сервера завершается, а серверный сокет закрывается (шаг 5).

Последовательность действий на стороне клиента более проста и включает: создание сокета (шаг 1) и установление соединения с сервером (шаг 2), обмен информацией (шаг 3) и закрытие соединения (шаг 4).

2 Краткая справка по необходимым программным компонентам

Достоинство Java в отношении организации сетевого взаимодействия состоит в том, что системные библиотеки скрывают от пользователя сложности низкоуровневой (ниже прикладного уровня) работы с сетью, предоставляя удобный прикладной интерфейс.

Основные программные абстракции для организации сетевого взаимодействия собраны в пакете `java.net`. Ключевые классы:

- `Socket` представляет клиентский сокет, ориентированный на соединение (протокол TCP);
- `ServerSocket` представляет серверный сокет, ориентированный на соединение (протокол TCP);
- `DatagramSocket` – сокет для отправки и получения дейтаграмм без установления соединений (протокол UDP);
- `MulticastSocket` – сокет для организации многоадресной рассылки без установления соединений (протокол UDP).

Вспомогательные классы:

- `InetAddress` предназначен для представления IP-адреса и содержит ряд полезных статических методов для работы с адресами;
- `InetSocketAddress` представляет адрес сокета (IP-адрес или имя узла + номер порта);
- `DatagramPacket` используется классами `DatagramSocket` и `MulticastSocket` для представления пакетов дейтаграмм.

Важнейшие методы экземпляров класса `Socket`:

Имя константы или метода	Описание

`void connect(`
`  SocketAddress endpoint)`
 	
`void connect(`
`  SocketAddress endpoint,`
`  int timeout)`
	Устанавливает соединение с удаленным сокетом, адрес которого передается через аргумент `endpoint`. Второй вариант метода задает таймаут, по истечении которого при невозможности соединения будет сгенерировано исключение `SocketTimeoutException`.

`void close()`	Разрывает установленное соединение и закрывает сокет.
`InputStream` `  getInputStream()`	Возвращает объект потока для чтения, связанного с данным сокетом.
`InputStream` `  getOutputStream()`	Возвращает объект потока для записи, связанного с данным сокетом.

`SocketAddress getRemoteSocketAddress()`	Возвращает адрес удаленного сокета, с которым установлено соединение. Этот метод полезен, когда объект `Socket` возвращен методом `accept()` серверного сокета (в это м случае о н позволяет определить адрес подключенного клиента).

Клиент-серверное взаимодействие предполагает, что метод `connect()` либо версия конструктора, устанавливающая соединение, заблокируют выполнение текущего потока до установления соединения, т.е. до того момента, когда для серверного сокета будет вызван метод `accept()`, либо до того момента, когда серверный сокет будет закрыт, не приняв клиентский запрос. Возможна также установка таймаута соединения, в этом случае по истечении заданного промежутка времени вызов `connect()` завершится генерацией исключения `SocketTimeoutException`.

Ряд вспомогательных методов `Socket` позволяют устанавливать и проверять значения служебных параметров (например, таймаут соединений), а также опрашивать состояние сокета. Информацию об этих методах можно найти в документации по API.

Конструкторы класса `Socket` позволяют установить соединение непосредственно при создании сокета:

Сигнатура конструктора	Описание
`Socket()`	Создает сокет без установления соединения. Для установления соединения должен быть отдельно вызван метод `connect()`.
`Socket(InetAddress address, int port)`	Создает сокет и устанавливает соединение с портом `port` на удаленном узле с адресом `address`.
`Socket(InetAddress address, int port, InetAddress localAddr, int localPort)`	Создает сокет, связывает его с локальным IP-адресом и портом, заданными параметрами `localAddr` и `localPort`, и устанавливает соединение с портом `port` на удаленном узле с адресом `address`.
`Socket(String host, int port)`	Создает сокет и устанавливает соединение с портом `port` на удаленном узле с именем `host`.
`Socket(String host, int port, InetAddress localAddr, int localPort)`	Создает сокет, связывает его с локальным IP-адресом и портом, заданными параметрами `localAddr` и `localPort`, и устанавливает соединение с портом `port` на удаленном узле с именем `host`.

Важнейшие методы экземпляров класса `ServerSocket`:

Имя константы или метода	Описание
`void bind( SocketAddress endpoint)`	Связывает данный сокет с заданным адресом `endpoint`. Второй вариант метода позволяет задать

`bind(SocketAddress endpoint, int backlog)`	предельный размер очереди запросов на подключение.
`Socket accept()`	Ожидает клиентского запроса на подключение, устанавливает соединение и возвращает новый сокет, позволяющий обмениваться данными с подключенным клиентом.
`void close()`	Закрывает сокет.

Дополнительные методы класса `ServerSocket` позволяют установить и проверить значение таймаута соединений, а также узнать текущее состояние сокета (не связан/связан/закрыт). Информацию об этих методах можно найти в документации по API.

Как и в случае с `Socket`, конструкторы класса `SocketServer` позволяют создать сокет, изначально связанный с определенным адресом:

Сигнатура конструктора	Описание
`ServerSocket()`	Создает серверный сокет, не привязанный к конкретному адресу. Для связывания должен быть отдельно вызван метод `bind()`.
`ServerSocket(int port)`	Создает серверный сокет, связанный с локальным портом `port`.
`ServerSocket(int port, int backlog)`	Создает серверный сокет, связанный с локальным портом `port`. Предельная длина очереди входящих запросов на подключение устанавливается равной `backlog`.
`ServerSocket(int port, int backlog, InetAddress bindAddr)`	Создает серверный сокет, связанный с портом `port` и IP-адресом `bindAddr`. Имеет смысл, если локальный компьютер снабжен более чем одним сетевым интерфейсом (подключен более чем к одной сети) и прием подключений должен выполняться только по одному из интерфейсов. Предельная длина очереди входящих запросов на подключение устанавливается равной `backlog`.

Интерес представляют также статические методы класса `InetAddress`, которые используются для работы с IP-адресами:

Имя константы или метода	Описание
`InetAddress[] getAllByName(String host)`	Возвращает набор адресов, относящихся к узлу с именем `host`, используя службу доменных имен DNS.
`InetAddress getByAddress(byte[] addr)`	Возвращает объект IP-адреса, соответствующий заданному байтовому представлению.

`InetAddress getByAddress(String host, byte[] addr)`	Возвращает объект IP-адреса, соответствующий заданному имени узла `host` и IP-адресу `addr`, без проверки соответствия через службу доменных имен DNS.
`InetAddress getByName(String host)`	Возвращает объект IP-адреса, соответствующий заданному имени узла `host`. Если `host` содержит доменное имя (например, “`java.sun.com`”), выполняется поиск соответствующего IP-адреса, используя службу доменных имен DNS. Если `host` содержит строковое представление IP-адреса (“`217.21.43.2`”) выполняется только проверка формата строки.
`InetAddress getLocalHost()`	Возвращает собственный адрес компьютера.

2.1 Пример реализации клиент-серверного приложения

Следующий пример демонстрирует реализацию эхо-сервера, возвращающего клиенту отправленное им сообщение. Эхо-сервер может использоваться для простейшего тестирования работоспособности сети.

```
public class EchoServer {
      public static final int PORT = 4488;

      public static void main(String[] args) throws IOException {
            // Шаги 1 и 2: создание серверного сокета,
            // связанного с портом PORT
            ServerSocket s = new ServerSocket(PORT);
            try {
                  while (true) {
                        // Шаг 3 - прием клиентского соединения
                        Socket socket = s.accept();
                        try {
                              // Получение объектов ввода и вывода,
                              // связанных с сокетом
                              BufferedReader in = new BufferedReader(
                                    new InputStreamReader(
                                          socket.getInputStream()));
                              PrintWriter out = new PrintWriter(
                                    new BufferedWriter(new OutputStreamWriter(
                                          socket.getOutputStream())), true);
                              // Шаг 4 - обмен данными:
                              // на каждую строку, полученную от клиента,
                              // сервер отвечает той же самой строкой;
                              // при получении строки “BYE” соединение
                              // завершается
                              while (true) {
                                    String str = in.readLine();
                                    if (str.equals("BYE"))
                                          break;
                                    out.println("ECHO: "+ str);
                              }
                        } finally {
                              // Клиентский сокет должен быть закрыт,
                              // даже если произошла ошибка
                              socket.close();
```

```
                        }
                    }
                } finally {
                    // Шаг 5 - серверный сокет должен быть закрыт,
                    // даже если произошла ошибка
                    s.close();
                }
        }
}
```

Соответствующее клиентское приложение представлено далее:

```
public class EchoClient {
        public static void main(String[] args) throws IOException {
                // Получение IP-адреса сервера на основе значения
                // из командной строки
                InetAddress addr = InetAddress.getByName(args[0]);

                // Шаги 1 и 2 – создание клиентского сокета и
                // установление соединения с сервером
                Socket socket = new Socket(addr, MySHMICQServer.PORT);
                try {
                    // Получение объектов ввода и вывода,
                    // связанных с сокетом
                    BufferedReader in = new BufferedReader(
                            new InputStreamReader(
                                    socket.getInputStream()));
                    PrintWriter out = new PrintWriter(new BufferedWriter(
                                    new OutputStreamWriter(
                                            socket.getOutputStream())), true);

                    // Получение объекта сканера для чтения строк
                    // из терминала
                    Scanner scanner = new Scanner(System.in);

                    // Шаг 3 – обмен данными: каждая строка,
                    // введенная в терминале, передается серверу,
                    // ответ выводится в терминал
                    while (scanner.hasNextLine()) {
                        out.println(scanner.nextLine());
                        String line = in.readLine();
                        if (line == null)
                            break;
                        System.out.println(line);
                    }
                } finally {
                    // Шаг 4 - сокет должен быть закрыт,
                    // даже если произошла ошибка
                    socket.close();
                }
        }
}
```

3 Пример приложения

Постановка задачи: реализовать программу обмена мгновенными сообщениями между узлами компьютерной сети. Ввод и отправка сообщений, а также отображение полученных сообщений должны выполняться средствами графического пользовательского интерфейса.

3.1 Реализация базового приложения

Разработку простейшего варианта приложения можно начать с проектирования пользовательского интерфейса. Набросок может быть выполнен на листе бумаги или в специализированном приложении, например, MS Visio. Один из вариантов внешнего вида окна представлен на рисунке 3.1.

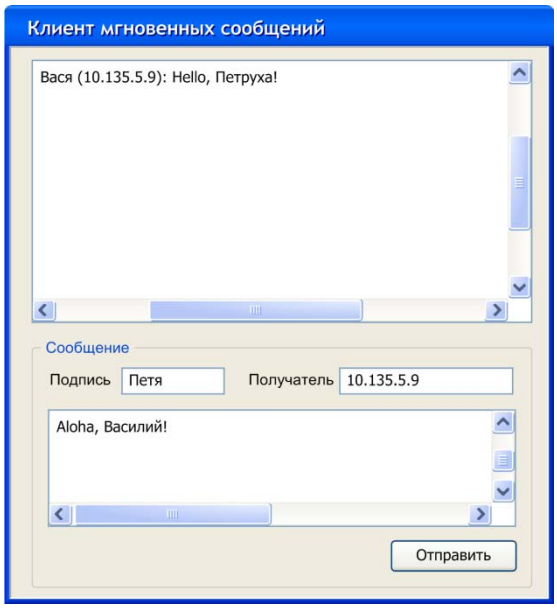

Рисунок 3.1 – Внешний вид главного окна приложения

Для размещения элементов управления на панели главного окна `MainFrame` воспользуемся диспетчером групповой компоновки (`GroupLayout`), который присутствует в библиотеке Java SE начиная с версии 6.

Менеджер `GroupLayout` в первую очередь предназначен для использования автоматизированными средствами разработки пользовательского интерфейса, такими как встроенный в NetBeans IDE редактор Matisse. Тем не менее, этот диспетчер компоновки может быть с успехом использован для ручного построение интерфейса, чем мы и займемся.

Диспетчер `GroupLayout` разделяет вертикальную и горизонтальную укладку компонентов – способ расположения компонентов в обоих направлениях задается независимо. Основной единицей компоновки является группа. Группа может содержать элементы управления, зазоры, а также другие группы. В зависимости от направления группа определяет горизонтальное или вертикальное размещение своих элементов. Способ укладки компонентов определяется типом группы:

– последовательная группа (объект класса `SequentialGroup`) располагает компоненты вдоль рассматриваемого направления (вертикального или горизонтального) последовательно один за другим, как и при использовании диспетчеров `BoxLayout` или `FlowLayout`. При этом позиция каждого следующего компонента определяется относительно границы предыдущего.
– параллельная группа (объект класса `ParallelGroup`) помещает элементы в одну и ту же область пространства вдоль текущего направления.

На рисунке 3.2 три элемента управления образуют последовательную горизонтальную группу и параллельную вертикальную группу.

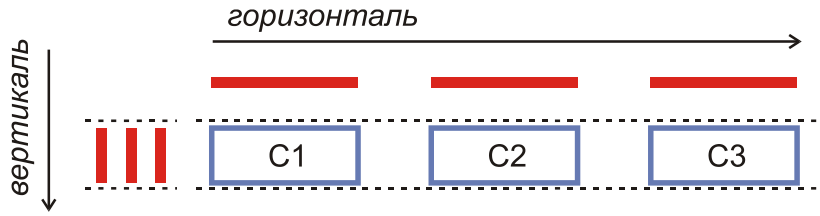

Рисунок 3.2 – Групповая раскладка

Классы `SequentialGroup` и `ParallelGroup` не имеют общедоступных конструкторов, поэтому создание объекта группы любого из типов выполняется методами класса `GroupLayout`: `createSequentialGroup()` и `createParallelGroup()`. Возвращаемое методами значение представляет ссылку на созданный объект группы требуемого типа. Подобный способ создания объектов отвечает шаблону проектирования «фабричный метод».

Метод `createParallelGroup()` принимает на вход параметр, задающий выравнивание компонентов в группе (значения приводятся для стандартного порядка следования компонент – слева направо и сверху вниз):

– `GroupLayout.Alignment.LEADING` – выравнивание по левому краю для горизонтальной группы и по верхнему краю – для вертикальной;
– `GroupLayout.Alignment.TRAILING` – выравнивание по правому краю для горизонтальной группы и по нижнему краю – для вертикальной;
– `GroupLayout.Alignment.CENTER` – выравнивание по центру;
– `GroupLayout.Alignment.BASELINE` – выравнивание по базовой линии текста – действительно только для вертикальной группы (применяется, в частности, для вертикального выравнивания подписи `JLabel` и соответствующего поля `JTextField`).

Для добавления компонентов в группу используется методы `addComponent()`, определенные в классах `SequentialGroup` и `ParallelGroup`. Методу `addComponent()` передается ссылка на добавляемый

элемент и при необходимости его желаемые размеры (минимальный, предпочтительный и максимальный).

В классе `SequentialGroup`:

```
public SequentialGroup addComponent(Component component);
public SequentialGroup addComponent(Component component, int min, int pref,
      int max);
```

В классе `ParallelGroup`:

```
public ParallelGroup addComponent(Component component);
public ParallelGroup addComponent(Component component, int min, int pref,
      int max);
```

В зависимости от заданных значений `min`, `pref` и `max` компонент может автоматически растягиваться при изменении размера окна либо сохранять неизменный размер.

Для добавления зазоров между компонентами последовательной группы используются методы:

```
public SequentialGroup addContainerGap();
public SequentialGroup addContainerGap(int pref, int max);
public SequentialGroup addGap(int size);
public SequentialGroup addGap(int min, int pref, int max);
```

Первые два метода используются для добавления зазора между границей компонента и содержащего его контейнера, а два последних – для добавления зазора между компонентами.

Легко заметить, что все перечисленные методы возвращают ссылку на исходную группу, поэтому вызовы этих методов могут быть записаны цепочкой. Это позволяет избежать определения множества промежуточных переменных. В качестве примера рассмотрим определение горизонтальной последовательной группы для компонентов на рисунке 3.2. Для удобства чтения вызов каждого из методов оформлен с новой строки:

```
// объекты layout, c1, c2 и c3 созданы ранее
// GAP_WIDTH - статическая константа
layout.createSequentialGroup()
     .addComponent(c1)
     .addGap(GAP_WIDTH)
     .addComponent(c2)
     .addGap(GAP_WIDTH)
     .addComponent(c3);
```

Ширина зазора `GAP_WIDTH` выбирается исходя из потребностей графического интерфейса. Аналогично создается параллельная вертикальная группа:

```
// объекты layout, c1, c2 и c3 созданы ранее
```

```
layout.createParallelGroup()
     .addComponent(c1)
     .addComponent(c2)
     .addComponent(c3);
```

Для связывания объекта групповой компоновки с контейнером (например, `JPanel`), компоненты которого подлежат упорядочиванию, необходимо выполнить два действия: создать объект `GroupLayout`, передав конструктору ссылку на объект контейнера, и установить созданный объект в качестве диспетчера компоновки для контейнера:

```
// объект panel создан ранее
final GroupLayout layout = new GroupLayout(panel);
panel.setLayout(layout);
```

Задание используемых диспетчером горизонтальной и вертикальной групп выполняется методами `setHorizontalGroup()` и `setVerticalGroup()` объекта `layout`. Полностью пример организации компоновки для элементов на рисунке 3.2 выглядит следующим образом:

```
final GroupLayout layout = new GroupLayout(panel);
panel.setLayout(layout);

layout.setHorizontalGroup(
      layout.createSequentialGroup()
            .addComponent(c1)
            .addGap(GAP_WIDTH)
            .addComponent(c2)
            .addGap(GAP_WIDTH)
            .addComponent(c3));
layout.setVerticalGroup(
      layout.createParallelGroup()
            .addComponent(c1)
            .addComponent(c2)
            .addComponent(c3));
```

Гибкость групповой компоновки достигается благодаря возможностям вложения групп друг в друга. Рассмотрим, каким образом групповая компоновка может быть применена для организации графического интерфейса, представленного на рисунке 3.1. Создание компонентов пользовательского интерфейса и их размещение на панели содержимого будем производить в конструкторе класса `MainFrame`.

Вполне естественным будет выделить на главной панели окна два крупных компонента: текстовую область вывода полученных сообщений и панель отправки «Сообщение», заключенную в рамку.

Для представления текстовой области используется объект класса `JTextArea` библиотеки Swing, который помещается в панель прокрутки `JScrollPane`:

```
// Текстовая область для отображения полученных сообщений
```

```
textAreaIncoming = new JTextArea(INCOMING_AREA_DEFAULT_ROWS, 0);
textAreaIncoming.setEditable(false);

// Контейнер, обеспечивающий прокрутку текстовой области
final JScrollPane scrollPaneIncoming = new JScrollPane(textAreaIncoming);
```

Создание панели «Сообщение» и добавление рамки производится известным по лабораторной работе №2 способом:

```
// Панель ввода сообщения
final JPanel messagePanel = new JPanel();
messagePanel.setBorder(BorderFactory.createTitledBorder("Сообщение"));
```

Текстовая область и панель «Сообщение» образуют параллельную горизонтальную группу, т.к. занимают по горизонтали одну и ту же область, и последовательную вертикальную группу. Используя описанные выше классы и методы это можно выразить следующим образом:

```
// Компоновка элементов фрейма
final GroupLayout layout1 = new GroupLayout(getContentPane());
setLayout(layout1);

layout1.setHorizontalGroup(
      layout1.createSequentialGroup()
            .addContainerGap()
            .addGroup(
                  layout1.createParallelGroup()
                        .addComponent(scrollPaneIncoming)
                        .addComponent(messagePanel))
            .addContainerGap());
layout1.setVerticalGroup(
      layout1.createSequentialGroup()
            .addContainerGap()
            .addComponent(scrollPaneIncoming)
            .addGap(MEDIUM_GAP)
            .addComponent(messagePanel)
            .addContainerGap());
```

Для добавления зазоров в горизонтальном направлении параллельная группа, содержащая элементы `scrollPaneIncoming` и `messagePanel`, дополнительно облекается в последовательную группу с двумя контейнерными зазорами.

Элементы управления, представленные на панели «Сообщение», создаются привычным образом:

```
// Подписи полей
final JLabel labelFrom = new JLabel("Подпись");
final JLabel labelTo = new JLabel("Получатель");

// Поля ввода имени пользователя и IP-адреса получателя
textFieldFrom = new JTextField(FROM_FIELD_DEFAULT_COLUMNS);
textFieldTo = new JTextField(TO_FIELD_DEFAULT_COLUMNS);

// Текстовая область для ввода сообщения
textAreaOutgoing = new JTextArea(OUTGOING_AREA_DEFAULT_ROWS, 0);
```

```
// Контейнер, обеспечивающий прокрутку текстовой области
final JScrollPane scrollPaneOutgoing = new JScrollPane(textAreaOutgoing);

// Кнопка отправки сообщения
final JButton sendButton = new JButton("Отправить");
```

Панель «Сообщение» может быть разделена на три области (рисунок 3.3), образующие параллельную горизонтальную группу и последовательную вертикальную. Выравнивание в параллельной горизонтальной группе осуществляется по правому краю, для того чтобы обеспечить положение кнопки «Отправить».

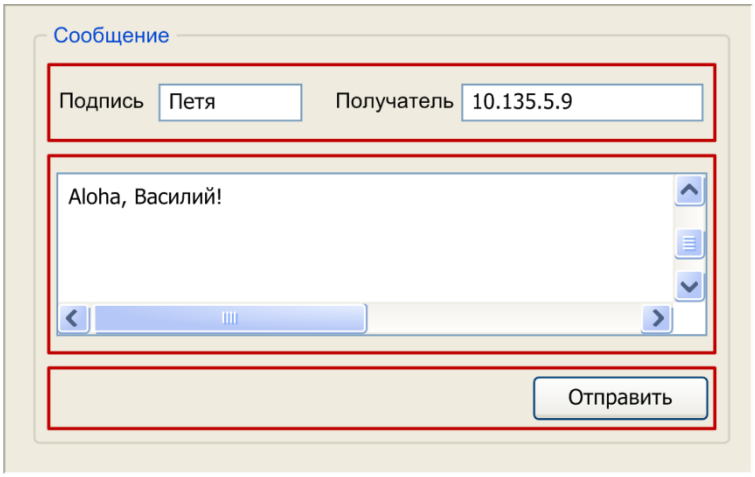

Рисунок 3.3 – Группы элементов на панели «Сообщение»

Следующий фрагмент определяет указанное расположение (создание группы для первой области пока опустим):

```
// Компоновка элементов панели "Сообщение"
final GroupLayout layout2 = new GroupLayout(messagePanel);
messagePanel.setLayout(layout2);

layout2.setHorizontalGroup(
      layout2.createSequentialGroup()
            .addContainerGap()
            .addGroup(
                  layout2.createParallelGroup(GroupLayout.Alignment.TRAILING)
                        .addGroup(/* Горизонтальная группа полей и меток */))
                        .addComponent(scrollPaneOutgoing)
                        .addComponent(sendButton))
            .addContainerGap());
layout1.setVerticalGroup(
      layout1.createSequentialGroup()
            .addContainerGap()
            .addGroup(/* Вертикальная группа полей и меток */)
            .addGap(MEDIUM_GAP)
            .addComponent(scrollPaneOutgoing)
            .addGap(MEDIUM_GAP)
            .addComponent(sendButton)
            .addContainerGap());
```

Две метки и два поля в верхней части панели «Сообщение» составляют последовательную горизонтальную группу и параллельную вертикальную группу с выравниванием по базовой линии. Размеры зазоров фиксированные. Горизонтальное расположение указанных компонентов определяется следующим образом:

```
layout1.createSequentialGroup()
      .addComponent(labelFrom)
      .addGap(SMALL_GAP)
      .addComponent(textFieldFrom)
      .addGap(LARGE_GAP)
      .addComponent(labelTo)
      .addGap(SMALL_GAP)
      .addComponent(textFieldTo)
```

а вертикальное расположение, в свою очередь:

```
layout1.createParallelGroup(GroupLayout.Alignment.BASELINE)
      .addComponent(labelFrom)
      .addComponent(textFieldFrom)
      .addComponent(labelTo)
      .addComponent(textFieldTo)
```

Эти фрагменты должны быть вставлены на место комментариев в предыдущем листинге.

Заметим, что при использовании групповой компоновки явно добавлять элементы на панель посредством метода `add()` не нужно.

Запуск приложения и отображение главного окна будем выполнять из метода `main()`:

```
public static void main(String[] args) {
      SwingUtilities.invokeLater(new Runnable() {

            @Override
            public void run() {
                  final MainFrame frame = new MainFrame();
                  frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                  frame.setVisible(true);
            }
      });
}
```

Статический метод `invokeLater()` класса `SwingUtilities` помещает переданный ему объект задания в очередь на выполнение потоком обработки пользовательских событий библиотеки Swing.

Получившееся в результате приложение отображает графическое окно, но не выполняет никаких полезных функций. Для добавления сетевой функциональности необходимо реализовать процедуры отправки и получения сообщений. За отправку сообщения будет отвечать метод `sendMessage()`, который мы добавим в класс `MainFrame`. Вызов метода

`sendMessage()` является единственным действием в обработчике нажатия на кнопку «Отправить»:

```
sendButton.addActionListener(new ActionListener() {

	@Override
	public void actionPerformed(ActionEvent e) {
		sendMessage();
	}
});
```

Последовательность работы метода `sendMessage()` такова:

1. считать значения полей `textFieldFrom`, `textFieldTo` и `textAreaOutgoing` и убедиться, что они не пустые;
2. установить сетевое соединение с узлом, адрес которого прочитан из поля `textFieldTo`;
3. получить поток для записи данных;
4. записать в поток собственное имя (содержимое поля `textFieldFrom`) и сообщение (содержимое области `textAreaOutgoing`);
5. разорвать соединение;
6. очистить содержимое области `textAreaOutgoing`.

При обнаружении ошибок метод `sendMessage()` должен оповестить пользователя. Полный текст метода можно найти в приложении 1.

Получение сообщений необходимо организовать в отдельном потоке, поскольку данная процедура включает вызов блокирующих методов. В случае если блокирующий метод будет вызван в потоке обработки событий пользовательского интерфейса, графическое окно на время блокировки станет невосприимчиво к действиям пользователя («зависнет»). Запуск нового потока будем выполнять в конструкторе после создания и компоновки всех элементов управления:

```
// Создание и запуск потока-обработчика запросов
new Thread(new Runnable() {

	@Override
	public void run() {
		/* Процедура приема сообщений */
	}
}).start();
```

Процедура приема сообщения включает:

1. создание серверного сокета;
2. запуск цикла приема сообщений:
   2.1. принять очередное соединение;
   2.2. получить поток для чтения данных;

2.3. считать имя отправителя и сообщение;
2.4. разорвать соединение;
2.5. сформировать строку, включающую имя пользователя, его адрес и полученное сообщение, и вывести ее на панель отображения полученных сообщений `textAreaIncoming`;
2.6. перейти к шагу 2.1.

Полный текст метода `run()` можно найти в приложении 1.

Существенными функциональными недостатками разработанного приложения являются:

- адресация исходящих сообщений на основе трудно запоминаемых IP-адресов;
- отсутствие возможности убедиться в достоверности имени отправителя.

Некоторые способы решения этих проблем рассматриваются в заданиях варианта С.

3.2 Рефакторинг приложения

Для обеспечения возможности дальнейшего расширения функциональности приложения полезно выполнить ряд структурных модификаций, не затрагивающих функциональные возможности и внешний вид разработанного приложения. Процедура модификации подобного рода получила название *рефакторинга* приложения.

Главным структурным недостатком приложения в том виде, в котором оно было реализовано в предыдущем разделе, является совмещение в одном классе задач несвязанного характера: отображение графического пользовательского интерфейса и поддержка сетевого взаимодействия. Для дальнейшего развития приложения эти аспекты должны быть разделены.

Для этого выделим некоторые элементы класса `MainFrame` в новый класс `InstantMessenger` (в классе `MainFrame` при этом определим поле типа `InstantMessenger`). Выделению подлежат, в первую очередь, фрагменты метода `sendMessage()` и конструктора, непосредственно отвечающие за сетевое взаимодействие. В результате в классе `InstantMessenger` появятся методы `sendMessage()` и `startServer()`:

```
public void sendMessage(String senderName, String destinationAddress,
      String message) throws UnknownHostException, IOException;

private void startServer();
```

Метод `sendMessage()` класса `InstantMessenger` является общедоступным и вызывается клиентами данного класса (в частности,

MainFrame), а метод startServer() закрыт и вызывается только из конструктора InstantMessenger.

Поскольку в существующих системах обмена сообщениями, как правило, исходящие сообщения подписываются одним и тем же именем (т.е. взаимодействие происходит от лица одной и той же учетной записи), введем в классе InstantMessenger поле sender строкового типа, обеспечив для него методы доступа и модификации.

Так как прием сообщений теперь выполняет объект класса InstantMessenger, необходимо обеспечить уведомление основного окна приложения о получении сообщения. Традиционным механизмом для такого оповещения является механизм обратных вызовов, реализуемый в Java на основе шаблона проектирования «наблюдатель» (observer). Механизм обратного вызова широко используется при разработке графических приложений на основе библиотеки Swing, где ему следуют обработчики событий нажатия кнопок ActionListener, мыши MouseListener и MouseMotionListener, а также событий окна WindowListener и другие.

Для начала определим интерфейс MessageListener, содержащий единственный метод messageReceived():

```
void messageReceived(String senderName, String message);
```

В классе InstantMessenger организуем список слушателей (поле listeners) и предоставим общедоступные методы для регистрации и удаления слушателей (наподобие того, как это делается с обработчиками нажатия на кнопку JButton):

```
public void addMessageListener(MessageListener listener) {
	synchronized (listeners) {
		listeners.add(listener);
	}
}

public void removeMessageListener(MessageListener listener) {
	synchronized (listeners) {
		listeners.remove(listener);
	}
}
```

Для выполнения оповещения реализуем метод notifyListeners(), в задачу которого входит вызов метода messageReceived() для каждого из зарегистрированных обработчиков:

```
private void notifyListeners(Peer sender, String message) {
	synchronized (listeners) {
		for (MessageListener listener : listeners) {
			listener.messageReceived(sender, message);
```

```
            }
        }
}
```

Синхронизация в данном случае необходима, поскольку добавление обработчиков и оповещение могут выполняться в различных потоках. Отсутствие блокировки в методе `notifyListeners()` чревато генерацией исключения `ConcurrentModificationException`.

В заключение целесообразно сгруппировать информацию, относящуюся к отдельному удаленному собеседнику (его имя и адрес), в отдельном классе `Peer` и модифицировать сигнатуры методов `messageReceived()` и `sendMessage()` для приема аргументов этого типа. Это позволит в дальнейшем расширять набор атрибутов, характеризующих собеседника, без модификации заголовков всех методов, которым требуется подобная информация. Так, например, тип параметра `destinationAddress` стоит заменить специализированным классом `InetSocketAddress`, включающим как IP-адрес, так и порт.

4 Задания

4.1 Вариант A

Модифицировать (по вариантам) приложение мгновенного обмена сообщениями.

№ п/п	Цель модификаций
1	Добавить проверку корректности ввода IP-адреса получателя. При попытке ввода некорректного значения принудительно возвращать курсор в поле ввода.
2	Добавить проверку корректности ввода IP-адреса получателя. Проверку выполнять при нажатии кнопки «Отправить», выводя в случае ошибки соответствующее сообщение.
3	Обеспечить возможность задания порта, используемого для приема сообщений. При этом необходимо реализовать считывание адреса из поля получатель, заданного в формате «*адрес*:*порт*».
4	Добавить переключатель, управляющий активностью клиента мгновенных сообщений.
5	Выполнять отправку сообщения при нажатии комбинации клавиш Ctrl-Enter, когда курсор находится в поле ввода сообщения. При выводе сообщения в текстовую область указывать время получения сообщения.

4.2 Вариант B

1. Модифицировать приложении мгновенного обмена сообщениями в соответствии с пунктом 3.2.

2. Реализовать дополнительные возможности (по вариантам):

№ п/п	Цель модификаций
1	Форматирование текста в сообщениях (полужирное и курсивное начертание, а также подчеркивание символов). Воспользоваться языком разметки HTML и компонентом отображения JEditorPane или JTextPane.
2	При отображении преобразовывать в сообщении некоторые последовательности символов (такие как “:)”, “8-)”) в соответствующие иконки. Воспользоваться языком разметки HTML и компонентом отображения JEditorPane или JTextPane.
3	В текстовой области отображать имя отправителя в виде гиперссылки, при нажатии на которую соответствующий IP-адрес должен помещаться в текстовое поле.
4	Добавить возможность пересылки в составе сообщения файлов произвольного типа.
5	Для диалога с каждым новым собеседником открывать новое окно.
6	Для диалога с каждым новым собеседником создавать новую вкладку (использовать компонент JTabbedPane).
7	Использовать функции класса ExecutorService для организации пула потоков, обрабатывающих соединения.

4.3 Вариант C

Модифицировать (по вариантам) приложение мгновенного обмена сообщениям.

№ п/п	Цель модификаций
1	Реализовать клиент-серверную модель системы обмена сообщениями, где центральный сервер используется для аутентификации пользователей (сличая переданный пользователем пароль с хранящимся в базе данных сервера) и передачи сообщений от пользователя к пользователю. В этом случае пользователи не знают сетевых адресов друг друга, сообщения посылаются центральному серверу, которых переправляет их адресатам. Добавить в приложение обмена сообщениями возможность поиска пользователей. Для диалога с отдельным собеседником открывать новое окно, либо создавать новую вкладку.
2	Реализовать одноранговую (пиринговую) модель системы обмена сообщениями, где сообщения между участниками передаются непосредственно. Для оповещения остальных участников о своем присутствии в сети каждый узел должен периодически выполнять рассылку уведомлений (применить сокеты для многоадресной рассылки). Предложить вариант аутентификации пользователей в подобной модели. Добавить в приложение обмена сообщениями окно списка контактов. Для диалога с отдельным собеседником открывать новое окно, либо создавать новую вкладку.

Приложение 1. Исходный код приложения

Класс главного окна приложения MainFrame

```
package bsu.rfe.java.teacher.lab7.varA1;

import java.awt.Dimension;
import java.awt.Toolkit;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;
import java.net.InetSocketAddress;
import java.net.ServerSocket;
import java.net.Socket;
import java.net.UnknownHostException;

import javax.swing.BorderFactory;
import javax.swing.GroupLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.JTextField;
import javax.swing.SwingUtilities;

@SuppressWarnings("serial")
public class MainFrame extends JFrame {

	private static final String FRAME_TITLE = "Клиент мгновенных сообщений";

	private static final int FRAME_MINIMUM_WIDTH = 500;
	private static final int FRAME_MINIMUM_HEIGHT = 500;

	private static final int FROM_FIELD_DEFAULT_COLUMNS = 10;
	private static final int TO_FIELD_DEFAULT_COLUMNS = 20;

	private static final int INCOMING_AREA_DEFAULT_ROWS = 10;
	private static final int OUTGOING_AREA_DEFAULT_ROWS = 5;

	private static final int SMALL_GAP = 5;
	private static final int MEDIUM_GAP = 10;
	private static final int LARGE_GAP = 15;

	private static final int SERVER_PORT = 4567;

	private final JTextField textFieldFrom;
	private final JTextField textFieldTo;

	private final JTextArea textAreaIncoming;
	private final JTextArea textAreaOutgoing;

	public MainFrame() {
		super(FRAME_TITLE);
		setMinimumSize(
			new Dimension(FRAME_MINIMUM_WIDTH, FRAME_MINIMUM_HEIGHT));
```

```
// Центрирование окна
final Toolkit kit = Toolkit.getDefaultToolkit();
setLocation((kit.getScreenSize().width - getWidth()) / 2,
            (kit.getScreenSize().height - getHeight()) / 2);

// Текстовая область для отображения полученных сообщений
textAreaIncoming = new JTextArea(INCOMING_AREA_DEFAULT_ROWS, 0);
textAreaIncoming.setEditable(false);

// Контейнер, обеспечивающий прокрутку текстовой области
final JScrollPane scrollPaneIncoming =
      new JScrollPane(textAreaIncoming);

// Подписи полей
final JLabel labelFrom = new JLabel("Подпись");
final JLabel labelTo = new JLabel("Получатель");

// Поля ввода имени пользователя и адреса получателя
textFieldFrom = new JTextField(FROM_FIELD_DEFAULT_COLUMNS);
textFieldTo = new JTextField(TO_FIELD_DEFAULT_COLUMNS);

// Текстовая область для ввода сообщения
textAreaOutgoing = new JTextArea(OUTGOING_AREA_DEFAULT_ROWS, 0);

// Контейнер, обеспечивающий прокрутку текстовой области
final JScrollPane scrollPaneOutgoing =
      new JScrollPane(textAreaOutgoing);

// Панель ввода сообщения
final JPanel messagePanel = new JPanel();
messagePanel.setBorder(
      BorderFactory.createTitledBorder("Сообщение"));

// Кнопка отправки сообщения
final JButton sendButton = new JButton("Отправить");
sendButton.addActionListener(new ActionListener() {

      @Override
      public void actionPerformed(ActionEvent e) {
            sendMessage();
      }
});

// Компоновка элементов панели "Сообщение"
final GroupLayout layout2 = new GroupLayout(messagePanel);
messagePanel.setLayout(layout2);

layout2.setHorizontalGroup(layout2.createSequentialGroup()
      .addContainerGap()
      .addGroup(layout2.createParallelGroup(Alignment.TRAILING)
                  .addGroup(layout2.createSequentialGroup()
                              .addComponent(labelFrom)
                              .addGap(SMALL_GAP)
                              .addComponent(textFieldFrom)
                              .addGap(LARGE_GAP)
                              .addComponent(labelTo)
                              .addGap(SMALL_GAP)
                              .addComponent(textFieldTo))
                  .addComponent(scrollPaneOutgoing)
                  .addComponent(sendButton))
      .addContainerGap());
layout2.setVerticalGroup(layout2.createSequentialGroup()
      .addContainerGap()
```

```
                .addGroup(layout2.createParallelGroup(Alignment.BASELINE)
                        .addComponent(labelFrom)
                        .addComponent(textFieldFrom)
                        .addComponent(labelTo)
                        .addComponent(textFieldTo))
                .addGap(MEDIUM_GAP)
                .addComponent(scrollPaneOutgoing)
                .addGap(MEDIUM_GAP)
                .addComponent(sendButton)
                .addContainerGap());

        // Компоновка элементов фрейма
        final GroupLayout layout1 = new GroupLayout(getContentPane());
        setLayout(layout1);

        layout1.setHorizontalGroup(layout1.createSequentialGroup()
                    .addContainerGap()
                    .addGroup(layout1.createParallelGroup()
                            .addComponent(scrollPaneIncoming)
                            .addComponent(messagePanel))
                    .addContainerGap());
        layout1.setVerticalGroup(layout1.createSequentialGroup()
                    .addContainerGap()
                    .addComponent(scrollPaneIncoming)
                    .addGap(MEDIUM_GAP)
                    .addComponent(messagePanel)
                    .addContainerGap());

        // Создание и запуск потока-обработчика запросов
        new Thread(new Runnable() {

                @Override
                public void run() {
                    try {

                        final ServerSocket serverSocket =
                                new ServerSocket(SERVER_PORT);
                        while (!Thread.interrupted()) {
                            final Socket socket = serverSocket.accept();
                            final DataInputStream in = new DataInputStream(
                                    socket.getInputStream());

                            // Читаем имя отправителя
                            final String senderName = in.readUTF();

                            // Читаем сообщение
                            final String message = in.readUTF();

                            // Закрываем соединение
                            socket.close();

                            // Выделяем IP-адрес
                            final String address =
                                    ((InetSocketAddress) socket
                                            .getRemoteSocketAddress())
                                                    .getAddress()
                                                            .getHostAddress();

                            // Выводим сообщение в текстовую область
                            textAreaIncoming.append(senderName +
                                    " (" + address + "): " +
                                    message + "\n");
                        }
```

```
                        } catch (IOException e) {
                              e.printStackTrace();
                              JOptionPane.showMessageDialog(MainFrame.this,
                                          "Ошибка в работе сервера", "Ошибка",
                                          JOptionPane.ERROR_MESSAGE);
                        }
                  }
            }).start();
      }

      private void sendMessage() {
            try {
                  // Получаем необходимые параметры
                  final String senderName = textFieldFrom.getText();
                  final String destinationAddress = textFieldTo.getText();
                  final String message = textAreaOutgoing.getText();

                  // Убеждаемся, что поля не пустые
                  if (senderName.isEmpty()) {
                        JOptionPane.showMessageDialog(this,
                                    "Введите имя отправителя", "Ошибка",
                                    JOptionPane.ERROR_MESSAGE);
                        return;
                  }
                  if (destinationAddress.isEmpty()) {
                        JOptionPane.showMessageDialog(this,
                                    "Введите адрес узла-получателя", "Ошибка",
                                    JOptionPane.ERROR_MESSAGE);
                        return;
                  }
                  if (message.isEmpty()) {
                        JOptionPane.showMessageDialog(this,
                                    "Введите текст сообщения", "Ошибка",
                                    JOptionPane.ERROR_MESSAGE);
                        return;
                  }

                  // Создаем сокет для соединения
                  final Socket socket =
                        new Socket(destinationAddress, SERVER_PORT);

                  // Открываем поток вывода данных
                  final DataOutputStream out =
                        new DataOutputStream(socket.getOutputStream());

                  // Записываем в поток имя
                  out.writeUTF(senderName);

                  // Записываем в поток сообщение
                  out.writeUTF(message);

                  // Закрываем сокет
                  socket.close();

                  // Помещаем сообщения в текстовую область вывода
                  textAreaIncoming.append("Я -> " + destinationAddress + ": "
                              + message + "\n");

                  // Очищаем текстовую область ввода сообщения
                  textAreaOutgoing.setText("");
            } catch (UnknownHostException e) {
                  e.printStackTrace();
                  JOptionPane.showMessageDialog(MainFrame.this,
                  "Не удалось отправить сообщение: узел-адресат не найден",
```

```
                        "Ошибка", JOptionPane.ERROR_MESSAGE);
            } catch (IOException e) {
                  e.printStackTrace();
                  JOptionPane.showMessageDialog(MainFrame.this,
                              "Не удалось отправить сообщение", "Ошибка",
                              JOptionPane.ERROR_MESSAGE);
            }
      }

      public static void main(String[] args) {
            SwingUtilities.invokeLater(new Runnable() {

                  @Override
                  public void run() {
                        final MainFrame frame = new MainFrame();
                        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                        frame.setVisible(true);
                  }
            });
      }
```