Оглавление

Введение

Системы обмена сообщениями устойчиво закрепились в жизни человека. Они используются не только для общения с друзьями или близкими людьми, но и для общения по работе или учебе. Особенностью нынешних систем обмена текстовыми сообщениями является возможность создавать группы, в которых могут общаться сразу несколько человек. Данная особенность избавляет пользователя от необходимости дублировать информацию нескольким людям. Кроме того в группах можно делиться файлами с участниками групп, что также упрощает взаимодействие, избавляя человека от необходимости создания электронного письма с вложением или загрузки файла на удаленный сервер.

Цель работы – разработать программный комплекс, реализующий обмен текстовыми сообщениями между пользователями в режиме реального времени.

Чтобы достигнуть поставленной цели, требуется решить следующие задачи:

- провести анализ существующих решений;
- проанализировать протоколы обмена сообщениями;
- реализовать в программном комплексе протокол обмена сообщениями;
- реализовать программное обеспечение для обмена сообщениями в режиме реального времени.

1 Аналитическая часть

В данном разделе представено описание существующих протоколов обмена сообщениями, анализ существующих решений и выбор протокола для реализации в работе.

1.1 Постановка задачи

В соответствии с заданием необходимо разработать программный комплекс, реализующий обмен текстовыми сообщениями между пользователями в режиме реального времени. Одновременно к приложению может быть подключено несколько пользователей. Предусмотреть возможность создавать комнаты для общения, подключаться к уже существующим комнатам по приглашению администратора (или администраторов) комнаты. При подключении к комнате не показывать новому пользователю историю сообщений. Предоставить пользователям возможность отправлять файлы. Реализовать консольный интерфейс для приложения. Для решения этой задачи необходимо изучить предментную область и проанализировать существующие решения.

1.2 Протоколы обмена мгновенными сообщениями

Система мгновенного обмена сообщениями [1] (англ. Instant messaging, IM) — система для обмена сообщениями в реальном времени через Интернет. В таких системах могут передаваться текстовые сообщения, звуковые сигналы, изображения, видео, а также производиться такие действия, как совместное рисование или игры. Многие из таких программ-клиентов могут применяться для организации групповых текстовых чатов или видеоконференций.

Системы мгновенного обмена сообщениями используют соответствующие протоколы или их модификации.

В данном подразделе буду рассмотрены открытые протоколы мгновенного обмена сообщениями.

1.2.1 Distributed Data Protocol

Distributed Data Protocol или DDP [2] (протокол распределенных данных) – протокол клиент-серверного взаимодействия, созданный для использования инфраструктурой JavaScript веб-платформы Meteor и использующий в качестве обмена сообщениями шаблон издатель-подписчик [3].

Стандартным способом передачи данных через DDP является передача EJSON через веб-сокеты. Вторым вариантом является использование Long Poll.

В случае, если браузер не поддерживает веб-сокеты, передача данных будет осуществляться с использованием Long Poll.

1.2.2 IRC

IRC [4] (англ. Internet Relay Chat) – протокол прикладного уровня для обмена сообщениями в режиме реального времени. Разработан в основном для группового общения, также позволяет общаться через личные сообщения и обмениваться данными, в том числе файлами.

IRC использует транспортный протокол TCP и криптографический TLS (опционально).

IRC предоставляет возможность как группового, так и приватного общения. Для группового общения существует несколько возможностей. Пользователь может отправить сообщение списку пользователей, при этом серверу отправляется список, сервер выделяет из него отдельных пользователей и отправляет копию сообщения каждому из них.

Более эффективным является использование каналов. В этом случае сообщение отправляется непосредственно серверу, а сервер отправляет его всем пользователям в канале.

Как при групповом, так и при приватном общении сообщения отправляются клиентам по кратчайшему пути и видимы только отправителю,

получателю и входящим в кратчайший путь серверам.

Кроме того, возможна отправка широковещательного сообщения. Сообщения клиентов, касающиеся изменения состояния сети (например, режима канала или статуса пользователя), должны отправляться всем серверам, входящим в сеть. Все сообщения, исходящие от сервера, также должны быть отправлены всем остальным серверам.

1.2.3 Matrix

Matrix [5] – открытый протокол мгновенного обмена сообщениями и файлами с поддержкой голосовой и видеосвязи. Это децентрализованный клиент-серверный протокол с передачей сообщений между серверами.

Протокол Matrix позиционирован создателями как замена для более ранних протоколов, он призван объединить мгновенные сообщения с голосовым и видео-общением, что не удалось сделать в рамках SIP, XMPP и RCS.

Ключевые особенности протокола Matrix – объединение в одном месте всех каналов непосредственного общения и децентрализация.

Концепция Matrix основана на принципах построения электронной почты. Внутренняя организация протокола похожа на IRC – доверенные серверы обмениваются сообщениями чатов друг с другом. При этом Matrix отличается от того же IRC низким порогом вхождения, для общения через Matrix не нужно быть опытным пользователем, идентификация проста и осуществляется по номеру телефона, адресу электронной почты, аккаунтам Facebook или Google или другим способом, привычным пользователю.

1.2.4 XMPP

XMPP [6] (англ. eXtensible Messaging and Presence Protocol «расширяемый протокол обмена сообщениями и информацией о присутствии») – открытый, основанный на XML, свободный для использования протокол для мгновенного обмена сообщениями и информацией о присутствии в режиме, близком к режиму реального времени. Изначально спроектированный лег-

ко расширяемым, протокол, помимо передачи текстовых сообщений, поддерживает передачу голоса, видео и файлов по сети.

Расширяемость протокола предназначена для добавления в единую коммуникационную сеть мессенджеров, социальных сетей, сайтов, использующих разные, несовместимые стандарты. Предполагалось, что крупные компании будут открывать межсерверное общение с другими IM и описывать свои методы шифрования, передачи мультимедиа и других данных через публикацию расширений XMPP. Расширения будут приниматься или отклоняться глобальным сообществом путём наибольшего распространения, но при этом всегда будет доступна базовая функциональность для передачи сообщений для пользователей разных мессенджеров. В реальности данная идея не получила должного распространения, и большинство крупных компаний не стало открывать возможность коммуникации для своих пользователей с другими сервисами.

В отличие от коммерческих систем мгновенного обмена сообщениями, таких как AIM, ICQ, WLM и Yahoo, XMPP является федеративной, расширяемой и открытой системой. Любой желающий может запустить свой сервер мгновенного обмена сообщениями, регистрировать на нём пользователей и взаимодействовать с другими серверами XMPP.

1.3 Выбор протокола для решения задачи

В качестве реализуемого в работе протокола, будет использован модифицированный протокол IRC с использованием шаблона издатель-подписчик. Данный выбор обоснован наиболее подходящим поведением протокола для реализации групповых чатов.

1.4 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.

1.5 Анализ существующих решений

В качестве существующих решений для анализа выбраны сервисы `cli-chat` [7], `go-cli-chat` [8], `crio` [9] и `chattt` [10].

В таблице 1.1 представлен сравнительный анализ существующих решений.

Таблица 1.1 – Анализ существующих решений

Сервис	**Максимальное количество комнат**	**Обработка непрочитанных сообщений**	**Максимальное количество пользователей**	**Передача файлов**
cli-chat	Не ограничено	Нет	Не ограничено	Нет
go-cli-chat	1	Нет	Не ограничено	Нет
crio	Не ограничено	Нет	Не ограничено	Нет
chatt	1	Нет	Не ограничено	Нет

Вывод

В данном разделе были рассмотрены и проанализированы протоколы мгновенного обмена сообщения, также выбран протокол, на основе которого будет реализовано программное обеспечение. В качестве основы для протокола был выбран протокол IRC в виду наличия поддержки реализации группового обмена сообщениями.

2 Конструкторская часть

В данном разделе представлены этапы проектирования 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.

В данной работе сервер – это программа, принимающая от пользователей сообщения (или команды), отправленные через клиент, и отправляющая другим пользователям принятые сообщения (шаблон издатель-подписчик), а клиент – программа, принимающая сообщения (или команды) от пользователя через устройство ввода и отправляющая полученные данные на сервер. Также клиент принимает сообщения от сервера. На основе этих сообщений строится логика взаимодействия пользователя с программой. Сервер в данном программном обеспечении выступает только как брокер сообщений.

2.2 Сценарий использования

На рисунках 2.1 и 2.2 представлены сценарии взаимодействия пользователя с клиентским приложением и сервера с полученным сообщением соответственно.

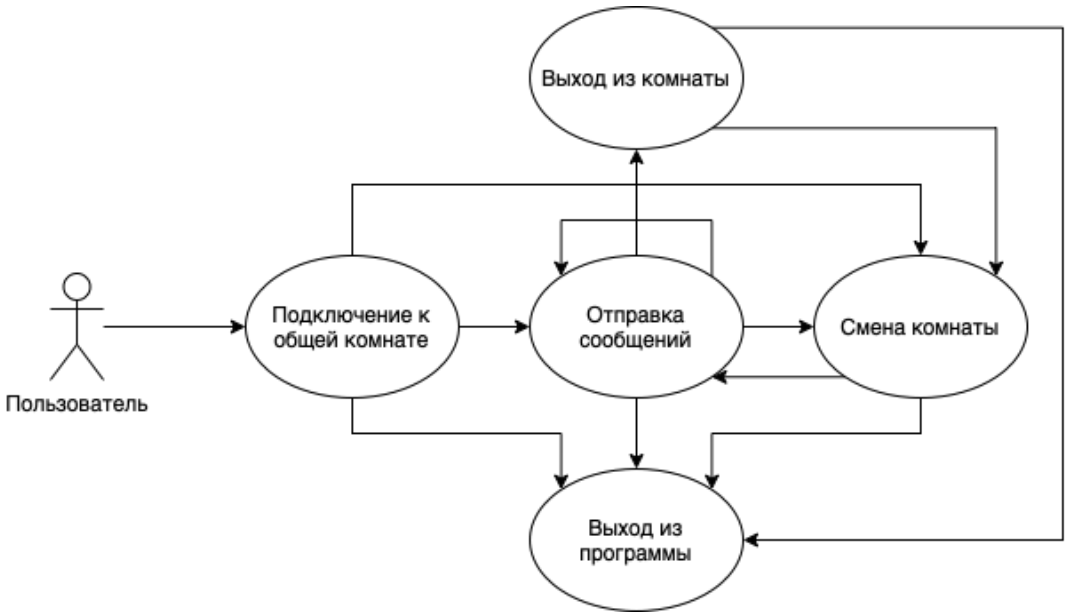

Рисунок 2.1 – Сценарий взаимодействия пользователя с клиентским приложением

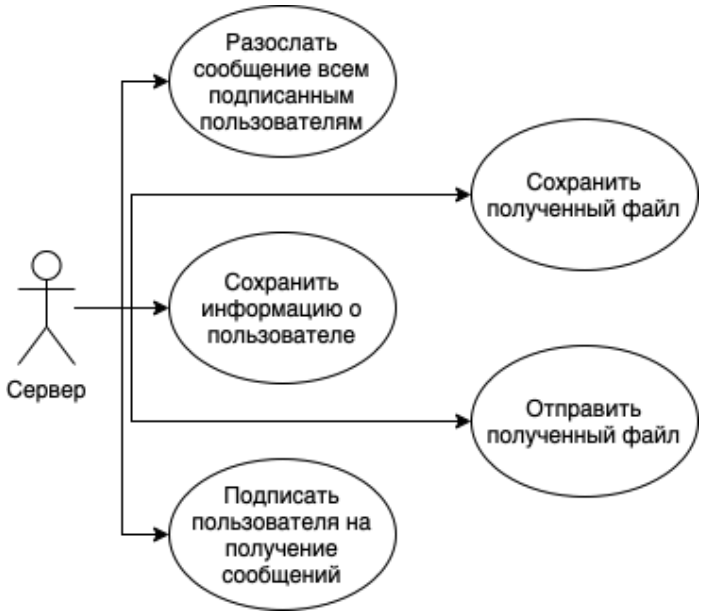

Рисунок 2.2 – Сценарий взаимодействия сервера с полученным сообщением

2.3 Проектирование зон ответственности компонентов

В данном программном обеспечении сервер играет роль только брокера сообщений. Вся логика их обработки сосредоточена на клиенте. В зависи-

мости от полученного сообщения от сервера клиент выполняет различные функции:

- если поступила команда (сообщение, которое начинается с символа /) – выполнить действие на основе команды (создание комнаты, смена комнаты, выход из комнаты, загрузка или отправка файла, получение информации о файлах);
- если поступил обычный текст – отобразить данный текст в окне чата.

Информация о существующих чатах, пользователях чата и файлах чата находится на сервере. Информация о чатах пользователя хранится только на клиенте в момент работы программы.

Сервер не хранит историю сообщений, так как реализован по модели издатель-подписчик. Как только сообщение попадает на сервер, оно тут же рассылается всем подписанным клиентам и никак не запоминается на сервере.

В случае, если пользователь находится в нескольких чатах одновременно, клиент на своей стороне хранит те сообщения, которые были получены в неактивных чатах (все чаты кроме открытого на данный момент считаются неактивными) и показывает их в тот момент, когда пользователь меняет активный чат на один из неактивных. Сообщения, которые были показаны до смены активного чата, показаны не будут.

Вывод

В данном разделе были представлены сценарии взаимодействия пользователя с клиентом и сервера с полученными сообщениями. Были спроектированы зоны ответственности клиентской и серверной части приложения.

3 Технологическая часть

В данном разделе представлены средства разработки программного обеспечения, детали реализации и пользовательский интерфейс.

3.1 Средства реализации

Для разработки клиентской части приложения был выбран язык `Go` [11]. Выбор обусловлен тем, что данный язык является компилируемым, следовательно в процессе разработке возникнет меньше промежуточных ошибок. Также Go предоставляет нативную поддержку многопоточности, что позволяет повысить производительность приложения. В качестве библиотеки для реализации пользовательского интерфейса была выбрана библиотека `gocui` [12]. Выбор библиотеки обусловлен опытом работы с данным инструментом.

Для реализации серверной части приложения была выбрана СУБД `Redis` [13]. Данная СУБД может выступить брокером сообщений нативно, без дополнительных надстроек, так как изначально подерживает шаблон издатель-подписчик [14] и имеет интерфейс, реализующий его работу. Кроме того, ее можно использовать для хранения информации о чатах (пользователи, файлы).

3.2 Детали реализации

В листингах 3.1 – 3.3 представлены листинги реализации интерфейса взаимодействия клиента с брокером сообщений (сервером), обработка полученных сообщений от сервера клиентом и функции обработки команд и обновления состояния на стороне клиента.

Листинг 3.1 – Реализация интерфейса взаимодействия клиента с брокером сообщений

```
1 func NewRedisClient(ctx context.Context, options *Options) (*RedisClient, error) {
2     lock.Lock()
3     defer lock.Unlock()
4
```

```
    if redisClient == nil {
        client, err := getRedisClient(ctx, options)
        if err != nil {
            return nil, fmt.Errorf("failed to initialize redis client, error is: %s", err)
        }
        redisClient = &RedisClient{
            client: client,
        }
        return redisClient, nil
    }

    return redisClient, nil
}

func getRedisClient(ctx context.Context, options *Options) (*redis.Client, error) {
    opts := redis.Options{
        Addr: options.Addr,
        Password: options.Password,
        DB: options.DB,
    }
    client := redis.NewClient(&opts)

    if err := client.Ping(ctx).Err(); err != nil {
        return nil, fmt.Errorf("redis client with ttl %s failed to ping address %s, error
            is: %s",
            opts.IdleTimeout, opts.Addr, err)
    }

    return client, nil
}

func (rc *RedisClient) Set(ctx context.Context, key string, value interface{},
    expiration time.Duration) error {
    return rc.client.Set(ctx, key, value, expiration).Err()
}

func (rc *RedisClient) Get(ctx context.Context, key string) ([]byte, error) {
    return rc.client.Get(ctx, key).Bytes()
}

func (rc *RedisClient) Publish(ctx context.Context, channel string, message interface{})
    error {
    return rc.client.Publish(ctx, channel, message).Err()
}

func (rc *RedisClient) Subscribe(ctx context.Context, channel string) <-chan string {
    redisChan := rc.client.Subscribe(ctx, channel).Channel()
    messagesChan := make(chan string)
```

```
    go func(ch <-chan *redis.Message) {
        for msg := range ch {
            messagesChan <- msg.Payload
        }
    }(redisChan)

    return messagesChan
}
```

Листинг 3.2 – Обработка полученных сообщений от сервера клиентом

```
func Connect(g *gocui.Gui, v *gocui.View) error {
    user = domain.NewUser(strings.TrimSpace(v.Buffer()))
    activeChat = getActiveChat(ctx, cacheClient, "global", user, true)
    _, err := addChatUser(user, activeChat, usersRepo)
    if err != nil {
        log.Fatalln(err)
    }
    handleJoin(ctx, cacheClient, activeChat, user)

    g.SetViewOnTop("messages")
    g.SetViewOnTop("users")
    g.SetViewOnTop("input")
    g.SetViewOnTop("chats")
    g.SetCurrentView("input")

    messagesView, _ := g.View("messages")
    usersView, _ := g.View("users")
    chatsView, _ := g.View("chats")

    updateChatsView(g, chatsView, user, true)

    go func() {
        for {
        chatLoop:
            for msg := range activeChat.GetMessages() {
                switch {
                case strings.HasPrefix(msg, "/users>"):
                    updateUsersView(g, usersView, activeChat, usersRepo, true)

                case getCommandFromMessage(msg) == "/join":
                    if getUserFromMessage(msg) != user.GetName() {
                        break
                    }

                    newChatName := strings.TrimSpace(strings.SplitAfter(msg, "/join")[1])
                    if _, ok := user.GetChats()[newChatName]; ok {
                        updateMessagesView(
                            g,
```

```
                        messagesView,
                        aurora.Sprintf(
                            aurora.Yellow("You are already in %s. Use %s to move
                                there"),
                            aurora.Green(newChatName),
                            aurora.Green(fmt.Sprintf("/switch %s", newChatName)),
                        ),
                        false,
                    )
                    break
                }

                activeChat = getActiveChat(ctx, cacheClient, newChatName, user, true)
                _, err = addChatUser(user, activeChat, usersRepo)
                if err != nil {
                    log.Fatalln(err)
                }
                handleJoin(ctx, cacheClient, activeChat, user)

                updateUsersView(g, usersView, activeChat, usersRepo, true)
                updateChatsView(g, chatsView, user, true)
                clearView(g, messagesView)

                updateMessagesCounter(g, chatsView, user)
                break chatLoop

            case getCommandFromMessage(msg) == "/switch":
                if getUserFromMessage(msg) != user.GetName() {
                    break
                }

                newChatName := strings.TrimSpace(strings.SplitAfter(msg, "/switch")[1])
                if _, ok := user.GetChats()[newChatName]; !ok {
                    updateMessagesView(
                        g,
                        messagesView,
                        aurora.Sprintf(
                            aurora.Yellow("There is no %s chat, Use %s to create it"),
                            aurora.Green(newChatName),
                            aurora.Green(fmt.Sprintf("/join %s", newChatName)),
                        ),
                        false,
                    )
                    break
                }
                if newChatName == activeChat.GetName() {
                    updateMessagesView(g, messagesView, aurora.Yellow("Can't switch to
                        current active chat").String(), false)
```

```
                    break
                }

                activeChat = getActiveChat(ctx, cacheClient, newChatName, user, false)
                _, err = addChatUser(user, activeChat, usersRepo)
                if err != nil {
                    log.Fatalln(err)
                }

                updateMessagesViewWithBuffer(g, messagesView, activeChat, true)
                updateUsersView(g, usersView, activeChat, usersRepo, true)
                updateChatsView(g, chatsView, user, true)

                updateMessagesCounter(g, chatsView, user)
                break chatLoop

            case getCommandFromMessage(msg) == "/leave":
                if getUserFromMessage(msg) != user.GetName() {
                    break
                }

                if activeChat.GetName() == "global" {
                    updateMessagesView(g, messagesView, aurora.Red("Can't␣leave␣
                        global").String(), false)
                    break
                }

                _, err = removeChatUser(user, activeChat, usersRepo)
                if err != nil {
                    log.Fatalln(err)
                }
                handleLeave(ctx, cacheClient, activeChat, user, "left␣the␣chat")
                user.RemoveChat(activeChat.GetName())
                activeChat = getActiveChat(ctx, cacheClient,
                    user.GetActiveChat().GetName(), user, false)

                updateMessagesViewWithBuffer(g, messagesView, activeChat, true)
                updateUsersView(g, usersView, activeChat, usersRepo, true)
                updateChatsView(g, chatsView, user, true)

                updateMessagesCounter(g, chatsView, user)
                break chatLoop

            case getCommandFromMessage(msg) == "/upload":
                if getUserFromMessage(msg) != user.GetName() {
                    break
                }

```

```
131                 fileName := strings.Split(msg, "␣")[3]
132                 data, err := os.ReadFile(fileName)
133                 if err != nil {
134                     updateMessagesView(g, messagesView, aurora.Red("Can't␣open␣file.␣
                        Check␣whether␣path␣is␣correct").String(), false)
135                     break
136                 }
137                 saveFileName := strings.Split(msg, "␣")[4]
138                 _, err = addChatFile(domain.NewFile(saveFileName, data), activeChat,
                    filesRepo)
139                 if err != nil {
140                     updateMessagesView(g, messagesView, aurora.Red("Error␣while␣
                        uploading␣file.␣Try␣again␣later").String(), false)
141                     break
142                 }
143
144                 updateMessagesView(g, messagesView, aurora.Green("File␣is␣
                    uploaded").String(), false)
145
146             case getCommandFromMessage(msg) == "/download":
147                 if getUserFromMessage(msg) != user.GetName() {
148                     break
149                 }
150
151                 files, err := filesRepo.GetFiles(activeChat.GetName())
152                 if err != nil {
153                     updateMessagesView(g, messagesView, aurora.Red("Error␣while␣
                        fetching␣chat␣files").String(), false)
154                     break
155                 }
156                 fileName := strings.Split(msg, "␣")[3]
157
158                 data, ok := files[fileName]
159                 if !ok {
160                     updateMessagesView(g, messagesView, aurora.Red("There␣is␣no␣file␣
                        with␣given␣name␣for␣this␣chat").String(), false)
161                     break
162                 }
163
164                 saveFileName := strings.Split(msg, "␣")[4]
165                 if err := os.WriteFile(saveFileName, data, 0644); err != nil {
166                     updateMessagesView(g, messagesView, aurora.Red("Error␣while␣saving␣
                        data").String(), false)
167                     break
168                 }
169
170                 updateMessagesView(g, messagesView, aurora.Green("File␣is␣
                    downloaded").String(), false)
```

```
171
172                 case getCommandFromMessage(msg) == "/files":
173                     if getUserFromMessage(msg) != user.GetName() {
174                         break
175                     }
176
177                     files, err := filesRepo.GetFiles(activeChat.GetName())
178                     if err != nil {
179                         updateMessagesView(g, messagesView, aurora.Red("Error while
                            fetching chat files").String(), false)
180                         break
181                     }
182                     if len(files) == 0 {
183                         updateMessagesView(g, messagesView, aurora.Yellow("No files
                            available for this chat").String(), false)
184                         break
185                     }
186
187                     header := "Chat files:\n"
188                     res := ""
189                     for k := range files {
190                         res += k + "\n"
191                     }
192                     res = strings.TrimSpace(res)
193
194                     updateMessagesView(g, messagesView, aurora.Sprintf("%s%s", header,
                        aurora.Blue(res)), false)
195
196                 default:
197                     updateMessagesView(g, messagesView, msg, false)
198                 }
199             }
200         }
201     }()
202
203     return nil
204 }
```

Листинг 3.3 – Обработка команд и обновление состояния на стороне клиента

```
1 func addChatUser(user *domain.User, chat *domain.Chat, repo usersrepo.UsersRepository)
      (map[string]struct{}, error) {
2     users, err := repo.GetUsers(chat.GetName())
3     if err != nil {
4         return nil, err
5     }
6
```

```
    if users == nil {
        users = make(map[string]struct{})
    }
    users[user.GetName()] = struct{}{}

    err = repo.SetUsers(chat.GetName(), users)
    if err != nil {
        return nil, err
    }

    return users, nil
}

func removeChatUser(user *domain.User, chat *domain.Chat, repo
    usersrepo.UsersRepository) (map[string]struct{}, error) {
    users, err := repo.GetUsers(chat.GetName())
    if err != nil {
        return nil, err
    }

    delete(users, user.GetName())

    err = repo.SetUsers(chat.GetName(), users)
    if err != nil {
        return nil, err
    }

    return users, nil
}

func addChatFile(file *domain.File, chat *domain.Chat, repo filesrepo.FilesRepository)
    (map[string][]byte, error) {
    files, err := repo.GetFiles(chat.GetName())
    if err != nil {
        return nil, err
    }

    if files == nil {
        files = make(map[string][]byte)
    }
    files[file.GetName()] = file.GetContent()

    err = repo.SetFiles(chat.GetName(), files)
    if err != nil {
        return nil, err
    }

    return files, nil
```

```
53 }
54
55 func updateChatsView(g *gocui.Gui, v *gocui.View, user *domain.User, toClear bool) {
56     g.Update(func(g *gocui.Gui) error {
57         chats := ""
58         for chatName, chat := range user.GetChats() {
59             counter := ""
60             count := len(chat.GetBuffer())
61             if count > 0 {
62                 counter = fmt.Sprintf("␣(%d)", count)
63             }
64             if chat.GetIsActive() {
65                 chats += aurora.Sprintf(aurora.Green("#%s%s\n"), chatName,
                       aurora.Magenta(counter))
66             } else {
67                 chats += aurora.Sprintf("#%s%s\n", chatName, aurora.Magenta(counter))
68             }
69         }
70
71         if toClear {
72             v.Clear()
73         }
74         fmt.Fprintln(v, chats)
75
76         return nil
77     })
78 }
79
80 func updateMessagesView(g *gocui.Gui, v *gocui.View, message string, toClear bool) {
81     g.Update(func(g *gocui.Gui) error {
82         if toClear {
83             v.Clear()
84         }
85         fmt.Fprintln(v, message)
86
87         return nil
88     })
89 }
90
91 func updateMessagesViewWithBuffer(g *gocui.Gui, v *gocui.View, chat *domain.Chat,
      toClear bool) {
92     g.Update(func(g *gocui.Gui) error {
93         if toClear {
94             v.Clear()
95         }
96         for _, bufferMessage := range chat.GetBuffer() {
97             fmt.Fprintln(v, bufferMessage)
98         }
```

```
99          chat.SetBuffer(nil)
100
101         return nil
102     })
103 }
104
105 func updateUsersView(g *gocui.Gui, v *gocui.View, chat *domain.Chat, repo
        usersrepo.UsersRepository, toClear bool) {
106     g.Update(func(g *gocui.Gui) error {
107         users, err := repo.GetUsers(chat.GetName())
108         if err != nil {
109             log.Fatalln(err)
110         }
111         chatUsers := ""
112         chatUsersCount := len(users)
113         for u := range users {
114             chatUsers += u + "\n"
115         }
116
117         v.Title = fmt.Sprintf("%d␣users:", chatUsersCount)
118         if toClear {
119             v.Clear()
120         }
121         fmt.Fprintln(v, chatUsers)
122
123         return nil
124     })
125 }
126
127 func clearView(g *gocui.Gui, v *gocui.View) {
128     g.Update(func(g *gocui.Gui) error {
129         v.Clear()
130         return nil
131     })
132 }
133
134 func getActiveChat(ctx context.Context, cc *cache.RedisClient, name string, user
        *domain.User, isNew bool) *domain.Chat {
135     var chat *domain.Chat
136     if isNew {
137         chat = domain.NewChat(name)
138         user.AddChat(chat)
139     } else {
140         user.SetActiveChat(name)
141         chat = user.GetActiveChat()
142     }
143     chat.SetMessages(cc.Subscribe(ctx, name))
144
```

```
145     return chat
146 }
147
148 func getUserFromMessage(message string) string {
149     return strings.TrimRight(strings.Split(message, "␣")[1], ":")
150 }
151
152 func getCommandFromMessage(message string) string {
153     return strings.Split(message, "␣")[2]
154 }
155
156 func handleJoin(ctx context.Context, cc *cache.RedisClient, chat *domain.Chat, user
        *domain.User) {
157     cc.Publish(ctx, chat.GetName(), "/users>")
158     cc.Publish(ctx, chat.GetName(), aurora.Sprintf(aurora.Green("%s␣just␣joined!"),
            aurora.Yellow(user.GetName())))
159 }
160
161 func handleLeave(ctx context.Context, cc *cache.RedisClient, chat *domain.Chat, user
        *domain.User, message string) {
162     cc.Publish(ctx, chat.GetName(), "/users>")
163     cc.Publish(ctx, chat.GetName(), aurora.Sprintf(aurora.Red("%s␣just␣%s␣:("),
            aurora.Yellow(user.GetName()), message))
164 }
165
166 func updateMessagesCounter(g *gocui.Gui, v *gocui.View, user *domain.User) {
167     for _, chat := range user.GetChats() {
168         go func(c *domain.Chat) {
169             if c.GetIsActive() {
170                 return
171             }
172             for msg := range c.GetMessages() {
173                 if c.GetIsActive() {
174                     return
175                 }
176                 msgSplit := strings.Split(msg, "␣")
177                 if strings.HasPrefix(msg, "/") || strings.HasPrefix(msgSplit[2], "/") {
178                     continue
179                 }
180                 c.SetBuffer(append(c.GetBuffer(), msg))
181                 updateChatsView(g, v, user, true)
182             }
183         }(chat)
184     }
185 }
```

3.3 Пользовательский интерфейс

На рисунках 3.1 – 3.3 представлены примеры работы программного обеспечения.

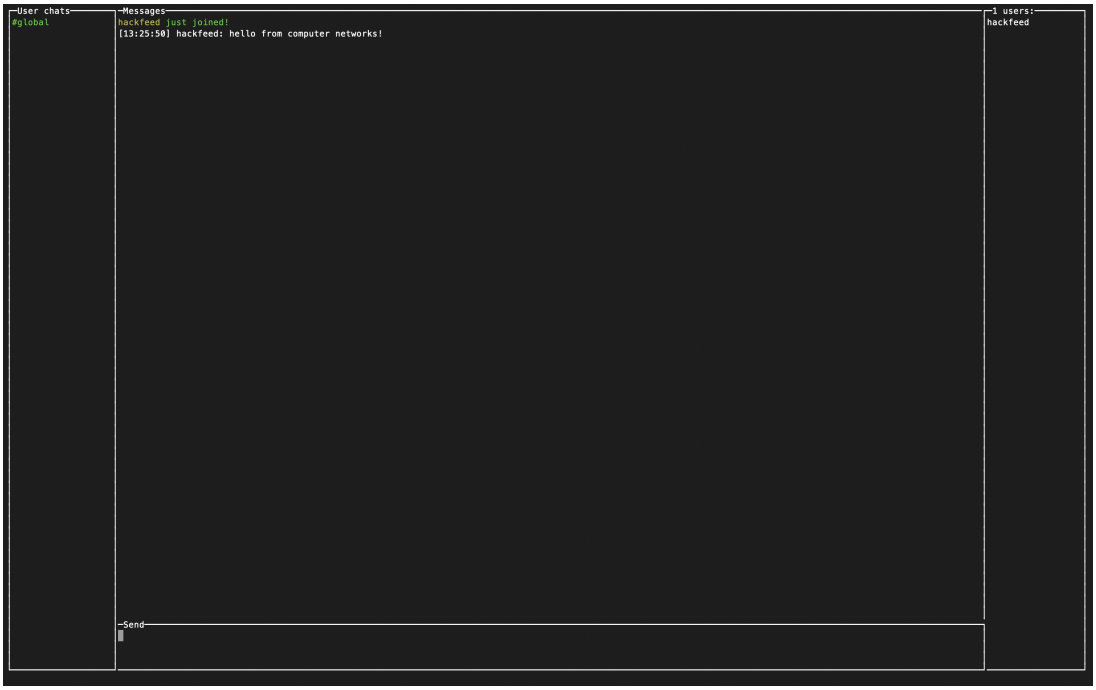

Рисунок 3.1 – Отправка сообщения в чат и подключение нового пользователя

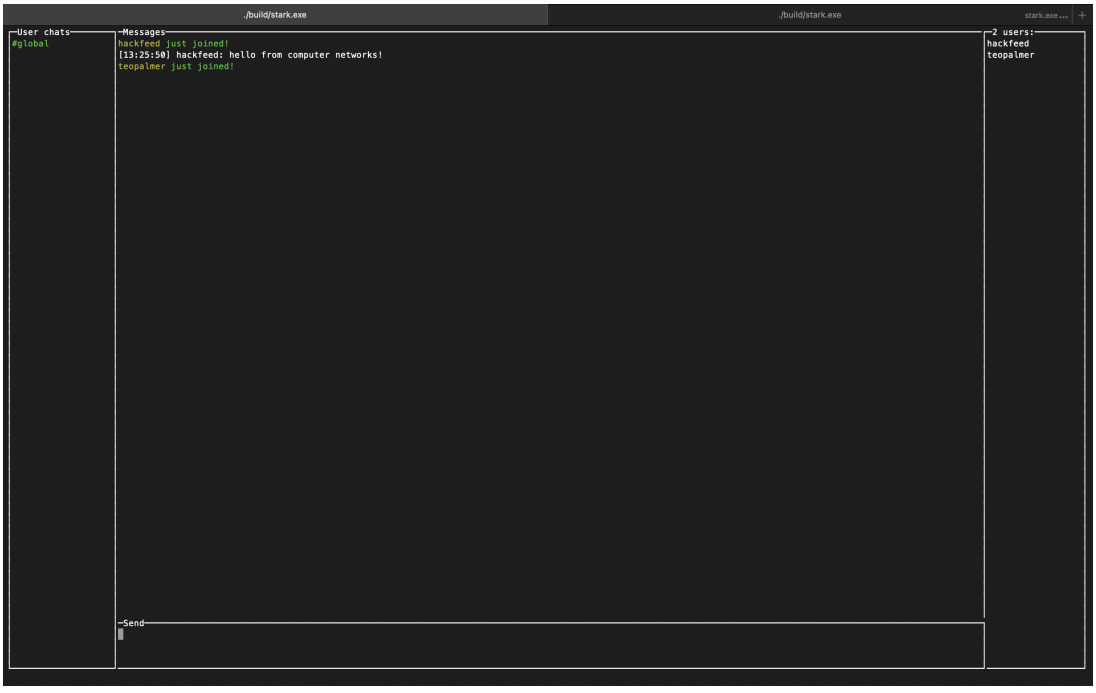

Рисунок 3.2 – Подключение нового пользователя

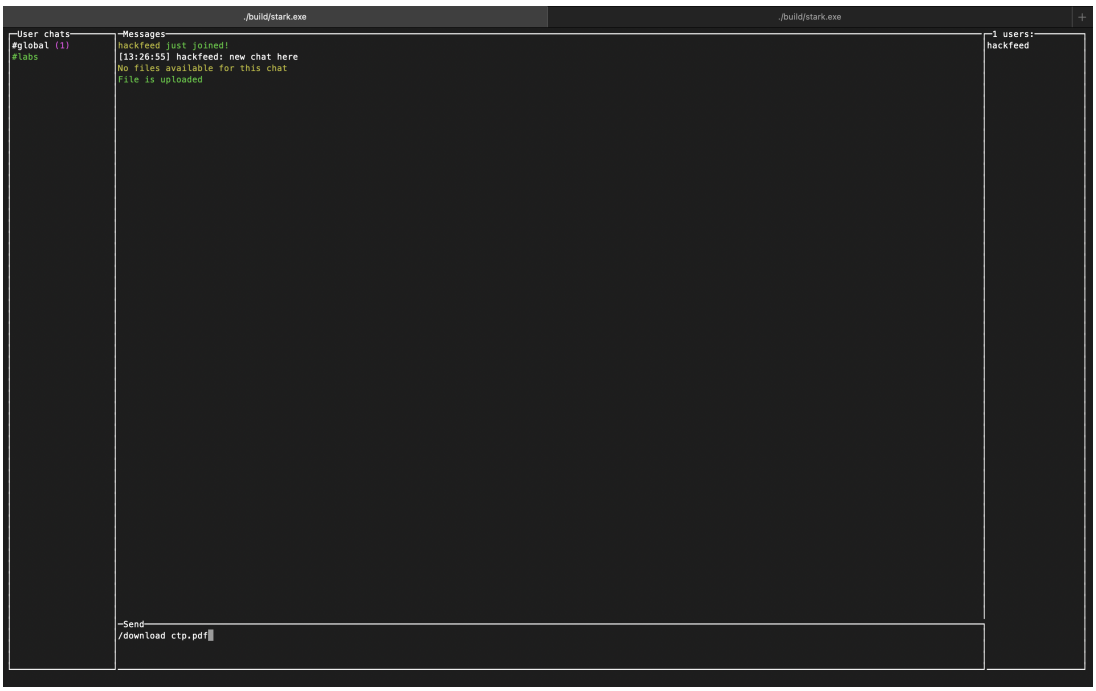

Рисунок 3.3 – Смена комнаты и загрузка файла

Вывод

В данном разделе были представлены средства реализации программного обеспечения, листинги ключевых компонентов системы а также представлен пользовательский интерфейс приложения.

Заключение

Во время выполнения курсовой работы было реализовано программное обеспечение, реализующее многопользовательский чат реального времени с несколькими комнатами и возможностью отправки файлов.

В ходе выполнения поставленной задачи были получены знания в области протоколов передачи мгновенных сообщений. Поиск подходящего решения для поставленной задачи позволил повысить навыки поиска и анализа информации.

В результате проведенной работы было разработано программное обеспечение, демонстрирующее работу чата с использованием шаблона издатель-подписчик на основе протокола IRC.

Литература

[1] instant messaging | communication | Britannica [Электронный ресурс]. Режим доступа: https://www.britannica.com/topic/instant-messaging (дата обращения: 24.12.2021).

[2] meteor/DDP.md at devel · meteor/meteor [Электронный ресурс]. Режим доступа: https://github.com/meteor/meteor/blob/devel/packages/ddp/DDP.md (дата обращения: 24.12.2021).

[3] Шаблон издателя и подписчика - Azure Architecture Center | Microsoft Docs [Электронный ресурс]. Режим доступа: https://docs.microsoft.com/ru-ru/azure/architecture/patterns/publisher-subscriber (дата обращения: 24.12.2021).

[4] What is IRC (Internet Relay Chat)? [Электронный ресурс]. Режим доступа: https://mason.gmu.edu/~montecin/IRC.html (дата обращения: 24.12.2021).

[5] Matrix.org [Электронный ресурс]. Режим доступа: https://matrix.org (дата обращения: 24.12.2021).

[6] XMPP | The universal messaging standard [Электронный ресурс]. Режим доступа: https://xmpp.org (дата обращения: 24.12.2021).

[7] ehrenjn/cli-chat: A basic command line chat program for linux and windows written in python [Электронный ресурс]. Режим доступа: https://github.com/ehrenjn/cli-chat (дата обращения: 24.12.2021).

[8] Luqqk/go-cli-chat: Chat server and command line interface client (CLI) written in Go. [Электронный ресурс]. Режим доступа: https://github.com/Luqqk/go-cli-chat (дата обращения: 24.12.2021).

[9] Crio Projects - CLI Based Chat Tool | Crio.Do | Project-Based Learning Platform for Developers [Электронный ресурс]. Режим доступа: https://www.crio.do/projects/python-cli-chat/ (дата обращения: 24.12.2021).

[10] Introducing Chattt - an open source CLI chat client — Steemit [Электронный ресурс]. Режим доступа: https://steemit.com/utopian-io/@the-dragon/introducing-chattt-an-open-source-cli-chat-client (дата обращения: 24.12.2021).

[11] The Go Programming Language [Электронный ресурс]. Режим доступа: https://golang.org/ (дата обращения: 24.12.2021).

[12] jroimartin/gocui: Minimalist Go package aimed at creating Console User Interfaces. [Электронный ресурс]. Режим доступа: https://github.com/jroimartin/gocui (дата обращения: 24.12.2021).

[13] Redis [Электронный ресурс]. Режим доступа: https://redis.io/ (дата обращения: 24.12.2021).

[14] Pub/Sub - Redis [Электронный ресурс]. Режим доступа: https://redis.io/topics/pubsub (дата обращения: 24.12.2021).