

Dokumentation

Matthias Vonend Jan Grübener Patrick Mischka
Michael Angermeier Troy Keßler Aaron Schweig

18. April 2020

[GitHub](#)

Inhaltsverzeichnis

1	Basisanforderungen	1
1.1	Chatfunktionalität	1
1.2	Clientfunktionalitäten	2
1.3	Fehlerbehandlung	3
1.3.1	Client	3
1.3.2	Server	4
2	Erweiterungen	4
2.1	Grafische Oberfläche bei den Nutzern	4
2.2	Verwendung von Emojis	5
2.3	Gruppenchats	5
2.4	Mehrere Chatverläufe pro Nutzer	5
2.5	Persistentes Speichern der Chatverläufe	5
2.6	Verschlüsselte Übertragung der Chat-Nachrichten	6
2.7	Verschlüsselte serverseitige Speicherung der Chats	8
3	Codewalkthrough	9
3.1	Server	9
3.2	Client	18

1 Basisanforderungen

1.1 Chatfunktionalität

Matthias Vonend

Die Anwendung ist mit einem Thin Client aufgebaut. Damit ein Chat ablaufen kann, muss zunächst eine Verbindung zu einem Server aufgebaut werden. Dazu wählt der Client zunächst einen zufälligen Server aus und versucht sich zu verbinden. Wurde eine Verbindung erfolgreich aufgebaut, kann sich der Nutzer mit seinem Nutzernamen und seinem Passwort anmelden. Sobald der Nutzer angemeldet ist, sendet der Server ihm alle benötigten Informationen inklusiver der verpassten Nachrichten. Jede Nachricht hat ein Datum, wann es erstmalig



Abbildung 1: Architektur

an einem Server eingetroffen ist. Anhand von diesem werden die Nachrichten sortiert, damit der Client die korrekte Reihenfolge darstellen kann.

Möchte dieser eine Nachricht an einen weiteren Client senden, schickt er diese an seine verbundene Node und überlässt die Zustellung dieser. Da alle Nachrichten aus Konsistenzgründen an alle Nodes verteilt werden müssen, brauchen die Nodes keine Information über die Clients anderer Nodes. Im Falle einer solchen Anforderung (z. B. Abfrage, ob ein anderer Nutzer aktiv ist) könnte ein Protokoll ähnlich zu Routing-Tabellen implementiert werden. Empfängt eine Node eine Nachricht, egal ob von einem Client oder von einer anderen Node, überprüft diese, ob die Nachricht für einem ihr bekannten Client bestimmt war und sendet diese gegebenenfalls an diesen.

1.2 Clientfunktionalitäten

Jan Grübener, Troy Keffler, Patrick Mischka, Michael Angermeier

Nach einer erfolgreichen Anmeldung kann der Nutzer zwischen verschiedenen Funktionen auswählen.

1. help
2. chats
3. contacts
4. createchat
5. openchat
6. exit

Funktion: help Diese Funktion ist nur im Commandline-Client implementiert und gibt dem Nutzer einen Überblick über alle möglichen Funktionen, die

er aufrufen kann. Alle Funktionen sind in ein paar Worten beschrieben, sodass der Benutzer gleich weiß, was diese Funktion genau macht.

Funktion: chats Bei einem Aufruf dieser Funktion werden alle Chats, die für den Nutzer zugänglich sind, angezeigt. Dafür werden zunächst alle Chats durch die Methode `getChats` aus der API in einem Set aus Chats gespeichert. Je nachdem, ob Chats verfügbar sind, bekommt der Benutzer unterschiedliche Antworten. Sind keine Chats vorhanden, wird dies in einer Meldung angezeigt. Sind Chats vorhanden, werden sowohl die Chatnamen als auch der/die User/s in diesem Chat angezeigt.

Funktion: contacts Bei dieser Funktionen werden alle Benutzer angezeigt, die gerade online sind. Nur mit diesen Benutzern kann ein neuer Chat erstellt werden, da sonst der Schlüsselaustausch nicht funktioniert.

Funktion: createchat Diese Funktion beginnt mit einer Aufforderung an den Benutzer, einen Chatnamen einzugeben. Danach wird die Anzahl der Teilnehmer in dem Chat abgefragt. Diese muss mindestens 1 betragen. Ist die Anzahl an Teilnehmern einmal gesetzt, müssen im nächsten Schritt alle Benutzernamen der Teilnehmer eingetragen werden. Hierfür wird jeder einzelne Benutzername abgefragt und im Falle eines invaliden Benutzernamens, wird der Benutzer durch eine Meldung darauf aufmerksam gemacht. Wurden alle 3 Attribute (Chatname, Teilnehmeranzahl, Username der Teilnehmer) erfolgreich eingegeben, wird ein neuer Chat erstellt.

Funktion: openchat Auch hier muss der Benutzer zuerst den Chatnamen eingeben. Ist dieser vorhanden, wird der Chat geöffnet, ansonsten bekommt der Benutzer wieder eine Meldung. Am Anfang eines Chats wird immer darauf hingewiesen, wie der Chat verlassen werden kann. Danach werden alle Nachrichten, die in diesem Chat bereits geschrieben wurden, geladen. Anschließend kann der Benutzer Nachrichten verschicken und empfangen.

1.3 Fehlerbehandlung

Matthias Vonend, Aaron Schweig, Troy Keßler

Aus den Anforderungen geht hervor, dass es mindestens zwei Server geben muss, die sämtliche Informationen des Chatsystems besitzen müssen. Bricht eine Node zusammen, muss eine andere Node dessen Aufgabe übernehmen.

1.3.1 Client

Nachdem der Client eingeloggt ist, wartet er kontinuierlich auf neue Pakete vom Server, wird beim Warten ein Fehler geworfen bedeutet das, dass diese Node ausgefallen ist. Nun versucht sich die Node wieder zu verbinden. Dabei können die Nodes über die Programmargumente eingegeben werden. Wenn sich der Client erfolgreich auf eine andere Node verbunden hat loggt er sich automatisch mit den bestehen Usernamen und Passwort ein, sodass der User von dem Ausfall nichts mitbekommt. Falls keine von den angegebenen Nodes verfügbar ist, probiert der Client nacheinander auf jede zu verbinden, bis eine davon wieder Online ist.

1.3.2 Server

Serverseitig können verschiedene Fehler auftreten. Viele Fehler werden bereits durch das TCP-Protokoll verhindert (z.B. mehrfach Zustellung, fehlerhafte Übermittlung). Dennoch können grundsätzlich die folgenden Fehlerfälle eintreten:

1. Nachricht des Clients kann nicht korrekt empfangen/gesendet werden
In diesem Fall muss der Server davon ausgehen, dass die Verbindung zusammengebrochen ist und er beendet seine Verbindung. Der Server verlässt sich darauf, dass der Client erneut eine Verbindung aufbaut. Alle für den Client relevanten Nachrichten werden dann zu diesem übertragen und der Client muss neue Informationen zurück übertragen.
2. Nachrichten einer Nachbarnode können nicht korrekt empfangen/gesendet werden
Die Server sind als Peer-to-Peer Server aufgebaut. Demnach muss der Server ähnlich zur Clientverbindung davon ausgehen, dass die Verbindung zusammengebrochen ist. Allerdings ist der Server hier selbst dafür zuständig sich erneut zu verbinden. Sämtliche Nachrichten, die an eine Node gesendet werden müssen, werden in einer Queue aufbewahrt und nacheinander gesendet. Scheitert die Verbindung, so bleibt die Queue unverändert und wird nach einem erneuten Verbinden weiter abgearbeitet. Es wird solange versucht zu verbinden, bis eine Verbindung zustande gekommen ist. Sobald eine Verbindung wieder aufgebaut wurde synchronisieren sich die Nodes, um wieder einen vollständigen Informationsstand zu besitzen. So kommen die Server auch mit Netzwerkpartitionierungen klar. Werden die Server getrennt sind die Clients immer noch in der Lage Nachrichten an ihre jeweiligen Server zu schicken und sobald die Verbindung zwischen den Servern wieder aufgebaut werden konnte gleichen sich diese an. Sind keine Nachrichten zu senden, hat die Node keine Möglichkeit festzustellen, ob eine Verbindung noch existiert. Zu diesem Zweck existiert ein Heartbeat, der periodisch die Nachbarnodes anpingt und so prüft, ob die Verbindung noch existiert.

Wie gerade beschrieben führen alle Beteiligten stets eine Synchronisation, wodurch diese immer den kompletten Informationsbestand besitzen. Die Replikationskontrolle wird nach der Write-All-Available Strategie umgesetzt. Ein Client schickt eine Nachricht an einen Server. Der Server versucht alle zu ihm verbundenen Nodes aktuell zu halten indem er die Nachricht oder die Änderung an alle verfügbaren Nodes weiterleitet.

Eine Veränderung des Datenbestandes muss dem entsprechend an alle anderen Nodes weitergegeben werden. Dadurch sind alle Server gleichwertige.

2 Erweiterungen

2.1 Grafische Oberfläche bei den Nutzern

Jan Grübener, Patrick Mischka

Wählt der Nutzer nach dem Starten des Clients die Variante mit Grafischer Benutzeroberfläche starten, ruft der Client die Methode `startGui()` auf. Hierbei

wird ein JFrame erzeugt, auf dem im Laufe der Zeit unterschiedliche JPanels hinzugefügt und entfernt werden. Je nachdem an welchem Punkt der Nutzer sich gerade befindet werden die entsprechenden Methoden wie `loginPanel()` oder `displayRecentConversations()` für die jeweilige Funktionalität aufgerufen.

Bei der Instanziierung des GUI-Objekts wird ebenfalls ein damit referenziertes API-Objekt instanziiert. Dieses API-Objekt beinhaltet die Methode `startPacketListener`. Diese Methode wird bei erfolgreichem login in der GUI aufgerufen. Die `startPacketListener` Methode startet einen Thread, der durchgehend einkommende Pakete annimmt, sie überprüft und entsprechende Methodenreferenzen in dem GUI-Objekt aufruft. Neue Chats, einkommende Nachrichten und neue Nutzer werden so in die GUI übernommen.

2.2 Verwendung von Emojis

Jan Grübener, Patrick Mischka

Java wandelt den Unicode automatisch in das zugehörige Emoji um, sodass keine Icons für die Emojiauswahl oder weitere Anpassungen im Frontend nötig waren. Bei der Auswahl eines Emojis wird ein Objekt der Klasse `EmojiMouseListener` instanziiert, welches den entsprechenden Unicode Wert an die `JTextArea` anhängt. So ist es jederzeit möglich die Anzahl der Emojis zu erhöhen oder Emojis zu tauschen. Die Darstellung der Emojis unterscheidet sich leicht je nach ausgewählter Schriftart oder Betriebssystem.

Auf Serverseite mussten hierfür keine Veränderungen vorgenommen werden.

2.3 Gruppenchats

Matthias Vonend, Aaron Schweig, Troy Keßler

Gruppenchats stellen nur eine Erweiterung der bestehenden Chatimplementierung dar. Statt das ein Chat nur die beiden Teilnehmer besitzt, besitzt es nun beliebig viele Nutzer. Wird eine Nachricht empfangen werden nun alle am chat-beteiligten Nutzer durchlaufen und die Nachricht wird an alle der Node bekannten Clients weitergeleitet. Außerdem wird wie bereits erwähnt die Nachricht aus Konsistenzgründen weiter gebroadcastet.

2.4 Mehrere Chatverläufe pro Nutzer

Matthias Vonend, Troy Keßler

Mehrere Chatverläufe sind ähnlich zu den Gruppenchats eine Erweiterung der bestehenden Chatimplementierung. Nun ist es möglich mehrere Chats zu erstellen und zwischen diesen zu wechseln. //TODO @Troy kannst du hier beschreiben wie das im Client ist (also `openChat`)

2.5 Persistentes Speichern der Chatverläufe

Matthias Vonend

Sämtliche der Node bekannten Informationen (Nutzer, Chats, Nachrichten) werden in einem Warehouse verwaltet. Um diese Informationen zwischen Node-Neustarts zu persistieren muss demnach das Warehouse als Datei gespeichert

werden und bei Node-Start wieder geladen werden. Existiert noch kein Speicherstand, wird die Node mit einem leeren Warehouse initialisiert. Während die Node ausgefallen ist, können andere Nodes gleichzeitig neue Informationen erhalten haben. Sobald die Node wieder verfügbar ist müssen andere Nodes dies bemerken und die ausgefallene Node mit Informationen versorgen. Um den Ausfall festzustellen ist ein Heart-Beat vorgesehen. Dieser pingt jede Sekunde alle benachbarten Nodes an, um sicherzustellen, dass die Verbindung noch existiert. Sollte eine Unterbrechung festgestellt, versucht die Node die Verbindung wieder aufzubauen. Gelingt dies, so wird das Warehouse übersendet. Jede Node prüft, ob sie alle Informationen des empfangenen Warehouses bereits besitzt und fügt neue Informationen hinzu. Sofern die Node neue Informationen erhalten haben, broadcastet diese ihren neuen Stand an alle benachbarten Nodes um diese auch auf den neuesten Stand zu bringen.

Der Speichervorgang kann je nach System und Warehousegröße längere Zeiten in Anspruch nehmen. In dieser Zeit können in der Regel keine weiteren Anfragen verarbeitet werden. Um diese Zeit zu minimieren kümmert sich ein eigener Thread um die Speicherung und speichert das Warehouse in Intervallen. Zu beachten ist der Fall, dass eine Node zusammenbricht während der Speichervorgang in Arbeit ist. In diesem Fall würde die Node sämtliche Informationen verlieren, da die Speicherdatei korrupt ist. Um dies zu verhindern wird der Speicherstand zunächst in eine Tempdatei geschrieben und nach erfolgreicher Speicherung an den Zielort verschoben.

2.6 Verschlüsselte Übertragung der Chat-Nachrichten

Troy Keffler, Michael Angermeier

Um einen sicheren Nachrichtenkanal zu gewährleisten wurde eine Ende-zu-Ende-Verschlüsselung implementiert. Dabei wird beim Erstellen eines Chats unter allen Teilnehmern ein Diffie-Hellman Schlüsselaustausch durchgeführt, sodass jeder Client denselben Schlüssel für einen Chat besitzt. Diese generierten Schlüssel werden beim Client zur Chat ID lokal gespeichert, sodass der auch nach einem Neustart weiter den Chat nutzen kann. Somit kann der Client, bevor er Nachrichten zum Server sendet, den Inhalt mit dem jeweiligen Chat Schlüssel verschlüsseln und ankommende Nachrichten entschlüsseln. Somit wird der Server ausschließlich verschlüsselte Nachrichten erhalten, auf die er keinen Zugriff hat.

Die öffentlichen Schlüssel wurden dabei für jeden Client festgeschrieben. Nach jedem Login und nach Erstellen eines Chats generiert der Client einen neuen 128bit Schlüssel und löscht den alten, um die Sicherheit zu erhöhen. Um auch Gruppenchats ermöglichen zu können musste der Diffie-Hellman-Schlüsselaustausch entsprechend erweitert werden. Dabei gibt es in Gruppen im Gegensatz zu zwei Teilnehmern mehrere Runden. Ein Schlüsselaustausch mit drei Teilnehmern kann aus Abbildung 2 entnommen werden.

Wie man erkennen kann werden in der ersten Runde (Schritte 1-3) die ersten Teilschlüssel wie im klassischen Diffie-Hellman generiert und weitergeschickt. In der zweiten Runde wird mithilfe der Ergebnisse der ersten Runde, die fertigen Chat Schlüssel berechnet (Schritte 4-9). Die Größe der Primzahl n beträgt 128bit, wobei die Länge der Generatorprimzahl 32bit ist. Somit ergibt sich für den Chat ein Schlüssel von ebenfalls 128bit.

g = Erzeugerprimzahl
 n = 128bit Primzahl

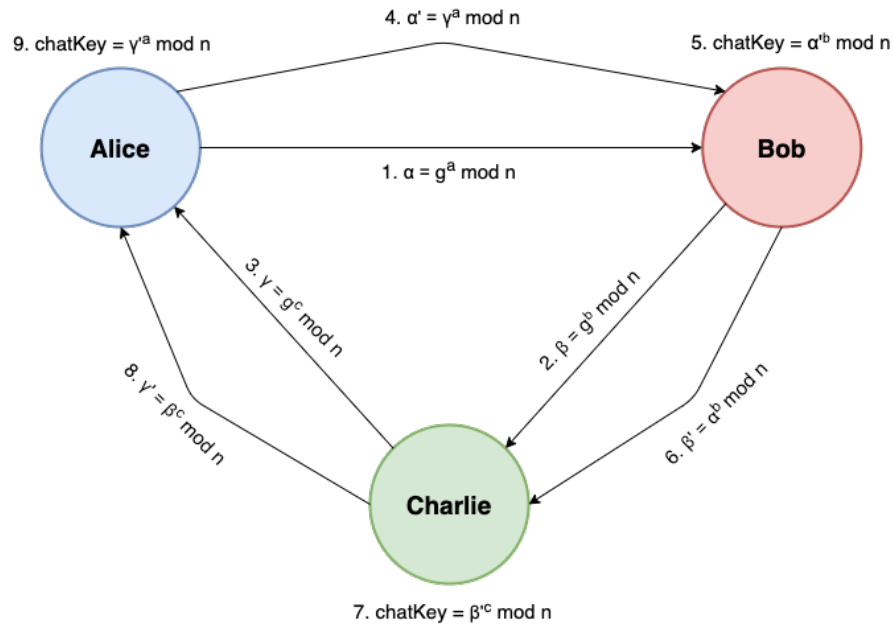


Abbildung 2: Erweiterter Diffie-Hellman

Dieses Verfahren kann mit beliebig vielen Teilnehmern durchgeführt werden, jedoch steigt die Anzahl der Runden linear und die Anzahl der Requests quadratisch.

$$\text{rounds} = n - 1$$

$$\text{requests} = n \cdot (n - 1)$$

Wobei hier n die Anzahl der Teilnehmer ist.

Aus Implementierungssicht braucht der Initiator dieses Schlüsselaustauschs eine Endbedingung, sodass er nachdem die Schlüssel unter allen Teilnehmern ausgetauscht wurden den Chat erstellen kann. Der Initiator dieses Schlüsselaustauschs kann nun mit dem Bedingung

$$\text{currentRequests} = (\text{targetRequests} - \text{userIndex})$$

prüfen, ob der Schlüsselaustausch vollständig durchgeführt wurde. Dabei steht *currentRequests* für die Anzahl, wie oft das Paket schon weitergeleitet wurde. Diese wird beim versenden inkrementiert. *targetRequests* ist die Anzahl der theoretischen Requests die gesendet werden müssen, diese beträgt wie oben definiert immer $n \cdot (n - 1)$. *userIndex* steht für die Position im Schlüsselaustausch, der Initiator ist der Erste, sodass dieser bei ihm 0 ist. Im obigen Beispiel wäre

also die Endbedingung erfüllt wenn der Initiator ein Paket erhält, welches schon sechs mal weitergeleitet wurde.

$$6 = (3 * (3 - 1)) - 0$$

Für alle anderen Clients gilt eine leicht abgeänderte Endbedingung.

Da nun alle Chatteilnehmer den gleichen Schlüssel besitzen, dieser lokal gesichert, und der Chat erstellt ist können Nachrichten mit einem symmetrischen Verfahren sicher versendet werden. Für dieses symmetrische Verfahren wurde die AES Verschlüsselung gewählt und implementiert.

2.7 Verschlüsselte serverseitige Speicherung der Chats

Troy Keßler

Durch die in 2.6 beschriebene Ende-zu-Ende Verschlüsselung liegen dem Server die Nachrichten nie in Klartextform vor, sondern stets in der verschlüsselten Form. Wie bereits in 2.5 erwähnt werden alle Nachrichten im Warehouse gespeichert und das gesamte Warehouse wird abgespeichert. So liegen auch in der Speicherdatei die Nachrichten niemals in Klartextform vor.

3 Codewalkthrough

Bei der Implementierung wird Java 11 eingesetzt.

3.1 Server

Einstiegspunkt des Servers ist der Serverbootstrapper. Dieser erstellt einen neuen Thread mit einem neuen Server.

```

8   public static void main(final String[] args) {
    var server = new Server(9876); // new NodeConfig("host", port)
10  var mainThread = new Thread(server);
    mainThread.start();
    }

```

../src/main/java/vs/chat/server/ServerBootstrapper.java

Der Server erstellt die Listener, die die zu empfangenen Pakete behandeln werden. Anschließend werden die Filter erstellt, die bestimmen, ob ein Paket gehandelt oder ignoriert werden soll (z. B. bei rekursiven Broadcasts).

```

64  private List<Filter> createFilters() {
    var filters = new ArrayList<Filter>();
66  filters.add(new PacketIdFilter());
    return filters;
68  }

70  private List<Listener<? extends Packet, ? extends Packet>>
    createListener() {
    var listeners = new ArrayList<Listener<? extends Packet, ?
72  extends Packet>>();
    listeners.add(new CreateChatListener());
    listeners.add(new GetMessagesListener());
74  listeners.add(new LoginListener());
    listeners.add(new MessageListener());
76  listeners.add(new NodeSyncListener());
    listeners.add(new KeyExchangeListener());
78  listeners.add(new BaseEntityBroadcastListener());
    return listeners;
80  }

```

../src/main/java/vs/chat/server/Server.java

Die Listener und die Filter werden in einen ServerContext gepackt, der allen Threads geteilt wird.

Sobald der ServerContext instanziiert wird, wird das Warehouse geladen. Zunächst wird versucht die Safe-Datei zu laden. Scheitert das Laden, wird das Warehouse leer instanziiert.

```

54  public synchronized void load() {
    try (var stream = new FileInputStream(this.saveFileName + ".dat
    ")) {
    var inputStream = new ObjectInputStream(stream);
56  this.warehouse = (ConcurrentHashMap<WarehouseResourceType,
    ConcurrentHashMap<UUID, Warehouseable>>) inputStream
        .readObject();
58  this.packetIds = (Set<UUID>) inputStream.readObject();
    System.out.println("Loaded warehouse:");
    }
    }

```

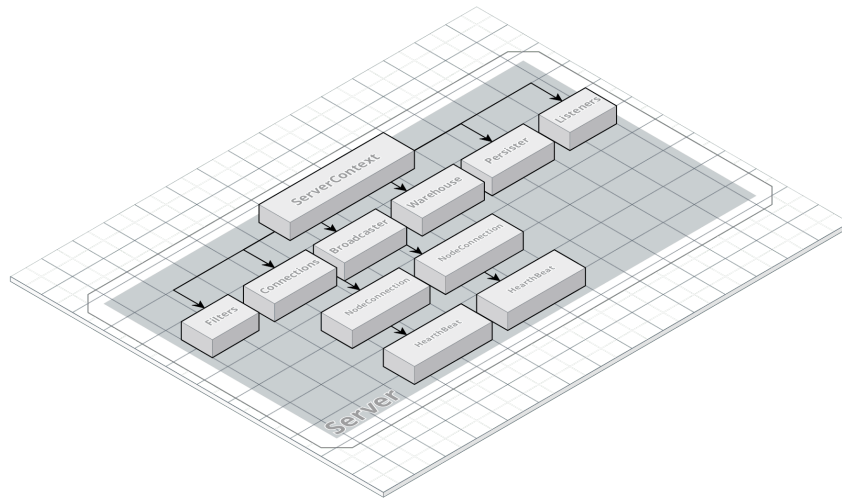


Abbildung 3: Server-Context Aufbau

```

60     this.print();
    } catch (ClassNotFoundException | IOException e) {
62         System.out.println("Couldn't load save file.");
    }
64 }

```

../src/main/java/vs/chat/server/warehouse/Warehouse.java

Das Warehouse hält sämtliche Daten, die persistiert werden müssen (z. B. Messages, Chats, Users). Der ServerContext erstellt außerdem den Persistor. Der Persistor ist ein Thread, der in regelmäßigen Abständen das Warehouse speichert.

```

public void run() {
18     var warehouse = this.ctx.getWarehouse();
    warehouse.load();
20     while (!this.ctx.isCloseRequested().get()) {
        System.out.println("Saving ...");
22
        try {
24             warehouse.save();
            System.out.println("Save completed :)");
26             Thread.sleep(SAVEINTERVAL);
        } catch (IOException | InterruptedException e1) {
28             e1.printStackTrace();
        }
30     }
}

```

../src/main/java/vs/chat/server/persistence/Persistor.java

```

66     public synchronized void save() throws IOException {
        File tempFile = File.createTempFile(this.saveFileName, ".tmp");
68         try (var stream = new FileOutputStream(tempFile)) {
            var outputStream = new ObjectOutputStream(stream);
70             outputStream.writeObject(this.warehouse);
        }
    }

```

```

72     outputStream.writeObject(this.packetIds);
    }
    Files.move(Paths.get(tempFile.getPath()), Paths.get(new File(
74     this.saveFileName + ".dat").getPath()),
        StandardCopyOption.ATOMIC_MOVE);
    }

```

../src/main/java/vs/chat/server/warehouse/Warehouse.java

Alle Entitäten und Pakete haben eine UUID (Universally Unique Identifier) um diese zu unterscheiden. Verweise auf andere Entitäten (z. B. ein Chat hat mehrere Nutzer) werden ähnlich zu Fremdschlüsseln in relationalen Datenbanken umgesetzt. Die Entität speichert nur die UUID der Verknüpfung und nicht direkt die Information. Dies erlaubt eine feinere Synchronisation und reduziert mögliche Konfliktsituationen zwischen verschiedenen Nodes. Eingesetzt werden UUIDv4, die pseudozufällig erstellt werden. Dadurch sind zwar theoretisch Konflikte möglich, jedoch in Praxis sehr unwahrscheinlich.

Außerdem wird der Broadcaster erstellt, der die Verbindungen zu anderen Nodes hält und empfangene Nachrichten an diese verteilt.

```

14 public NodeBroadcaster(final ServerContext context, final
    NodeConfig... configs) {
    for(var config: configs) {
16         var conn = new NodeConnection(config.getAddress(), config.
            getPort(), context);
            conn.start();
18         nodes.add(conn);
    }
20 }

22 public void send(final Packet packet) {
    System.out.println("Broadcasting: " + packet);
24     for (var node : nodes) {
        node.send(packet);
26     }
}

```

../src/main/java/vs/chat/server/node/NodeBroadcaster.java

Weitere Variablen sind isCloseRequested, die die endlosSSchleifen aller Threads steuert und connections, welche alle Connections zu Clients, die direkt zu dieser Node verbunden sind, hält.

Der Nodebroadcaster erstellt bei Instanziierung je node einen eigenen Thread, der sich um das Senden und das neu verbinden kümmert. Nachrichten, die an eine Node gesendet werden soll werden sollen werden vom Broadcaster in die Queue geschrieben und die NodeConnection wird durch einen Semaphore aufgeweckt. Die NodeConnection versucht eine Nachricht zu senden. Scheitert das Senden wird von einer Verbindungstrennung ausgegangen und die Verbindung wird neu verbunden.

```

38     runSemaphore.acquire();
    try {
        if (this.currentSocket != null && out != null) {
40         var packet = this.sendQueue.peek();
            if (packet != null) {
42             out.writeObject(packet);

```

```

44         out.flush();
           this.sendQueue.remove();
46     }
    } catch (IOException e) {
48         this.reconnect();
           this.runSemaphore.release();
50     }

```

../src/main/java/vs/chat/server/node/NodeConnection.java

Zusätzlich besitzt die NodeConnection jeweils einen HeartBeat-Thread. Dieser Thread sendet regelmäßig einen Ping, um zu testen, ob die Verbindung noch steht.

```

16 public void run() {
    while (!this.isCloseRequested) {
18         try {
           out.send(new NoOpPacket());
           Thread.sleep(BEATRATE);
20         } catch (InterruptedException e) {
           e.printStackTrace();
22         }
24     }
    }

```

../src/main/java/vs/chat/server/node/NodeHeartBeatThread.java

Nachdem nun alle initialisierungs Vorgänge abgeschlossen sind, kann der Server-Socket erstellt werden und Clients akzeptiert werden. Der Hauptserver-Thread ist dabei nur zuständig neue Verbindungen entgegenzunehmen. Für jede Verbindung wird ein ConnectionHandler-Thread erstellt, der sämtliche Nachrichten des Clients verarbeitet.

```

42     try (var socket = new ServerSocket(PORT)) {
        while (!this.context.isCloseRequested().get()) {
44             try {
                 var clientSocket = socket.accept();
                 var outputStream = new ObjectOutputStream(clientSocket.
getOutputStream());
46                 var inputStream = new ObjectInputStream(clientSocket.
getInputStream());

48                 var connectionHandler = new ConnectionHandler(
clientSocket, this.context, outputStream,
inputStream);
50                 this.context.getConnections().add(connectionHandler);
connectionHandler.start();
52             } catch (IOException e) {
                 e.printStackTrace();
54             }
        }
56         System.out.println("Stopping Server...");
           this.context.close();
58     } catch (IOException | InterruptedException e) {
           e.printStackTrace();
60     }

```

../src/main/java/vs/chat/server/Server.java

Nachrichten zwischen Servern und Clients werden als Pakete ausgetauscht. Der Handler versucht dabei ein Paket vom Client zu lesen. Die Filter prüfen nun, ob das Paket gehandelt werden darf (und nach dem Verarbeiten dessen werden diese aktualisiert).

```

48         var object = inputStream.readObject();
        var packet = (Packet) object;
        System.out.println("Read Packet: " + packet.getClass().
getSimpleName());
50         var canActivate = this.context.getFilters().stream()
            .allMatch(f -> f.canActivate(packet, this.context, this
));
52         if (canActivate) {
            this.handlePacket(packet);
        }
    }
}

```

../src/main/java/vs/chat/server/ConnectionHandler.java

Anschließend werden die passenden Listener gesucht und diese mit dem Paket aufgerufen.

```

    }
62     private void handlePacket(final Packet packet) throws IOException
    {
64         if (packet instanceof LogoutPacket) {
            this.pushTo(new LogoutSuccessPacket());
66             this.closeRequested = true;
            return;
68         }
        for (var listener : this.context.getListeners()) {
69             try {
70                 var methods = listener.getClass().getDeclaredMethods();
71                 for (var method : methods) {
72                     if (method.getName().equals("next") && !method.
isSynthetic()) {
73                         var packetType = method.getParameters()[0];
74                         if (packetType.getType().isAssignableFrom(packet.
getClass())) {
75                             var retu = (Packet) method.invoke(listener, packet,
76                             this.context, this);
77                             this.pushTo(retu);
78                         }
79                     }
80                 }
81             } catch (SecurityException | IllegalArgumentException |
IllegalAccessException
82                 | InvocationTargetException e) {
83                 e.printStackTrace();
84             }
85         }
86     }
}

```

../src/main/java/vs/chat/server/ConnectionHandler.java

Filter:

- PacketIdFilter

Der PacketId-Filter testet, ob ein Paket mit der Id bereits gesehen wurde. Nur wenn die Id neu ist darf das Paket gehandelt werden, um rekursive Broadcasts zu vermeiden. Bereits gesehene Pakete werden im Warehouse

mitgespeichert. Im seltenen Fall, dass die Node genau zwischen den Listenern und dem Aktualisieren der Filter abstürzt kann es vorkommen, dass die gespeicherten Paket-ids nicht konsistent zum Nutzdatenbestand sind. Hier könnte ein Transaktionsprotokoll implementiert werden. Da aber die Wahrscheinlichkeit dieses Fehlers äußerst gering ist wird hier darauf verzichtet.

```

10  @Override
    public boolean canActivate(final Packet packet, final
        ServerContext context, final ConnectionHandler handler) {
        return !context.getWarehouse().knowsPacket(packet.getId());
12  }

14  @Override
    public void postHandle(final Packet packet, final
        ServerContext context, final ConnectionHandler handler) {
16      context.getWarehouse().savePacket(packet.getId());
    }

```

../src/main/java/vs/chat/server/filter/PacketIdFilter.java

Listener:

- BaseEntityBroadcastListener

Der Listener behandelt BaseEntityBroadcastPakete, die ausgestrahlt werden, sobald ein neuer Nutzer, ein neuer Chat oder eine neue Nachricht erstellt wird. Die empfangene Entität wird in das Warehouse aufgenommen und weiter gesendet, falls es dieser Node neu war.

```

22      var entity = packet.getBaseEntity();
24      var exists = context.getWarehouse().get(entity.getType()).
        containsKey(entity.getId());
        if (!exists) {
26          context.getWarehouse().get(entity.getType()).put(entity.
            getId(), entity);
            context.getBroadcaster().send(packet);
        }

```

../src/main/java/vs/chat/server/listener/BaseEntityBroadcastListener.java

Nachdem ein Chat erstellt wurde müssen die Clients, die an diesem Chat teilnehmen informiert werden. Da jede Node nur die direkt zu ihr verbundenen Clients kennt, muss jede Node prüfen ob sie einen teilnehmenden Client kennt und diesen informieren. Ähnliches gilt für neue Nutzer.

```

34      var chat = (Chat) entity;
        distributionUser = chat.getUsers();
    } else if (entity instanceof Message) {
36      var message = (Message) entity;
        distributionUser = ((Chat) context.getWarehouse().get(
            WarehouseResourceType.CHATS)
            .get(message.getTarget())).getUsers();
38      } else if (entity instanceof User) {
        distributionUser = context.getWarehouse().get(
40      WarehouseResourceType.USERS).keySet();
        distributionPacket = new BaseEntityBroadcastPacket(
42      new User(entity.getId(), ((User) entity).
            getUsername()));
    }

```

```

    }
44
    for (var user : distributionUser) {
46        var localConnection = context.getConnectionForUserId(
            user);
48        if (localConnection.isPresent()) {
            localConnection.get().pushTo(distributionPacket);
50        }
    }

```

../src/main/java/vs/chat/server/listener/BaseEntityBroadcastListener.java

- CreateChatListener

Dieser Listener erstellt Chats anhand von einem CreateChatPacket. Sofern das Paket von einem Nutzer kommt, wird ein neuer Chat mit allen Teilnehmern und dem Absender erstellt und weiter verteilt.

```

    packet.getUsers().add(currentUser.get());
24
    var knownUsers = context.getWarehouse().get(
        WarehouseResourceType.USERS);
26    var filteredUsers = packet.getUsers().stream().filter(u ->
        knownUsers.containsKey(u)).toArray(UUID[]::new);

28
    Chat newChat = new Chat(packet.getName(), filteredUsers);
30
    context.getWarehouse().get(WarehouseResourceType.CHATS).
        put(newChat.getId(), newChat);
32
    var broadcastPacket = new BaseEntityBroadcastPacket(
        newChat);
34    context.getBroadcaster().send(broadcastPacket);

36    for (var user: filteredUsers) {
        var localConnection = context.getConnectionForUserId(
            user);
38        if (localConnection.isPresent()) {
            localConnection.get().pushTo(broadcastPacket);
40        }
    }

```

../src/main/java/vs/chat/server/listener/CreateChatListener.java

- GetMessagesListener

Mithilfe eines GetMessagePackets können alle Nachrichten abgefragt werden, die in einem Chat gesendet wurden.

```

20    var currentUser = handler.getConnectedToUserId();
    if (currentUser.isEmpty())
22        return null;

24    var chatId = packet.getChatId();
    var chat = (Chat) context.getWarehouse().get(
        WarehouseResourceType.CHATS).get(chatId);
26    if (!chat.getUsers().contains(currentUser.get()))
        return null;

28
    var messages = context.getWarehouse().get(
        WarehouseResourceType.MESSAGES).values().stream()

```

```

30|         .map(m -> (Message) m).filter(m -> m.getTarget().
    equals(chatId)).collect(Collectors.toSet());
32|     return new GetMessagesResponsePacket(chatId, messages);
    ../src/main/java/vs/chat/server/listener/GetMessagesListener.java

```

- KeyExchangeListener

Dieser Listener leitet KeyExchangePakete von einem Client an andere Clients weiter (gegebenenfalls über andere Nodes).

```

16|     var currentUser = handler.getConnectedToUserId();
    if (currentUser.isPresent()) {
18|         packet.setOrigin(handler.getConnectedToUserId().get());
    }
    if (null == packet.getOrigin()) {
20|         return null;
    }
22|
    var localConnection = context.getConnectionForUserId(
    packet.getTarget());
24|    if (localConnection.isPresent()) {
        localConnection.get().pushTo(packet);
26|    }
    context.getBroadcaster().send(packet);
    ../src/main/java/vs/chat/server/listener/KeyExchangeListener.java

```

- LoginListener

Der LoginListener kümmert sich um die Authentifizierung eines Clients. Er prüft, ob ein Nutzer besteht und falls ja wird das Passwort geprüft. Außerdem wird die Information der Connection gesetzt, zu welchem Client sie verbunden ist, um ein gezieltes Senden zu ermöglichen (wie z. B. beim KeyExchange oder bei Messages).

```

26|     var res = context.getWarehouse().get(WarehouseResourceType.
    USERS).values().stream()
        .filter(u -> ((PasswordUser) u).getUsername().equals(
    packet.getUsername()).findFirst();
    if (res.isPresent()) {
28|         if (!((PasswordUser) res.get()).hasPassword(packet.
    getPassword())) {
            return new NoOpPacket();
30|         } else {
            var id = res.get().getId();
32|            handler.setConnectedToUserId(id);
            System.out.println("connected id: " + id);
34|        }
    }
    ../src/main/java/vs/chat/server/listener/LoginListener.java

```

Existiert noch kein Nutzer, wird ein passender Nutzer erstellt.

```

38|     var user = new PasswordUser();
    var id = user.getId();
    user.setUsername(packet.getUsername());
40|    user.setPassword(packet.getPassword());

42|    context.getWarehouse().get(WarehouseResourceType.USERS).
    put(id, user);

```



```

44      System.out.println("created user with id:" + id);

      var broadcastPacket = new BaseEntityBroadcastPacket(user
46      );

      var b = new BaseEntityBroadcastPacket(new User(user.
      getId(), user.getUsername()));
48      for (var others : context.getWarehouse().get(
      WarehouseResourceType.USERS).keySet()) {
          var localConnection = context.getConnectionForUserId(
      others);
50          if (localConnection.isPresent()) {
              localConnection.get().pushTo(b);
52          }
      }

54      context.getBroadcaster().send(broadcastPacket);
56      handler.setConnectedToUserId(id);

../src/main/java/vs/chat/server/listener/LoginListener.java

```

Anschließend wird der Client auf den neuesten Stand gebracht indem ein LoginSyncPacket an den Client gesendet wird. Dieses enthält die User-Id des aktuellen Nutzers, die anderen registrierten Nutzer und alle Chats, an dem der Client teilnimmt.

```

      var chats = context.getWarehouse().get(
      WarehouseResourceType.CHATS).values().stream().map(chat ->
      (Chat) chat)
60      .filter(chat -> chat.getUsers().contains(handler.
      getConnectedToUserId().get()))
      .collect(Collectors.toSet());
62      var users = context.getWarehouse().get(
      WarehouseResourceType.USERS).values().stream().map(user ->
      {
          var u = (PasswordUser) user;
64          return new User(u.getId(), u.getUsername());
      }).collect(Collectors.toSet());

../src/main/java/vs/chat/server/listener/LoginListener.java

```

- MessageListener

Messages, die vom Client an einen Chat gesendet werden, werden von diesem Listener bearbeitet. Der Listener kümmert sich dabei auch um die Verteilung der Nachrichten an alle anderen Chatteilnehmer.

```

24      Message newMessage = new Message(handler.
      getConnectedToUserId().get());
      newMessage.setTarget(packet.getTarget());
26      newMessage.setContent(packet.getContent());

      System.out.println("found a new message with target " +
      newMessage.getTarget());

28

30      var correspondingChat = (Chat) context.getWarehouse().get(
      WarehouseResourceType.CHATS)
          .get(newMessage.getTarget());
32      if (correspondingChat == null) {
          return null;
34      }

```

```

context.getWarehouse().get(WarehouseResourceType.MESSAGES)
.put(newMessage.getId(), newMessage);
36
var broadcastPacket = new BaseEntityBroadcastPacket(
newMessage);
38
for (var user : correspondingChat.getUsers()) {
    var localConnection = context.getConnectionForUserId(
user);
40
    if (localConnection.isPresent()) {
        localConnection.get().pushTo(broadcastPacket);
42
    }
}
44
context.getBroadcaster().send(broadcastPacket);

```

../src/main/java/vs/chat/server/listener/MessageListener.java

- NodeSyncListener

Wie in Fehlerbehandlung beschrieben müssen Nodes auf dem neuesten Stand gezogen werden, falls diese ausgefallen waren. Bei einem Reconnect wird ein NodeSyncPacket mit den aktuellen Informationen an die neu startende Node gesendet. Dieser Listener verarbeitet diese Pakete indem er prüft ob eine Änderung vorliegt und wenn ja diese übernimmt und broadcastet.

```

16
var needsBroadcast = false;
for (var id : packet.packetIds) {
18
    if (!context.getWarehouse().knowsPacket(id)) {
        context.getWarehouse().savePacket(id);
20
        needsBroadcast = true;
    }
22
}
for (var type : WarehouseResourceType.values()) {
24
    for (var entry : packet.warehouse.get(type).entrySet()) {
        if (null == context.getWarehouse().get(type).get(entry
.getKey())) {
26
            context.getWarehouse().get(type).put(entry.getKey(),
entry.getValue()); //BaseEntityBroadcast
            needsBroadcast = true;
28
        }
    }
30
}
32
if (needsBroadcast) {
    context.getBroadcaster().send(packet);
34
}

```

../src/main/java/vs/chat/server/listener/NodeSyncListener.java

Theoretisch kann es sein, dass ein Nutzer sich anmeldet bevor die Node ihre Synchronisation abgeschlossen hat. Dieser Fehlerfall wird aber nicht weiter behandelt, da die Synchronisationszeit und die Ausfallwahrscheinlichkeit einer Node als zu gering eingestuft wird.

3.2 Client

API-Funktionen

- Funktion - login

Bei der Funktion login werden der Benutzername und das Passwort des Benutzers entgegengenommen. Diese werden als neues LoginPacket an den Server geschickt. Dort wird unter anderem überprüft, ob es sich um einen neuen Benutzer handelt (Es wird ein neues angelegt) oder es ein bereits existierender Benutzer ist (Passwort wird überprüft). An dieser Stelle wartet der Client auf ein LoginSyncPacket (Login war erfolgreich). Danach werden die Attribute (userId, chats, contacts) abgespeichert. Für die Verschlüsselung wird auch noch die Keyfile geladen, in der die Schlüssel der eigenen Chats gespeichert sind. Sollte es noch keine Keyfile geben, wird eine neue erzeugt. Diese Datei ist ähnlich wie die Warehouse-Datei aufgebaut (hier wird über die ChatId, die Schlüssel geladen). Abschließend wird aus den beiden hardgecodeten public Keys noch ein private Key generiert, der später für den Diffie Hellman Schlüsseltausch notwendig ist.

```

58     }
60
61     public BufferedReader getUserIn() {
62         return this.userIn;
63     }
64
65     public void login(String username, String password) throws
66         LoginException {
67         try {
68             LoginPacket loginPacket = new LoginPacket(username
69             , password);
70
71             this.networkOut.writeObject(loginPacket);
72             this.networkOut.flush();
73
74             Object response = this.networkIn.readObject();
75
76             if (response instanceof NoOpPacket) {
77                 throw new LoginException();
78             }
79
80             LoginSyncPacket loginSyncPacket = (LoginSyncPacket
81             ) response;
82
83             this.lastUsername = username;
84             this.lastPassword = password;
85
86             this.userId = loginSyncPacket.userId;
87             this.chats = loginSyncPacket.chats;
88             this.contacts = loginSyncPacket.users;
89
90             this.keyfile = new Keyfile(username);
91             keyfile.load();

```

../src/main/java/vs/chat/client/ClientApiImpl.java

- Funktion -generatePrivateKey

Mit dieser Funktion wird ein 128 private Key aus den beiden public Keys n und g generiert. Der private Key wird für die Verschlüsselung verwendet.

```

96         return this.userId;

```

```

98 |     public Set<Chat> getChats() {
100 |         return this.chats;
    |     }
102 |     private void generatePrivateKey() {
        SecureRandom random = new SecureRandom();
    |
    |     ../../src/main/java/vs/chat/client/ClientApiImpl.java

```

- Funktion - exchangeKeys

Um den Diffie-Hellman Schlüsseltausch zu starten muss die exchangeKeys Methode vom Client aufgerufen werden. Dabei erstellt die Methode das erste KeyExchangePacket, welches mit den erforderlichen Informationen ausgestattet wird. Nachdem das Paket gesendet wurde, haben die Teilnehmer 10 Sekunden Zeit, um den Schlüsseltausch durchzuführen. Wenn dies innerhalb der angegebenen Zeit nicht geschieht gehen wir von einem Fehler ausgegangen und es wird die Error Handling Methode onTimeout aufgerufen.

```

154 |         if (user != null) {
156 |             return user.getUsername();
        }
158 |         return null;
    |     }
160 |     public void exchangeKeys(String chatName, List<UUID>
        userIds, OnTimeout onTimeout) throws IOException,
        InterruptedException {
162 |         userIds.add(0, this.userId);
        KeyExchangePacket keyExchangePacket = new
        KeyExchangePacket(
164 |             this.nextKey,
                1,
166 |             this.userId,
                userIds,
168 |             chatName,
                userIds.get(1));
170 |
172 |         this.networkOut.writeObject(keyExchangePacket);
        this.networkOut.flush();
174 |         this.creatingChat = true;
    |
    |     ../../src/main/java/vs/chat/client/ClientApiImpl.java

```

- Funktion - createChat

Für einen neuen Chat werden 2 Parameter benötigt: Name des Chats und Liste mit Usern. Um einen neuen Chat zu erstellen, wird ein CreateChat-Packet mit dem Chatnamen und einem Array der UserIds an den Server geschickt.

```

154 |         if (this.creatingChat) {
156 |             this.creatingChat = false;
                onTimeout.run();
158 |         }
    |     }
160 |

```

```
private void createChat(String chatName, List<UUID>
userIds) throws IOException {
```

```
../src/main/java/vs/chat/client/ClientApiImpl.java
```

- Funktion - sendMessage

Hier wird die eigentliche Nachricht und eine ChatId entgegengenommen. Diese werden dann in ein MessagePacket an den Server geschickt.

```
164         chatUsers = userIds.toArray(chatUsers);
        CreateChatPacket createChatPacket = new
CreateChatPacket(chatName, chatUsers);
166         this.networkOut.writeObject(createChatPacket);
168         this.networkOut.flush();
    }
```

```
../src/main/java/vs/chat/client/ClientApiImpl.java
```

- Funktion - Verschlüsselung (encryptAES, decryptAES, setKey)

Für die symmetrische Verschlüsselung der Nachrichten wird der AES-Algorithmus benutzt. Hierfür wird vor jedem Senden die Methode encryptAES und nach jeder empfangener Nachricht decryptAES aufgerufen. Für die Verschlüsselung wird zunächst der Schlüssel in das richtige Format (SecretKeySpec), durch die Funktion setKey gebracht. Anschließend wird eine Instanz der Klasse Cipher erzeugt und mit dem Verschlüsselungsmodus und dem Key initialisiert. Abschließend wird der eigentlich Text verschlüsselt und in einem String zurückgegeben.

Die Entschlüsselung ist fast identisch zur Verschlüsselung. Hier wird die Instanz in dem Entschlüsselungsmodus initialisiert.

```
172         String chatKey = loadKey(chatId).toString();
        MessagePacket messagePacket = new MessagePacket(chatId
, encryptAES(chatKey, message));
174         this.networkOut.writeObject(messagePacket);
        this.networkOut.flush();
176     }

178     public String encryptAES(String key, String message) {
        try {
180         Cipher cipher = Cipher.getInstance("AES/ECB/
PKCS5Padding");
        cipher.init(Cipher.ENCRYPT_MODE, setKey(key));
182         return Base64.getEncoder().encodeToString(cipher.
doFinal(message.getBytes(StandardCharsets.UTF_8)));
        } catch (Exception e) {
184             e.printStackTrace();
        }
186         return null;
    }

188     public String decryptAES(String key, String cifre) {
190         try {
        Cipher cipher = Cipher.getInstance("AES/ECB/
PKCS5PADDING");
```

```
../src/main/java/vs/chat/client/ClientApiImpl.java
```

- Funktion - Keyfile (addKey, loadKey, deleteKey) addKey
Mit diesen Funktionen wird auf die Keyfile des Benutzers zugegriffen. Die Keyfile besteht aus PrivateKeyEntitys. Der Schlüssel ist die chatId. Wird ein neuer Key gespeichert, wird eine neue PrivateKeyEntity mit der chatId und dem Schlüssel in der keyfile abgespeichert.

addKey

Müssen Nachrichten angezeigt werden, muss der Schlüssel aus der Datei für die chatId ausgelesen werden.

loadKey

Sollte ein Chat irgendwann gelöscht werden, kann der die PrivateKeyEntity über die chatID entfernt werden.

deleteKey

//TODO hier fehlt noch text + das listing ist verschoben

```

                return new String(cipher.doFinal(Base64.getDecoder
210            ().decode(cifre.getBytes(StandardCharsets.UTF_8))));
                } catch (Exception e) {
212            e.printStackTrace();
                }
214        }

216        //Formatting key to SerectKeySpec
        public SecretKeySpec setKey(String myKey) {
218            MessageDigest sha;
            byte[] key;
220            try {
                key = myKey.getBytes(StandardCharsets.UTF_8);
222                sha = MessageDigest.getInstance("SHA-256");
                key = sha.digest(key);
224                key = Arrays.copyOf(key, KEY_BYTELENGTH);
                return new SecretKeySpec(key, "AES");
226            } catch (NoSuchAlgorithmException e) {
                e.printStackTrace();
228            }
            return null;
230        }

232        public void addKey(UUID chatId, BigInteger key) {
            var pKEntry = new PrivateKeyEntity(chatId, key);
234            this.keyfile.get(KeyfileResourceType.PRIVATEKEY).put(
                chatId, pKEntry);
            try {
236                keyfile.save();
            } catch (IOException e1){
238                e1.printStackTrace();
            }
240        }

242        public BigInteger loadKey(UUID chatId) {
            var res = keyfile.get(KeyfileResourceType.PRIVATEKEY).
                values().stream()
244                .filter(u -> ((PrivateKeyEntity) u).equals(
                    chatId)).findFirst();
            if (res.isPresent()) {
246                PrivateKeyEntity pke = (PrivateKeyEntity) keyfile.
                    get(KeyfileResourceType.PRIVATEKEY).get(chatId);
                try {

```

```

248         keyfile.save();
        } catch (IOException e1){
250             e1.printStackTrace();
        }
252         return pke.getPrivateKey();
    }

```

../src/main/java/vs/chat/client/ClientApiImpl.java

- Funktion - exit

Bei dieser Methode wird ein LogoutPacket an den Server geschickt. Der User wird hier noch nicht ausgeloggt!

```

248         try {
            keyfile.save();
250         } catch (IOException e1){
            e1.printStackTrace();
252         }

```

../src/main/java/vs/chat/client/ClientApiImpl.java

- PacketListener-Thread

Dieser Thread läuft die gesamte Zeit im Hintergrund und nimmt alle (außer LoginSynPacket) Pakete, die vom Server an den Client geschickt werden, auf und verarbeitet sie. Zuerst wird überprüft, ob es sich um ein BaseEntityBroadcastPacket handelt. Anschließend gibt es verschiedene Möglichkeiten:

1. Es wurde ein Chat erstellt, in dem der User enthalten ist. Hier wird der Key in der Keyfile abgespeichert und der Chat zu den Chats des Users hinzugefügt.
2. Es wurde eine Nachricht an den User geschrieben. Hier wird der Schlüssel aus der Keyfile geladen und die Nachricht damit entschlüsselt.
3. Es hat sich ein neuer User angemeldet. Dieser User muss den Usern des Clients hinzugefügt werden, damit auch Chats mit diesem erstellt werden können. (Diffie Hellman funktioniert nur, wenn alle Benutzer online sind.)

```

262         new Thread(new Runnable() {
            @Override
264             public void run() {
                while (true) {
266                     try {
                        Object packet = networkIn.readObject()
                    }
                    ;
268                     if (packet instanceof
                        BaseEntityBroadcastPacket) {
270                         BaseEntityBroadcastPacket base = (
                            BaseEntityBroadcastPacket) packet;
272
                        if (base.getBaseEntity()
                            instanceof Chat) {
274                             creatingChat = false;

```

```

276         getBaseEntity();
278         nextKey);
280         generatePrivateKey();
282         chats.add(newChat);
284         onCreateChat.run(newChat);
instanceof Message) {
    Message m = (Message) base.
getBaseEntity();
    Message d = new Message(m.
getOrigin(), m.getReceiveTime());
    d.setTarget(m.getTarget());
290    String chatKey = loadKey(m.
getTarget()).toString();
    d.setContent(decryptAES(
chatKey, m.getContent()));
}

../src/main/java/vs/chat/client/ClientApiImpl.java

```

Handelt es sich um ein `KeyExchangePacket`, wird überprüft, ob der Schlüsseltausch schon fertig ist, falls ja erstellt der Initiator des Schlüsseltauschs den Chat, falls nicht werden die nächsten Teilschlüssel berechnet und weitergeschickt.

```

294         } else if (base.getBaseEntity()
instanceof User) {
    User u = (User) base.
getBaseEntity();
    contacts.add(u);
    System.out.println("Added new
User " + u.getUsername() + " ");
    System.out.print("> ");
}
} else if (packet instanceof
KeyExchangePacket) {
    KeyExchangePacket keyExchangePacket
= (KeyExchangePacket) packet;
    List<UUID> participants =
keyExchangePacket.getParticipants();
    int currentRequests =
keyExchangePacket.getRequests();
    BigInteger currentContent =
keyExchangePacket.getContent();
    int targetRequests = participants.
size() * (participants.size() - 1);
    int userIndex = participants.
indexOf(userId);
    int rounds = currentRequests /
participants.size();
}

```



```

312|                                     if (keyExchangePacket.getInitiator
314|                                     () .equals(userId)) {
314|                                     nextKey = currentContent .
modPow(privateKey , n);
316|                                     } else {
316|                                     rounds++;
318|                                     }
318|                                     System.out.println("\n->
Exchanging keys round " + rounds);
320|                                     // check if package has to be
forwarded
322|                                     if (currentRequests <
targetRequests) {
324|                                     // forwards package to next
participant
KeyExchangePacket
newExchangePacket = new KeyExchangePacket(
326|                                     nextKey ,
keyExchangePacket .
getRequests() + 1,
328|                                     keyExchangePacket .
getInitiator() ,
keyExchangePacket .
getParticipants() ,
330|                                     keyExchangePacket .
getChatName() ,
participants .get((
userIndex + 1) % participants .size()));
332|                                     networkOut.writeObject(
newExchangePacket);
332|                                     networkOut.flush();
334|                                     }
334|                                     nextKey = currentContent.modPow(
privateKey , n);
336|                                     // check if participant has
finished key exchange
338|                                     if (keyExchangePacket.getInitiator
() .equals(userId)) {
338|                                     if (currentRequests == (
targetRequests - userIndex)) {

```

../src/main/java/vs/chat/client/ClientApiImpl.java

Wird ein GetMessagesResponsePacket erhalten, wurde an den Server zuvor die Anfrage gestellt, dass die Chat Historie geladen werden soll. Dieses Packet enthält alle Nachrichten, die in dem geladenen Chat bereits geschrieben wurden. Dafür wird zunächst der passende Schlüssel aus der Keyfile geladen und danach alle Nachrichten entschlüsselt. Diese Nachrichten können dann im Chat angezeigt werden.

```

342|                                     participants.remove(0);
createChat(
keyExchangePacket.getChatName() , participants);
}

```

```

344         } else if (currentRequests == (
targetRequests - (participants.size() - userIndex))) {
        System.out.println(
346         getUsernameFromId(userId) + " → " + nextKey);
        }

348         } else if (packet instanceof
GetMessagesResponsePacket) {
        Set<Message> messages = ((
GetMessagesResponsePacket) packet).getMessages();
350         Set<Message> decrypted = new
TreeSet<>();

352         for (Message m: messages) {
        Message d = new Message(m.
getOrigin(), m.getReceiveTime());
354         d.setTarget(m.getTarget());

```

../src/main/java/vs/chat/client/ClientApiImpl.java

Abschließend wird geprüft, ob es ein LogoutSuccessPacket ist. Dieses Paket wird vom Server geschickt, wenn der User die exit Methode aufgerufen hat und erfolgreich vom Server ausgeloggt wurde. Anschließend wird der Client heruntergefahren.

```

356         } catch (EOFException | SocketException e)
{
        reconnect();
358         } catch (IOException |
ClassNotFoundException e) {
        e.printStackTrace();
360         }
        }

362         try {
364         socket.close();

```

../src/main/java/vs/chat/client/ClientApiImpl.java

Verwendung von Emojis:

Wie bereits in der Doku erwähnt nutzen wir für die Darstellung von Emojis Unicode.

```

622     public JPanel renderEmojiPanel() {
        float fontsize = 32f;
        JPanel emojiSelection = new JPanel(new GridLayout(0,
624     5));
        String[] unicodeemoji = {"\uD83D\uDE04", "\uD83D\uDE02",
        "\uD83D\uDE43", "\uD83D\uDE09", "\uD83D\uDE07", "\uD83D\uDE18",
        "\uD83D\uDE0B", "\uD83E\uDD14", "\uD83D\uDE0F", "\uD83D\uDE37"};
        for (int i = 1; i <= unicodeemoji.length; i++) {
626         JLabel label = new JLabel(unicodeemoji[i - 1]);
        label.setHorizontalAlignment(JLabel.CENTER);
628         label.setFont(label.getFont().deriveFont(fontsize));
        }
        label.addMouseListener(new
SelectEmojiMouseListener(unicodeemoji[i - 1]));

```

```
630 |         emojiSelection.add(label);  
    |     }
```

../src/main/java/vs/chat/client/UI/ClientGUI.java

Die im String Array unicodeemoji erhaltenen Unicode Zeichen werden jeweils in einem JLabel hinzugefügt und auf ein Panel gesetzt. Jedes JLabel bekommt den selben MouseListener zugewiesen, der als Übergabeparameter das entsprechende Unicodezeichen erhält. Im EmojiMouseListener wird mit der Funktion append() die JTextArea um das jeweils übergebene Unicodezeichen erweitert.

```
90 |     }  
92 |     @Override  
    |     public void mouseExited(MouseEvent e) {  
94 |     }  
96 | }  
98 | private class SelectEmojiMouseListener implements  
    | MouseListener {  
    |     String unicode;  
100 |  
    |     public SelectEmojiMouseListener(String unicode) {  
102 |         this.unicode = unicode;  
    |     }  
104 |  
    |     @Override  
106 |     public void mouseClicked(MouseEvent mouseEvent) {  
    |         messageInput.append(unicode);  
108 |     }
```

../src/main/java/vs/chat/client/UI/ClientGUI.java

Durch das Erweitern oder Ändern des unicodeemoji Arrays können die Emoticons getauscht oder beliebig erweitert werden. Weiter Anpassungen sind nicht nötig.