

# Dokumentation

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

20. April 2020

[GitHub](#) · [Video](#)

## Inhaltsverzeichnis

<b>0</b>	<b>Quick Start Guide</b>	<b>2</b>
<b>1</b>	<b>Basisanforderungen</b>	<b>2</b>
1.1	Chatfunktionalität . . . . .	2
1.2	Clientfunktionalitäten . . . . .	3
1.3	Fehlerbehandlung . . . . .	4
1.3.1	Client . . . . .	4
1.3.2	Server . . . . .	4
<b>2</b>	<b>Erweiterungen</b>	<b>5</b>
2.1	Grafische Benutzeroberfläche . . . . .	5
2.2	Verwendung von Emojis . . . . .	6
2.3	Gruppenchats & Mehrere Chatverläufe pro Nutzer . . . . .	6
2.4	Persistentes Speichern der Chatverläufe . . . . .	6
2.5	Verschlüsselte Übertragung der Chat-Nachrichten . . . . .	7
2.6	Verschlüsselte serverseitige Speicherung der Chats . . . . .	8
<b>3</b>	<b>Codewalkthrough</b>	<b>9</b>
3.1	Server . . . . .	9
3.2	Client . . . . .	19

## 0 Quick Start Guide

Zunächst sollte sichergestellt werden, dass als Java-Version 11 eingestellt ist. Das Projekt kann als Maven-Projekt importiert werden. Danach kann die erste Node gestartet werden. Dafür muss eine Instanz der Klasse `Server-Bootstrapper` erzeugt werden und wenn es erwünscht ist, ein anderer Port (standardmäßig 9876) eingestellt werden. Außerdem können mit der Klasse `NodeConfig` weitere Nodes angegeben werden, zu denen eine Verbindung gehalten werden soll. Wurde mindestens eine Node erfolgreich hochgefahren, können beliebig viele Clients gestartet werden. Bei der Klasse `Client` können noch Programmargumente in der Form `java xy.jar node1.net:1234 node2.de:9877` mitgegeben werden. Stehen in den Programmargumenten die Adressen der Nodes, werden diese ausgewählt, oder es wird standardmäßig `localhost` ausgewählt. Sind 2 Clients online, kann ein Chat erstellt werden.

## 1 Basisanforderungen

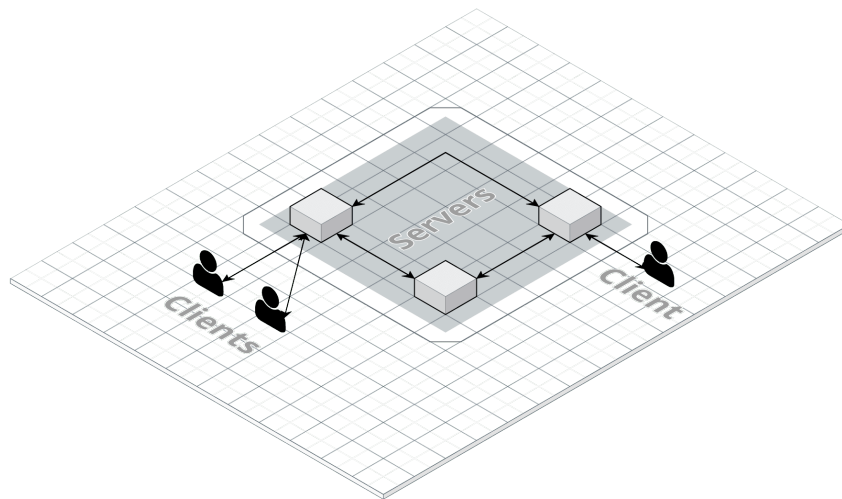


Abbildung 1: Architektur

### 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 inklusive der verpassten Nachrichten zu. Der Server vergibt jeder eintreffenden Nachricht einen Timestamp, um zu dokumentieren, wann sie erstmalig eingetroffen ist. Anhand des Timestamps werden die Nachrichten sortiert, damit der Client die korrekte Reihenfolge der Nachrichten darstellen kann.

Wenn ein Client eine Nachricht versenden möchte, wird das Nachrichtenpaket an die Node gesendet, mit der er verbunden ist. Die Node kümmert sich im Hintergrund darum, die Nachricht an den Zielclient zuzustellen. 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, um die zusätzliche Funktionalität bereitzustellen. Empfängt eine Node eine Nachricht, egal ob von einem Client oder von einer anderen Node, wird überprüft, ob die Nachricht für einen ihr bekannten Client bestimmt war. Wird ein Client gefunden, sendet die Node die Nachricht an den Zielclient.

## 1.2 Clientfunktionalitäten

*Jan Grübener, Troy Keßler, Patrick Mischka, Michael Angermeier*

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

### **/help:**

Diese Funktion gibt dem Nutzer einen Überblick über alle möglichen Funktionen, die er aufrufen kann. Alle Funktionen sind kurz beschrieben, sodass der Nutzer einen Überblick über die Funktionen erhält.

### **/chats:**

Bei einem Aufruf dieser Funktion werden alle Chats, die für den Nutzer zugänglich sind, angezeigt. Wenn Chats für den Nutzer verfügbar sind, werden diese in einer Übersicht mit Chatname und teilnehmenden Nutzen dargestellt. Sind noch keine Chats vorhanden, wird darauf hingewiesen.

### **/contacts:**

Wird `/contacts` aufgerufen, werden alle registrierten Nutzer angezeigt.

### **/createchat:**

Diese Funktion beginnt mit einer Aufforderung an den Nutzer, einen Chatnamen einzugeben. Danach wird die Anzahl der Teilnehmer für den Chat erfragt, wobei mindestens ein Teilnehmer im Chat enthalten sein muss. Im nächsten Schritt müssen alle Nutzernamen der Teilnehmer eingetragen werden. Jeder Nutzername wird auf seine Gültigkeit geprüft. Ist ein Nutzername ungültig, so wird darauf hingewiesen. Wurden alle drei Attribute (Chatname, Teilnehmeranzahl, Nutzername der Teilnehmer) erfolgreich eingegeben, wird ein neuer Chat erstellt.

### **/openchat:**

Will der Nutzer einen Chat öffnen, so muss er zuerst den Chatnamen eingeben. Ist der Chat vorhanden, so wird er geöffnet. Kann der Chat nicht geöffnet werden, so wird eine Meldung für den Nutzer ausgegeben. 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 Nutzer Nachrichten versenden und empfangen.

**/exit:**

Mithilfe dieser Funktion wird der Nutzer abgemeldet und Client beendet.

### 1.3 Fehlerbehandlung

*Matthias Vonend, Aaron Schweig, Troy Keffler*

Um Fehler und Datenverluste zu vermeiden, ist in dem Chatsystem sichergestellt, dass immer mindestens zwei Server (Nodes) **alle** Informationen besitzen. So kann während eines Nodeausfalls gewährleistet werden, dass eine Andere alle Aufgaben übernehmen kann.

#### 1.3.1 Client

Nachdem der Client eingeloggt ist, wartet er kontinuierlich auf neue Pakete vom Server. Stürzt eine Node ab, oder verliert der Client die Netzwerkverbindung, so versucht er sich neu zu verbinden. Dabei wird eine neue, zufällige Node ausgewählt. Kann der Client erfolgreich eine neue Verbindung aufbauen, meldet sich der Client mit den bestehenden Zugangsdaten an der Node an. Ist diese nicht verfügbar, so wird so lange eine neue Node ausgewählt, bis eine Verbindung zustande kommt. Im Regelfall findet der beschriebene Reconnect-Vorgang im Hintergrund statt.

#### 1.3.2 Server

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

##### **Nachricht des Clients wird nicht korrekt gesendet/empfangen**

In diesem Fall muss der Server davon ausgehen, dass die Verbindung zusammengebrochen ist. Diese wird im Anschluss vom Server beendet und wie bereits oben beschrieben versucht der Client erneut eine Verbindung aufzubauen. Ist eine neue Verbindung wiederhergestellt, werden alle für den Client relevanten Nachrichten zu diesem übertragen und der Client muss fehlende Informationen erneut an den Server übertragen.

##### **Nachrichten einer Nachbarnode werden nicht korrekt empfangen/gesendet**

Wie bereits im Kapitel 1.1 beschrieben, tauschen Nodes alle Nachrichten untereinander aus. Innerhalb des Chatsystems können mehrere Netzwerktopologien realisiert werden. Die ausfallsicherste Variante stellt dabei eine Mesh-Struktur dar. Diese ermöglicht es, auch bei einem Ausfall mehrerer Nodes, alle Nachrichten mit allen bekannten Nachbarnodes zu synchronisieren. Netzwerkpartitionierungen sind damit deutlich schwerer zu erreichen und auch mehrerer Nodeausfälle

können mithilfe der Mesh-Topologie kompensiert werden. Ebenfalls wird durch die Wahl der Write-All-Available Replikationsstrategie sichergestellt, dass alle bekannten Nachbarnodes komplett synchronisiert sind.

Tritt ein Fehler in der Verbindung zwischen verschiedenen Nodes auf, so muss die Node ähnlich wie bei einem Fehler im Client davon ausgehen, dass die Verbindung zusammengebrochen ist. Allerdings sind die Nodes hier selbst für eine Fehlerbehandlung zuständig und versucht eine neue Verbindung aufzubauen. Um sicherzustellen, dass keine konkurrierenden Schreibzugriffe auf den OutputStream stattfinden, werden Nachrichten in einer Queue aufbewahrt bevor sie an andere Nodes übertragen werden.

Werden die Nodes getrennt sind die Clients immer noch in der Lage Nachrichten an ihre jeweiligen Nodes zu schicken. Sobald eine Verbindung wieder aufgebaut wurde, synchronisieren sich die Nodes, um einen vollständigen Informationsstand wiederherzustellen. 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. Jede Node prüft dabei, ob sie alle Informationen des empfangenen Warehouses bereits besitzt und fügt geg. neue Informationen hinzu. Sofern die Node neue Informationen erhalten hat, broadcastet diese ihren neuen Stand an alle benachbarten Nodes, um diese auch auf den neuesten Stand zu bringen.

## 2 Erweiterungen

### 2.1 Grafische Benutzeroberfläche

*Jan Grübener, Patrick Mischka*

Bei dem Start eines Clients kann der Nutzer zwischen einem Commandline-Interface oder einer GUI wählen. In der GUI wird der Nutzer aufgefordert sich mittels Nutzernamen und Passwort zu identifizieren. Wenn dies erfolgreich war wird dem Nutzer eine Übersicht seiner Chats angezeigt. Am oberen Seitenbeginn ist der eigene Nutzername angezeigt, daneben ist ein Plus-Symbol, mit dem neue Chats erzeugt werden können. Hierbei gibt der Nutzer den neuen Chatnamen ein, als auch die Anzahl der zusätzlichen Chatteilnehmer. Folgend müssen die konkreten Chatteilnehmer genannt werden. Damit ein neuer Chat erstellt werden kann, müssen alle beteiligten Nutzer eingeloggt sein, damit der Schlüsselaustausch für die Verschlüsselung stattfinden kann. Ist das erfüllt, wird der Chat erstellt und der Nutzer gelangt wieder auf die Chatübersicht. Wenn mindestens einer der Nutzer nicht eingeloggt ist, wird dies dem Nutzer angezeigt. Er kann dann entweder erneut versuchen den Chat zu erstellen oder navigiert mithilfe des Pfeil-Icons zurück in die Chatübersicht. Durch Anklicken eines Chats in der Chatübersicht, wird der konkrete Chat geöffnet. Hierbei werden alle Nachrichten des Chats angezeigt. Am unteren Ende der Seite ist ein Texteingabefeld, mit dem der Nutzer auch Nachrichten in den Chat schreiben kann. Diese werden entweder durch eine Enter-Eingabe oder das Anklicken des Flugzeug-Icons verschickt. Nachrichten werden mit dem Nutzernamen des Erstellers für andere Nutzer angezeigt. Neben dem Chatnamen ist ein Pfeil-Icon, mithilfe dessen der Chat wieder geschlossen werden kann und sich wieder die Chat-Übersicht öffnet.

## 2.2 Verwendung von Emojis

*Jan Grübener, Patrick Mischka*

Bei der Verwendung von Emojis haben wir auf Unicode-Zeichen zurückgegriffen. Wir wollten die Nutzung von Icons als Bilddateien und den damit verbundenen Netzwerkverkehr und benötigten Speicherplatz so weit wie möglich vermeiden. Neben dem Texteingabefeld eines Chat ist ein Icon, das beim Anklicken eine Übersicht über die verwendbaren Emojis öffnet und diese bei erneutem klicken auch wieder schließt. Die Emojis in der Übersicht können angeklickt werden, damit sie in Form eines Unicode-Zeichens an die aktuelle Stelle im Texteingabefeld übernommen werden.

Auf die Implementierung wird im Codewalkthrough eingegangen.

## 2.3 Gruppenchats & Mehrere Chatverläufe pro Nutzer

*Matthias Vonend, Aaron Schweig, Troy Keßler*

Die Chatimplementierung innerhalb des Systems ist so aufgebaut, dass einige der Erweiterungen keine Codeänderungen erfordern. Ein Chat besteht aus einer Liste von Nutzern. Direkte Chats bedeuten somit nur, dass die Anzahl der Chatbeteiligten bei zwei liegt. Durch diese Form der Implementierung ist es möglich **Gruppenchats** einfach darzustellen. Die Liste der Chatteilnehmer kann theoretisch beliebig lang werden. Serverseitig wird dann nur eine längere Liste durchlaufen, wenn es darum geht, die Nachrichten zu verteilen. So stellen Gruppenchats eigentlich die natürliche Version dar und Chats zwischen zwei Nutzern sind nichts anderes, als ein Sonderfall eines Gruppenchats.

Möchte nun ein Nutzer **mehrere Chats** führen, so ist das, dank der bestehenden Chatimplementierung ohne Probleme möglich. Es kann abgefragt werden, in welchem Chat die entsprechende Nutzer-Id hinterlegt ist. Es kann also sein, dass eine Nutzer-Id sowohl in *Chat A* als auch in *Chat B* als Teilnehmer hinterlegt ist. Innerhalb des Clients können dann die Chats für den aktuellen Nutzer geladen werden und es können auch verschiedene Chats geöffnet und verwendet werden.

## 2.4 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 das Warehouse als Datei gespeichert werden. Bei einem Node-Start wird das Warehouse wieder geladen. Existiert noch kein Speicherstand, wird die Node mit einem leeren Warehouse initialisiert. Während die Node ausgefallen war, können andere Nodes neue Informationen erhalten haben. Sobald die Node wieder verfügbar ist, müssen andere Nodes dies bemerken und die ausgefallene Node mit Informationen versorgen.

Der Speichervorgang kann je nach System und Warehousegröße länger dauern. In dieser Zeit können in der Regel keine weiteren Anfragen verarbeitet werden. Um keine eingehenden Anfragen zu blockieren, kümmert sich ein eigener Thread um die Speicherung des Warehouses. Bricht eine Node während des Speichervorgangs zusammen 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.

## 2.5 Verschlüsselte Übertragung der Chat-Nachrichten

*Troy Keßler, Michael Angermeier*

Um einen sicheren Nachrichtenkanal zu gewährleisten, ist eine Ende-zu-Ende-Verschlüsselung implementiert. Beim Erstellen eines Chats wird ein Schlüssel mithilfe des Diffie-Hellman-Key-Exchange unter allen Teilnehmern generiert. Dieser Schlüssel wird im Client mit der korrespondierenden Chat-Id gespeichert, sodass verschlüsselte Nachrichten auch nach einem Neustart des Clients wieder gelesen werden können. Dies ist notwendig, da auf dem Server nur die verschlüsselten Nachrichten gespeichert sind und sonst keinerlei Möglichkeit bestehen würde, diese wiederherzustellen.

Die öffentlich bekannte Primzahl, sowie die Generatorprimzahl wurden dabei für jeden Client festgeschrieben. Nach jedem Login und dem 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-Key-Exchange entsprechend erweitert werden. Dabei gibt es in Gruppen im Gegensatz zu zwei Teilnehmern mehrere Runden. Ein Schlüsselaustausch mit drei Teilnehmern kann aus dieser Abbildung entnommen werden.

$g$  = Erzeugerprimzahl  
 $n$  = 128bit Primzahl

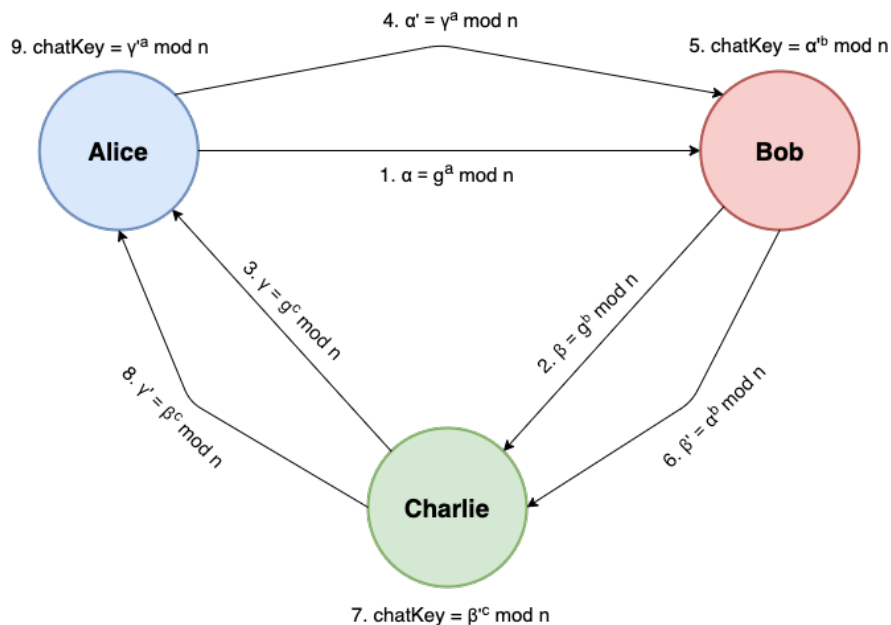


Abbildung 2: Erweiterter Diffie-Hellman

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 werden mithilfe der Ergebnisse der ersten Runde, die fertigen Chat-Schlüssel berechnet (Schritte 4-9). Die Größe der Primzahl  $n$  beträgt 128bit und die Länge der Generatorprimzahl 32bit. Die Schlüssellänge beträgt ebenfalls 128bit.

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

Sei  $n$  die Anzahl der Teilnehmer, so gilt:

$$\begin{aligned} rounds &= n - 1 \\ requests &= n \cdot (n - 1) \end{aligned}$$

Nachdem alle Chatteilnehmer den finalen Schlüssel berechnet haben kann der Chat erstellt werden. Dies ist der Fall, wenn die Bedingung

$$currentRequests = (targetRequests - userIndex)$$

erfüllt ist, wobei *targetRequests* die theoretische Anzahl an Requests die gesendet werden müssen darstellt, mit

$$targetRequests = n \cdot (n - 1).$$

Die Variable *currentRequests* beschreibt, wie oft das Paket schon weitergeleitet wurde. Der *userIndex* steht für die aktuelle Position im Schlüsselaustausch. Dabei ist der Initiator, mit einem Index von 0, der Erste. Im obigen Beispiel mit drei Teilnehmern wäre also die Endbedingung erfüllt, wenn der Initiator ein Paket erhält, welches sechs mal weitergeleitet wurde.

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

Für alle anderen Clients gilt eine ähnliche Endbedingung:

$$currentRequests - 1 = (targetRequests - userIndex)$$

Damit der Schlüsselaustausch abgeschlossen werden kann müssen alle Teilnehmer während der Chaterstellung online sein. Ist dies nicht der Fall wird darauf hingewiesen. Da nun alle Chatteilnehmer den gleichen Schlüssel besitzen und der Chat erstellt ist, können Nachrichten mit einem symmetrischen Verfahren sicher versendet werden. Hierfür wird die AES-Verschlüsselung verwendet.

## 2.6 Verschlüsselte serverseitige Speicherung der Chats

*Troy Keßler*

Durch die in 2.5 beschriebene Ende-zu-Ende Verschlüsselung liegen dem Server die Nachrichten **nie** als Klartext vor. Wie bereits in 2.4 erwähnt werden alle Nachrichten im Warehouse gespeichert und das gesamte Warehouse wird abgespeichert. So liegen auch in der Speicherdatei die Nachrichten niemals als Klartext 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      var mainThread = new Thread(server);  
      mainThread.start();  
10  }
```

../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, ?  
68    extends Packet>>();  
72    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 mit allen Threads geteilt wird.

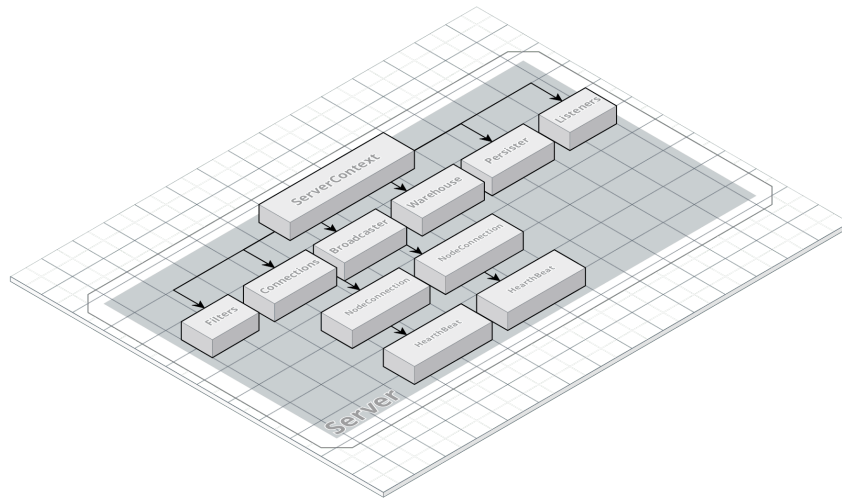


Abbildung 3: Server-Context Aufbau

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:");
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.

```

18 public void run() {
    var warehouse = this.contex.getWarehouse();
    warehouse.load();
20 while (!this.contex.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/Persister.java

66 public synchronized void save() throws IOException {
67     File tempFile = File.createTempFile(this.saveFileName, ".tmp");
68     try (var stream = new FileOutputStream(tempFile)) {
69         var outputStream = new ObjectOutputStream(stream);
70         outputStream.writeObject(this.warehouse);
71         outputStream.writeObject(this.packetIds);
72     }
73     Files.move(Paths.get(tempFile.getPath()), Paths.get(new File(
74         this.saveFileName + ".dat").getPath()),
75         StandardCopyOption.ATOMIC_MOVE);
76 }

../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
15     NodeConfig... configs) {
16     for (var config: configs) {
17         var conn = new NodeConnection(config.getAddress(), config.
18             getPort(), context);
19         conn.start();
20         nodes.add(conn);
21     }
22 }
23
24 public void send(final Packet packet) {
25     System.out.println("Broadcasting: " + packet);
26     for (var node : nodes) {
27         node.send(packet);
28     }
29 }
30 }

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

```

Weitere Variablen sind *isCloseRequested*, die die 'endlos' Schleifen aller Threads steuert und *connections*, welche alle Verbindungen 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 sollen werden vom Broadcaster in

die Queue geschrieben und die NodeConnection wird durch einen Semaphor aufgeweckt. Die NodeConnection versucht eine Nachricht zu senden. Scheitert das Senden wird von einer Verbindungstrennung ausgegangen und die Verbindung wird neu aufgebaut.

```

38         runSemaphore.acquire();
39         try {
40             if (this.currentSocket != null && out != null) {
41                 var packet = this.sendQueue.peek();
42                 if (packet != null) {
43                     out.writeObject(packet);
44                     out.flush();
45                     this.sendQueue.remove();
46                 }
47             }
48             catch (IOException e) {
49                 this.reconnect();
50                 this.runSemaphore.release();
51             }
52         }
53     }
54 }
55
56 ..../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() {
17     while (!this.isCloseRequested) {
18         try {
19             out.send(new NoOpPacket());
20             Thread.sleep(BEATRATE);
21         } catch (InterruptedException e) {
22             e.printStackTrace();
23         }
24     }
25 }
26
27 ..../src/main/java/vs/chat/server/node/NodeHeartBeatThread.java

```

Nachdem nun alle Initialisierungsvorgänge abgeschlossen sind, kann der Server-Socket erstellt 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)) {
43             while (!this.context.isCloseRequested().get()) {
44                 try {
45                     var clientSocket = socket.accept();
46                     var outputStream = new ObjectOutputStream(clientSocket.getOutputStream());
47                     var inputStream = new ObjectInputStream(clientSocket.getInputStream());
48
49                     var connectionHandler = new ConnectionHandler(
50                         clientSocket, this.context, outputStream,
51                         inputStream);
52                     this.context.getConnections().add(connectionHandler);
53                     connectionHandler.start();
54                 } catch (IOException e) {
55                     // ...
56                 }
57             }
58         }
59     }
60 }
61
62 ..../src/main/java/vs/chat/server/Server.java

```

```

54         e.printStackTrace();
55     }
56     System.out.println("Stopping Server...");
57     this.context.close();
58 } catch (IOException | InterruptedException e) {
59     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();
49     var packet = (Packet) object;
50     System.out.println("Read Packet: " + packet.getClass().
    getSimpleName());
51     var canActivate = this.context.getFilters().stream()
    .allMatch(f -> f.canActivate(packet, this.context, this
52 ));
53     if (canActivate) {
54         this.handlePacket(packet);
55     }
56 }

```

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

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

```

64 private void handlePacket(final Packet packet) throws IOException
65 {
66     if (packet instanceof LogoutPacket) {
67         this.pushTo(new LogoutSuccessPacket());
68         this.closeRequested = true;
69         return;
70     }
71     for (var listener : this.context.getListeners()) {
72         try {
73             var methods = listener.getClass().getDeclaredMethods();
74             for (var method : methods) {
75                 if (method.getName().equals("next") && !method.
76 isSynthetic()) {
77                     var packetType = method.getParameters()[0];
78                     if (packetType.getType().isAssignableFrom(packet.
79 getClass())) {
80                         var retu = (Packet) method.invoke(listener, packet,
81 this.context, this);
82                         this.pushTo(retu);
83                     }
84                 }
85             }
86         } catch (SecurityException | IllegalArgumentException |
87 IllegalAccessException
88 | InvocationTargetException e) {
89             e.printStackTrace();
90         }
91     }
92     if (!(packet instanceof NoOpPacket))

```

```

88 |         this.context.getWarehouse().print();
    |     }
    |
    | ..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 Packet gehandelt werden, um rekursive Broadcasts zu vermeiden. Bereits gesehene Pakete werden im Warehouse mitgespeichert. Im seltenen Fall, dass die Node genau zwischen den Listnern 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)
38         .get(message.getTarget())).getUsers();
    } else if (entity instanceof User) {
40         distributionUser = context.getWarehouse().get(
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);
        if (localConnection.isPresent()) {
48             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()) {
        packet.setOrigin(handler.getConnectedToUserId().get());
18     }
    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).

```

    var res = context.getWarehouse().get(WarehouseResourceType
        .USERS).values().stream()
26     .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();
            handler.setConnectedToUserId(id);
32             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();
39 |         var id = user.getId();
40 |         user.setUsername(packet.getUsername());
41 |         user.setPassword(packet.getPassword());
42 |
43 |         context.getWarehouse().get(WarehouseResourceType.USERS).
44 |         put(id, user);
45 |         System.out.println("created user with id:" + id);
46 |
47 |         var broadcastPacket = new BaseEntityBroadcastPacket(user
48 |         );
49 |
50 |         var b = new BaseEntityBroadcastPacket(new User(user.
51 |         getId(), user.getUsername()));
52 |         for (var others : context.getWarehouse().get(
53 |         WarehouseResourceType.USERS).keySet()) {
54 |             var localConnection = context.getConnectionForUserId(
55 |             others);
56 |             if (localConnection.isPresent()) {
57 |                 localConnection.get().pushTo(b);
58 |             }
59 |         }
60 |
61 |         context.getBroadcaster().send(broadcastPacket);
62 |         handler.setConnectedToUserId(id);
63 |
64 |     }
    |_____|
    |../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.

```

60 |         var chats = context.getWarehouse().get(
61 |         WarehouseResourceType.CHATS).values().stream().map(chat ->
62 |         (Chat) chat)
63 |         .filter(chat -> chat.getUsers().contains(handler.
64 |         getConnectedToUserId().get()))
65 |         .collect(Collectors.toSet());
66 |         var users = context.getWarehouse().get(
67 |         WarehouseResourceType.USERS).values().stream().map(user ->
68 |         {
69 |             var u = (PasswordUser) user;
70 |             return new User(u.getId(), u.getUsername());
71 |         }).collect(Collectors.toSet());
72 |
73 |     }
    |_____|
    |../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());

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

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 Nutzer handelt (es wird ein Neuer angelegt) oder es ein bereits existierender Nutzer 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 Chat-Id, die Schlüssel geladen). Abschließend wird aus den beiden festgeschriebenen, öffentliche Primzahlen noch ein Schlüssel generiert, der später für den Diffie-Hellman-Key-Exchange notwendig ist.

```

126     public void login(String username, String password) throws
        LoginException {
128         try {
            LoginPacket loginPacket = new LoginPacket(username
            , password);

130             this.networkOut.writeObject(loginPacket);
            this.networkOut.flush();

132             Object response = this.networkIn.readObject();

134             if (response instanceof NoOpPacket) {
136                 throw new LoginException();
            }

138             LoginSyncPacket loginSyncPacket = (LoginSyncPacket
            ) response;

140             this.lastUsername = username;
142             this.lastPassword = password;

144             this.userId = loginSyncPacket.userId;
            this.chats = loginSyncPacket.chats;
146             this.contacts = loginSyncPacket.users;

148             this.keyfile = new Keyfile(username);
            keyfile.load();
150             keyfile.save();

152             this.generatePrivateKey();

```

```

154         } catch (IOException | ClassNotFoundException e) {
            throw new LoginException();
156     }

```

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

- Funktion - generatePrivateKey

Mit dieser Funktion wird ein 128 Schlüssel aus den beiden öffentlichen Primzahlen *n* und *g* generiert. Der Schlüssel wird für die Verschlüsselung verwendet.

```

166     private void generatePrivateKey() {
        SecureRandom random = new SecureRandom();
168         byte[] bytes = new byte[KEY_BYTELENGTH];
        random.nextBytes(bytes);

170         this.privateKey = BigInteger.valueOf(ByteBuffer.wrap(
            bytes).getLong()).abs();
172         this.nextKey = this.g.modPow(this.privateKey, this.n);
        System.out.println("\nGenerated private key: " + this.
            privateKey);
174     }

```

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

- Funktion - exchangeKeys

Um den Diffie-Hellman-Key-Exchange 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 zehn Sekunden Zeit, um den Schlüsseltausch durchzuführen. Wenn dies innerhalb der angegebenen Zeit nicht geschieht, wird von einem Fehler ausgegangen und die Error-Handling-Methode *onTimeout* aufgerufen.

```

202     public void exchangeKeys(String chatName, List<UUID>
        userIds, OnTimeout onTimeout) throws IOException,
        InterruptedException {
            userIds.add(0, this.userId);
204         KeyExchangePacket keyExchangePacket = new
            KeyExchangePacket(
                this.nextKey,
206                 1,
                this.userId,
208                 userIds,
                chatName,
210                 userIds.get(1));

212         this.networkOut.writeObject(keyExchangePacket);
214         this.networkOut.flush();
        this.creatingChat = true;
216
        Thread.sleep(10000);
218
        if (this.creatingChat) {
220             this.creatingChat = false;
            onTimeout.run();
222     }

```

```
}

```

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

```

- Funktion - createChat

Für einen neuen Chat werden zwei Parameter benötigt: Der Name des Chats und eine Liste mit Usern. Um einen neuen Chat zu erstellen, wird ein CreateChatPacket mit dem Chatnamen und einem Array der User-Ids an den Server geschickt.

```

226 private void createChat(String chatName, List<UUID>
      userIds) throws IOException {
228     UUID[] chatUsers = new UUID[userIds.size()];
      chatUsers = userIds.toArray(chatUsers);

      CreateChatPacket createChatPacket = new
230 CreateChatPacket(chatName, chatUsers);
      this.networkOut.writeObject(createChatPacket);
232     this.networkOut.flush();
    }

```

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

```

- Funktion - sendMessage

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

```

234 public void sendMessage(String message, UUID chatId)
      throws IOException {
236     String chatKey = loadKey(chatId).toString();
      MessagePacket messagePacket = new MessagePacket(chatId
      , encryptAES(chatKey, message));

238     this.networkOut.writeObject(messagePacket);
      this.networkOut.flush();
240 }

```

```
../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.

```

242 public String encryptAES(String key, String message) {
      try {
244         Cipher cipher = Cipher.getInstance("AES/ECB/
      PKCS5Padding");
        cipher.init(Cipher.ENCRYPT_MODE, setKey(key));
246         return Base64.getEncoder().encodeToString(cipher.
      doFinal(message.getBytes(StandardCharsets.UTF_8)));
    }

```

```

248         } catch (Exception e) {
            e.printStackTrace();
        }
250     return null;
    }

252     public String decryptAES(String key, String ciffre) {
254         try {
            Cipher cipher = Cipher.getInstance("AES/ECB/
PKCS5PADDING");
256             cipher.init(Cipher.DECRYPT_MODE, setKey(key));
            return new String(cipher.doFinal(Base64.getDecoder
().decode(ciffre.getBytes(StandardCharsets.UTF_8))));
258         } catch (Exception e) {
            e.printStackTrace();
260         }
        return null;
262     }

264     //Formatting key to SerectKeySpec
    public SecretKeySpec setKey(String myKey) {
266         MessageDigest sha;
        byte[] key;
268         try {
            key = myKey.getBytes(StandardCharsets.UTF_8);
270             sha = MessageDigest.getInstance("SHA-256");
            key = sha.digest(key);
272             key = Arrays.copyOf(key, KEY_BYTELENGTH);
            return new SecretKeySpec(key, "AES");
274         } catch (NoSuchAlgorithmException e) {
            e.printStackTrace();
276         }
        return null;
278     }

```

../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.

loadKey

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

deleteKey

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

```

280     public void addKey(UUID chatId, BigInteger key) {
        var pKEntry = new PrivateKeyEntity(chatId, key);
282         this.keyfile.get(KeyfileResourceType.PRIVATEKEY).put(
            chatId, pKEntry);
        try {
284             keyfile.save();
        } catch (IOException e1){
286             e1.printStackTrace();
        }
    }

```

```

288     }
290     public BigInteger loadKey(UUID chatId) {
291         var res = keyfile.get(KeyfileResourceType.PRIVATEKEY).
292             values().stream()
293                 .filter(u -> ((PrivateKeyEntity) u).equals(
294                     chatId)).findFirst();
295         if (res.isPresent()) {
296             PrivateKeyEntity pke = (PrivateKeyEntity) keyfile.
297                 get(KeyfileResourceType.PRIVATEKEY).get(chatId);
298             try {
299                 keyfile.save();
300             } catch (IOException e1){
301                 e1.printStackTrace();
302             }
303             return pke.getPrivateKey();
304         }
305         try {
306             keyfile.save();
307         } catch (IOException e1){
308             e1.printStackTrace();
309         }
310         return null;
311     }
312     public void deleteKey(UUID chatId) {
313         keyfile.get(KeyfileResourceType.PRIVATEKEY).remove(
314             chatId);
315         try {
316             keyfile.save();
317         } catch (IOException e1){
318             e1.printStackTrace();
319         }
320     }

```

../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!

```

320     public void exit() throws IOException {
321         LogoutPacket logoutPacket = new LogoutPacket();
322         this.networkOut.writeObject(logoutPacket);
323         this.networkOut.flush();
324     }

```

../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 Nutzer online sind.)

```

326         new Thread(new Runnable() {
327             @Override
328             public void run() {
329                 while (true) {
330                     try {
331                         Object packet = networkIn.readObject()
332                     }
333                     ;
334                     if (packet instanceof
335                         BaseEntityBroadcastPacket) {
336                         BaseEntityBroadcastPacket base = (
337                             BaseEntityBroadcastPacket) packet;
338
339                         if (base.getBaseEntity()
340                             instanceof Chat) {
341                             creatingChat = false;
342                             Chat newChat = (Chat) base.
343                                 getBaseEntity();
344                             addKey(newChat.getId(),
345                                 generatePrivateKey();
346                                 nextKey = g.modPow(privateKey,
347                                     n);
348                             chats.add(newChat);
349                             onCreateChat.run(newChat);
350                         } else if (base.getBaseEntity()
351                             instanceof Message) {
352                             Message m = (Message) base.
353                                 getBaseEntity();
354                             Message d = new Message(m.
355                                 getOrigin(), m.getReceiveTime());
356                             d.setTarget(m.getTarget());
357                             String chatKey = loadKey(m.
358                                 getTarget().toString());
359                             d.setContent(decryptAES(
360                                 chatKey, m.getContent()));

```

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

//TODO 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.

```

358         } else if (base.getBaseEntity()
359             instanceof User) {

```



```

360         getBaseEntity();
361         User u = (User) base.contacts.add(u);
362         System.out.println("Added new User " + u.getUsername() + " ");
363         System.out.print("> ");
364     }
365     } else if (packet instanceof KeyExchangePacket) {
366         KeyExchangePacket keyExchangePacket = (KeyExchangePacket) packet;
367         List<UUID> participants = keyExchangePacket.getParticipants();
368         int currentRequests = keyExchangePacket.getRequests();
369         BigInteger currentContent = keyExchangePacket.getContent();
370         int targetRequests = participants.size() * (participants.size() - 1);
371         int userIndex = participants.indexOf(userId);
372         int rounds = currentRequests / participants.size();
373         if (keyExchangePacket.getInitiator().equals(userId)) {
374             nextKey = currentContent.modPow(privateKey, n);
375         } else {
376             rounds++;
377         }
378         System.out.println("\n-> Exchanging keys round " + rounds);
379         // check if package has to be forwarded
380         if (currentRequests < targetRequests) {
381             // forwards package to next participant
382             KeyExchangePacket newExchangePacket = new KeyExchangePacket(
383                 nextKey,
384                 keyExchangePacket.getRequests() + 1,
385                 keyExchangePacket.getInitiator(),
386                 keyExchangePacket.getParticipants(),
387                 keyExchangePacket.getChatName(),
388                 participants.get((userIndex + 1) % participants.size()));
389             networkOut.writeObject(newExchangePacket);
390             networkOut.flush();
391         }

```

```

398                                     nextKey = currentContent.modPow(
privateKey, n);
400
finished key exchange // check if participant has
402                                     if (keyExchangePacket.getInitiator
().equals(userId)) {
                                     if (currentRequests == (
targetRequests - userIndex)) {
..../src/main/java/vs/chat/client/ClientApiImpl.java

```

//TODO 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.

```

406                                     participants.remove(0);
keyExchangePacket.getChatName(), participants);
408                                     } else if (currentRequests == (
targetRequests - (participants.size() - userIndex))) {
                                     System.out.println(
410                                     getUsernameFromId(userId) + " -> " + nextKey);
                                     }
412                                     } else if (packet instanceof
GetMessagesResponsePacket) {
                                     Set<Message> messages = ((
414                                     GetMessagesResponsePacket) packet).getMessages();
                                     Set<Message> decrypted = new
TreeSet<>();
416                                     for (Message m: messages) {
                                     Message d = new Message(m.
418                                     getOrigin(), m.getReceiveTime());
                                     d.setTarget(m.getTarget());
..../src/main/java/vs/chat/client/ClientApiImpl.java

```

//TODO 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.

```

430                                     } catch (EOFException | SocketException e)
{
432                                     reconnect();
                                     } catch (IOException |
ClassNotFoundException e) {
434                                     e.printStackTrace();
                                     }
}

```

```

436         try {
438             socket.close();
        }
    }
}

```

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

//TODO Verwendung von Emojis:

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

```

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

```

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

//TODO 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
```

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