Dokumentation

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

16. April 2020

Inhaltsverzeichnis

1 Basisanforderungen

1.1 Chatfunktionalität

Matthias Vonend

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

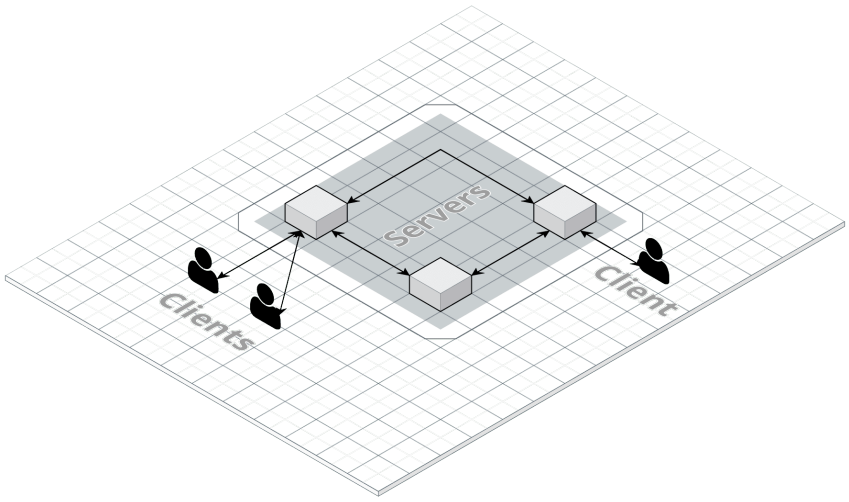

Abbildung 1: Architektur

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 Persistenzgründen 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 einen ihr bekannten Client bestimmt war und sendet diese gegebenenfalls an diesen.

//TODO Nach einer erfolgreichen Anmeldung kann der Nutzer

1.2 Clientfunktionalitäten

Jan Grübener, Troy Keßler, 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 Chats angezeigt.

Funktion: contacts //TODO

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 eingetraggen werden. Hierfür wird jeder einzelener 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 dementsprechend eine andere Node dessen Aufgabe übernehmen.

1.3.1 Client

//TODO Troy //TODO Fat Client

Im Fehlerfall scheitert das Senden einer Nachricht an den Server. In diesem Fall versucht sich der Client mit einer anderen Node zu verbinden und sendet die Nachricht erneut.

1.3.2 Server

Serverseitig können verschiedene Fehler auftretten. Viele Fehler werden bereits durch das TCP-Protokoll verhindert (z.B. mehrfach Zustellung, fehlerhafte

Übermittlung). Dennoch können grundsätzlich die folgende Fehlerfälle eintretten:

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 müssen ähnlich zur Clientverbindung der Server 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 Netzwerkpatitionierungen klar. Werden die Server getrennt sind die Client immernoch 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 durch 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 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 weiter gegeben 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 dis-

playRecentConversations() für die jeweilige Funktionalität aufgerufen.

//TODO: Erkläen: Wie bekommt die GUI neue Chatnachrichten (eigener "Listener"Thread)
//TODO (Wenn noch Platz vorhanden): startGui() erklären wie Daten aus Entites gelesen wird

2.2 Verwendung von Emojis

Jan Grübener, Patrick Mischka

Hier haben wir es uns zu nutze gemacht, dass die meisten gängigen Emojis bereits als Unicode Zeichen vorhanden sind. Durch das Verwenden der Unicode Zeichen kann eine zu versenden Nachricht weiterhin als String an eine Message Entität übergeben werden. 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, indem das Gridlayout welches die Methode renderEmojiPanel() liefert angepasst wird.

Auf Serverseite mussten hierfür keine Veränderungen vorgenommen werden.

2.3 Mehrere Chatverläufe pro Nutzer

Matthias Vonend, Troy Keßler

//TODO

2.4 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 nur beliebig viele Nutzer. Wird eine Nachricht empfangen werden nun alle am Chatbeteiligten Nutzer durchlaufen und die Nachricht wird an alle der Node bekannten Clients weitergeleitet. Außerdem wird wie bereits erwähnt die Nachricht aus Konsistenzgründen weiter gebroadcastet.

2.5 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 Keßler, 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 ankommenende 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.

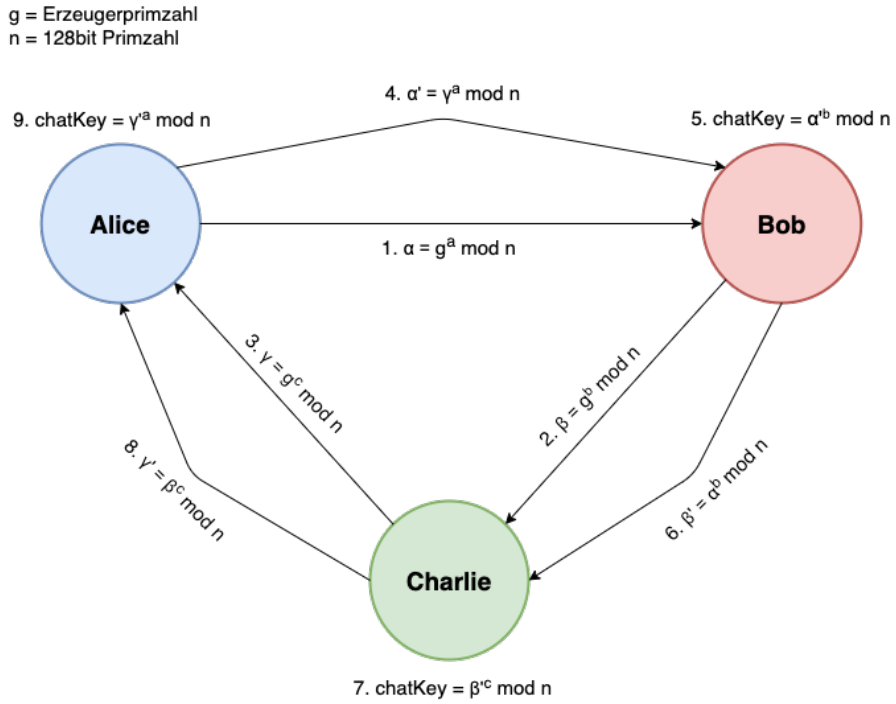

Abbildung 2: Erweiterter Diffie-Hellman

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

$$rounds = n - 1$$

$$requests = n \cdot (n - 1)$$

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

$$currentRequests = (targetRequests - userIndex)$$

prüfen, ob der Schlüsselaustausch vollständig durchgeführt wurde. Dabei steht $currentRequests$ für die Anzahl, wie oft das Packet schon weitergeleitet wurde, diese wird beim versenden immer 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 diese bei ihm immer 0

ist. Im obigen Beispiel wäre also die Endbedingung erfüllt wenn der Initiator ein Packet 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.

```
  public static void main(final String[] args) {
8   var server = new Server(9876, new NodeConfig("localhost", 9877)
    );
    var mainThread = new Thread(server);
10  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, ?
    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 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.

```
  public synchronized void load() {
54  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();
```

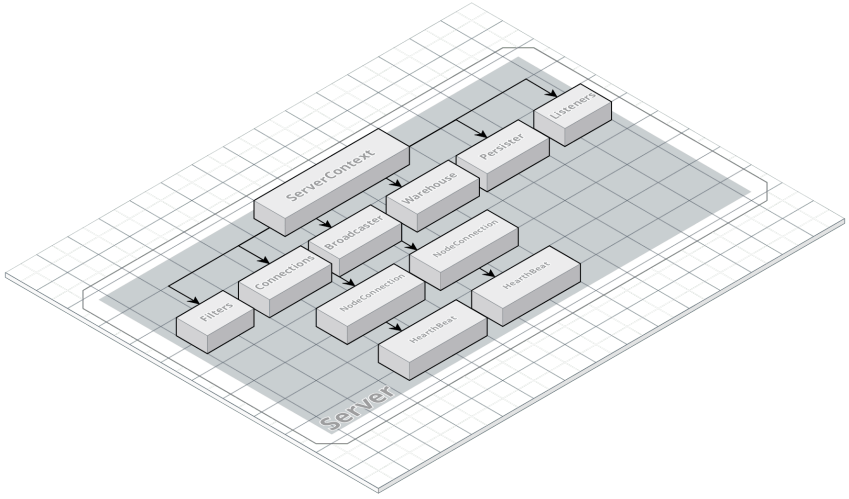

Abbildung 3: Server-Context Aufbau

```
        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 Persister. Der Persister ist ein Thread, der in regelmäßigen Abständen das Warehouse speichert.

```
  public void run() {
18  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(SAVE_INTERVAL);
      } catch (IOException | InterruptedException e1) {
28      e1.printStackTrace();
      }
30  }
  }
```

../src/main/java/vs/chat/server/persistence/Persister.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);
            outputStream.writeObject(this.packetIds);
72        }
          Files.move(Paths.get(tempFile.getPath()), Paths.get(new File(
          this.saveFileName + ".dat").getPath()),
74            StandardCopyOption.ATOMIC_MOVE);
      }
```

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

Alle Entitäten und Packete haben eine UUID (Universally Unique Identifier) um diese zu unterscheiden. Verweiße 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 granularere Syncronisation 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 ëndlosSSchleifen aller Threads steuert und connections, welche alle Connections zu Clients, die direkt zu dieser Node verbunden sind, hällt.

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 eine Semaphore aufgeweckt. Die NodeConnection versucht eine Nachricht zu senden. Scheitert das senden wird von einer Verbindungstrennung ausgegangen und die Verbindung wird neu verbunden.

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

```
42                out.writeObject(packet);
                  out.flush();
44                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());
20        Thread.sleep(BEAT_RATE);
        } catch (InterruptedException e) {
22        e.printStackTrace();
        }
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.

```
    try (var socket = new ServerSocket(PORT)) {
42    while (!this.context.isCloseRequested().get()) {
        try {
44        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 Packete ausgetauscht. Der Handler versucht dabei ein Packet vom Client zu lesen. Die Filter prüfen nun, ob das Packet gehandelt werden darf (und nach dem handeln werden diese aktualisiert).

```
        var object = inputStream.readObject();
48      var packet = (Packet) object;
        var canActivate = this.context.getFilters().stream().
    allMatch(f -> f.canActivate(packet, this.context, this));
50      if (canActivate) {
          this.handlePacket(packet);
52        this.context.getFilters().stream().forEach(f ->
    postHandle(packet, this.context, this));
        }
```

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

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

```
  private void handlePacket(final Packet packet) throws IOException
     {
62    if (packet instanceof LogoutPacket) {
        this.pushTo(new LogoutSuccessPacket());
64      this.closeRequested = true;
        return;
66    }
      for (var listener : this.context.getListeners()) {
68      try {
          var methods = listener.getClass().getDeclaredMethods();
70        for (var method : methods) {
            if (method.getName().equals("next") && !method.
    isSynthetic()) {
72            var packetType = method.getParameters()[0];
              if (packetType.getType().isAssignableFrom(packet.
    getClass())) {
74              var retu = (Packet) method.invoke(listener, packet,
    this.context, this);
                this.pushTo(retu);
76            }
            }
78        }

80      } catch (SecurityException | IllegalArgumentException |
    IllegalAccessException
            | InvocationTargetException e) {
82        e.printStackTrace();
        }
84    }
      this.context.getWarehouse().print();
86  }
```

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

Filter:

- PacketIdFilter
  Der PacketId-Filter testet, ob ein Packet 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

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

```
  @Override
10 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 BaseEntityBroadcastPackete, 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.

```
        var chat = (Chat) entity;
34      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()));
```

```
      }

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

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

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

```
    packet.getUsers().add(currentUser.get());

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


    Chat newChat = new Chat(packet.getName(), filteredUsers);

    context.getWarehouse().get(WarehouseResourceType.CHATS).
    put(newChat.getId(), newChat);

    var broadcastPacket = new BaseEntityBroadcastPacket(
    newChat);
    context.getBroadcaster().send(broadcastPacket);

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

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

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

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

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

    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 KeyExchangePackete von einem Client an andere Clients weiter (gegebenenfalls über andere Nodes).

```
    var currentUser = handler.getConnectedToUserId();
16  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();
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.

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

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

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

      var broadcastPacket = new BaseEntityBroadcastPacket(user
    );

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

      context.getBroadcaster().send(broadcastPacket);
      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)
        .filter(chat -> chat.getUsers().contains(handler.
    getConnectedToUserId().get()))
        .collect(Collectors.toSet());
    var users = context.getWarehouse().get(
    WarehouseResourceType.USERS).values().stream().map(user ->
     {
      var u = (PasswordUser) user;
      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.

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

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

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

```
    context.getWarehouse().get(WarehouseResourceType.MESSAGES)
    .put(newMessage.getId(), newMessage);

    var broadcastPacket = new BaseEntityBroadcastPacket(
    newMessage);
    for (var user : correspondingChat.getUsers()) {
      var localConnection = context.getConnectionForUserId(
    user);
      if (localConnection.isPresent()) {
        localConnection.get().pushTo(broadcastPacket);
      }
    }
    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 Packete indem er prüft ob eine Änderung vorliegt und wenn ja diese übernimmt und broadcastet.

```
    var needsBroadcast = false;
    for (var id : packet.packetIds) {
      if (!context.getWarehouse().knowsPacket(id)) {
        context.getWarehouse().savePacket(id);
        needsBroadcast = true;
      }
    }
    for (var type : WarehouseResourceType.values()) {
      for (var entry : packet.warehouse.get(type).entrySet())
    {
        if (null == context.getWarehouse().get(type).get(entry
    .getKey())) {
          context.getWarehouse().get(type).put(entry.getKey(),
     entry.getValue());//BaseEntityBroadcast
          needsBroadcast = true;
        }
      }
    }

    if (needsBroadcast) {
      context.getBroadcaster().send(packet);
    }
```

../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.

Sollen hier andere Packete auch erkärt werden? Also Logout, GetMessageResponsePacket, NoOp, ...?

3.2 Client

API-Funktionen

- Funktion - login
  Bei der Funktion login werden er Benutzername und des 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 File ist wie die warehouse File aufgebaut (hier wird über die ChatId, die Schlüssel geladen). Abschließend wird aus den beiden hardgecodeden public Keys noch ein private Key generiert, der später für den Diffie Hellman Schlüsseltausch notwendig ist.

```
58      public void login(String username, String password) throws
         LoginException {
            try {
60              LoginPacket loginPacket = new LoginPacket(username
        , password);

62              this.networkOut.writeObject(loginPacket);
                this.networkOut.flush();
64
                Object response = this.networkIn.readObject();
66
                if (response instanceof NoOpPacket) {
68                  throw new LoginException();
                }
70
                LoginSyncPacket loginSyncPacket = (LoginSyncPacket
        )response;
72
                this.userId = loginSyncPacket.userId;
74              this.chats = loginSyncPacket.chats;
                this.contacts = loginSyncPacket.users;
76
                this.keyfile = new Keyfile(username);
78              keyfile.load();
                keyfile.save();
80
                this.generatePrivateKey();
82          } catch (IOException | ClassNotFoundException e) {
                throw new LoginException();
84          }
        }
```

../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. Dieser ist für die Verschlüsselung wichtig.

```
        private void generatePrivateKey() {
96          SecureRandom random = new SecureRandom();
```

```
        byte[] bytes = new byte[KEY_BYTE_LENGTH];
98      random.nextBytes(bytes);

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

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

- Funktion - exchangeKeys

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

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

158     CreateChatPacket createChatPacket = new
    CreateChatPacket(chatName, chatUsers);
        this.networkOut.writeObject(createChatPacket);
160     this.networkOut.flush();
    }
```

../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.

```
    public void sendMessage(String message, UUID chatId)
    throws IOException {
164   String chatKey = loadKey(chatId).toString();
        MessagePacket messagePacket = new MessagePacket(chatId
    , encryptAES(chatKey, message));
166
        this.networkOut.writeObject(messagePacket);
168     this.networkOut.flush();
    }
```

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

- Funktion - Verschlüsselung (encryptAES, decyptAES, 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 initalisiert. 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.

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

182 public String decryptAES(String key, String ciffre) {
        try {
184         Cipher cipher = Cipher.getInstance("AES/ECB/
    PKCS5PADDING");
            cipher.init(Cipher.DECRYPT_MODE, setKey(key));
186         return new String(cipher.doFinal(Base64.getDecoder
    ().decode(ciffre.getBytes(StandardCharsets.UTF_8))));
        } catch (Exception e) {
188         e.printStackTrace();
        }
190     return null;
    }
192
    //Formatting key to SerectKeySpec
194 public SecretKeySpec setKey(String myKey) {
        MessageDigest sha;
196     byte[] key;
        try {
198         key = myKey.getBytes(StandardCharsets.UTF_8);
            sha = MessageDigest.getInstance("SHA-256");
200         key = sha.digest(key);
            key = Arrays.copyOf(key, KEY_BYTE_LENGTH);
202         return new SecretKeySpec(key, "AES");
        } catch (NoSuchAlgorithmException e) {
204         e.printStackTrace();
        }
206     return null;
    }
```

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

- Funktion - Keyfile (addKey, loadKey, deleteKey) 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

```
    public void addKey(UUID chatId, BigInteger key) {
```

```
210         var pKEntry = new PrivateKeyEntity(chatId, key);
            this.keyfile.get(KeyfileResourceType.PRIVATEKEY).put(
        chatId, pKEntry);
212         try {
                keyfile.save();
214         } catch (IOException e1){
                e1.printStackTrace();
216         }
        }
218
        public BigInteger loadKey(UUID chatId) {
220         var res = keyfile.get(KeyfileResourceType.PRIVATEKEY).
        values().stream()
                    .filter(u -> ((PrivateKeyEntity) u).equals(
        chatId)).findFirst();
222         if (res.isPresent()) {
                PrivateKeyEntity pke = (PrivateKeyEntity) keyfile.
        get(KeyfileResourceType.PRIVATEKEY).get(chatId);
224             try {
                    keyfile.save();
226             } catch (IOException e1){
                    e1.printStackTrace();
228             }
                return pke.getPrivateKey();
230         }
            try {
232             keyfile.save();
            } catch (IOException e1){
234             e1.printStackTrace();
            }
236         return null;
        }
```

../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     public void exit() throws IOException {
            LogoutPacket logoutPacket = new LogoutPacket();
250         this.networkOut.writeObject(logoutPacket);
            this.networkOut.flush();
252     }
```

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