Dokumentation

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

14. 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.

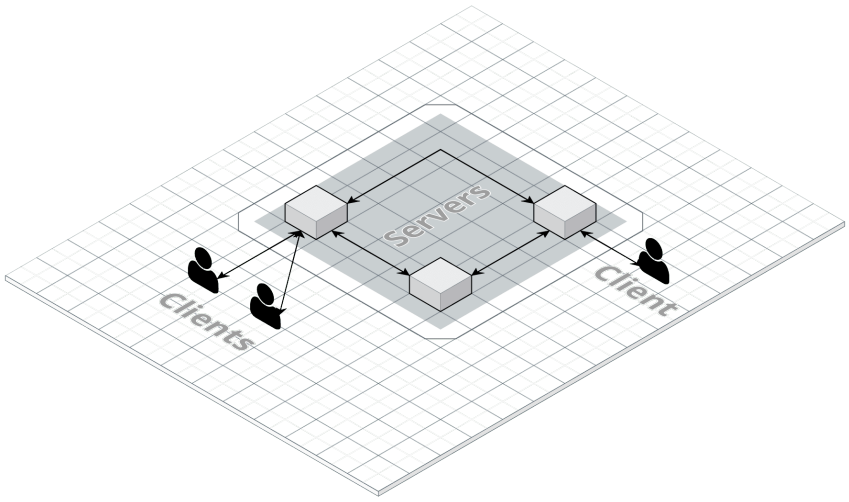

Abbildung 1: Architektur

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 Command-Line-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

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. 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
   Ähnlich zur Clientverbindung muss 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. 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 Servern.

2 Erweiterungen

2.1 Grafische Oberfläche bei den Nutzern

Jan Grübener, Patrick Mischka

Wählt der Nutzer nach dem Starten des Clients die Variante mit Grafischer Benutzeroberfläche starten, ruft der Client die Methode startGui() auf. Hierbei wird ein JFrame erzeugt, auf dem im Laufe der Zeit unterschiedliche JPanels hinzugefügt und entfernt werden. Je nachdem an welchem Punkt der Nutzer sich gerade befindet werden die entsprechenden Methoden wie loginPanel() oder displayRecentConversations() für die jeweilige Funktionalität aufgerufen.

//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. //TODO update this section to include packet ids

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.

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 dem untererem Schaubild 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.

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.

Da nun alle Chatteilnehmer den gleichen Schlüssel besitzen und dieser lokal gesichert ist können Nachrichten mit einem symetrischen Verfahren sicher versendet werden.

Für das symmetrische Verfahren wurde die AES Verschlüsselung implementiert.

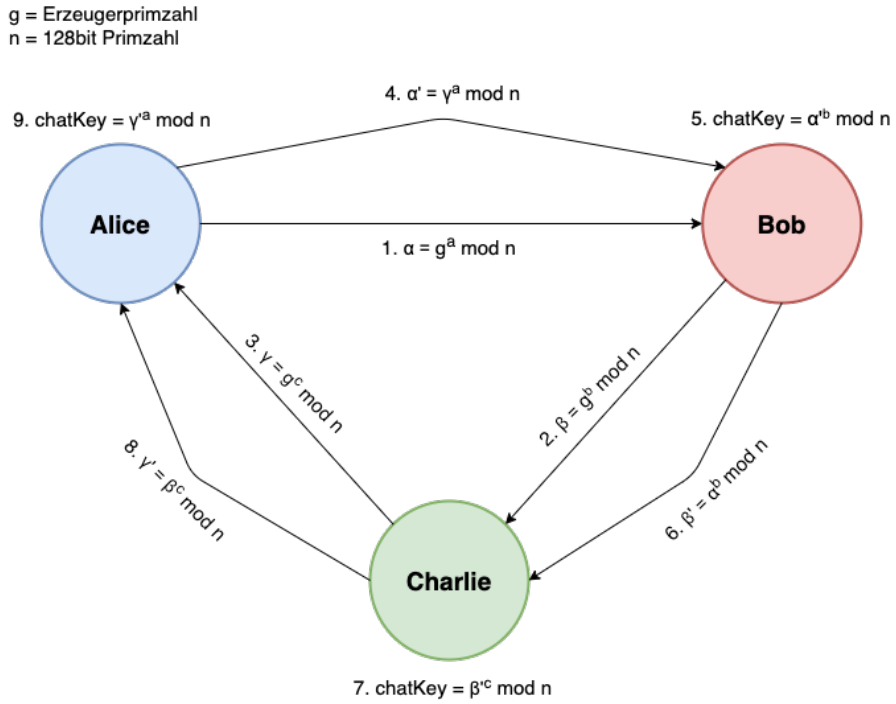

Abbildung 2: Erweiterter Diffie-Hellman

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 stehts 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.

```
    this.outputStream = outputStream;
    this.inputStream = inputStream;
  }

  @Override
  public void run() {
    Thread.currentThread().setName("Connection Handler");
    while (!this.context.isCloseRequested().get() && !
    closeRequested) {
      System.out.println("Handling");
      try {
        var object = inputStream.readObject();
        var packet = (Packet) object;
        var canActivate = this.context.getFilters().stream().
    allMatch(f -> f.canActivate(packet, this.context, this));
        if (canActivate) {
          this.handlePacket(packet);
          this.context.getFilters().stream().forEach(f ->
    postHandle(packet, this.context, this));
        }
      } catch (IOException | ClassNotFoundException e) {
        break;
      }
    }
    this.close();
  }

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

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

Entrypoint ist der Serverbootstrapper. Dieser erstellt einen neuen Thread mit einem neuen Server. Der Server erstellt die Listener, die die zu empfangenen Packete handeln werden. Anschließend werden die Filter erstellt, die bestimmen, ob ein Packet gehandelt oder ignoriert werden soll (z.B. bei recursiven Broadcasts). Die Listener und die Filter werden in einen ServerContext gepackt, der allen Threads geteilt wird.

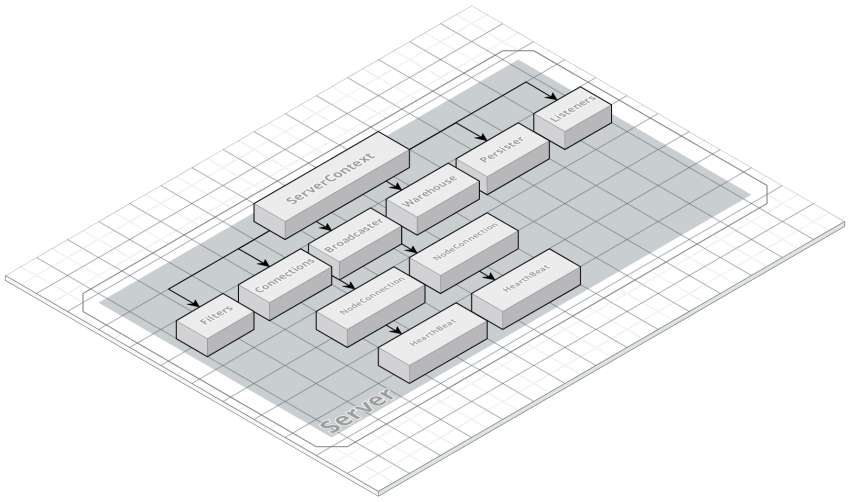

Abbildung 3: Server-Context Aufbau

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. 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 regelmaßigen Abständen das Warehouse speichert (müsste in ener section schon beschrieben sein). Außerdem wird der Broadcaster erstellt, der die Verbindungen zu anderen Nodes hält und empfangene Nachrichten an diese verteilt. Weitere Variablen sind isCloseRequested, die die ëndlosSSchleifen aller Threads steuert und connections. Diese Liste hällt alle Connections zu Clients, die direkt zu dieser Node verbunden sind.

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

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. 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 wenn ja werden die passenden Handler mit dem Packet aufgerufen und die Filter aktualisieren sich.

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 recursive Broadcasts zu vermeiden. Bereits gesehene Packete 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.

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. 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.

- 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.

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

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

- LoginListener
  Der LoginListener kümmert sich um die Authentifizierung eines Clients. Er prüft ob ein Nutzer besteht und falls ja wird das Passwort geprüft. Existiert noch kein Nutzer wird ein passender Nutzer erstellt. 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. 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).

- 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.

- 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. Theoretisch kann es sein, dass ein Nutzer sich anmeldet bevor die Node ihre Syncronisation abgeschlossen hat. Dieser Fehlerfall wird aber ignoriert, da die Syncronisationszeit und die Ausfallwahrscheinlichkeit einer Node als zu gering eingestuft wird.

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

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.

4 TODOS

Code walkthrough + kommentieren Titlepage überarbeiten

Code-Listings