Dokumentation

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

8. April 2020

Inhaltsverzeichnis

1 Basisanforderungen

1.1 Chatfunktionalität

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

Möchte dieser eine Nachricht an einen weiteren Client senden, schickt er diese an seine verbundene Node und überlässt die Zustellung dieser. Da alle Nachrich-

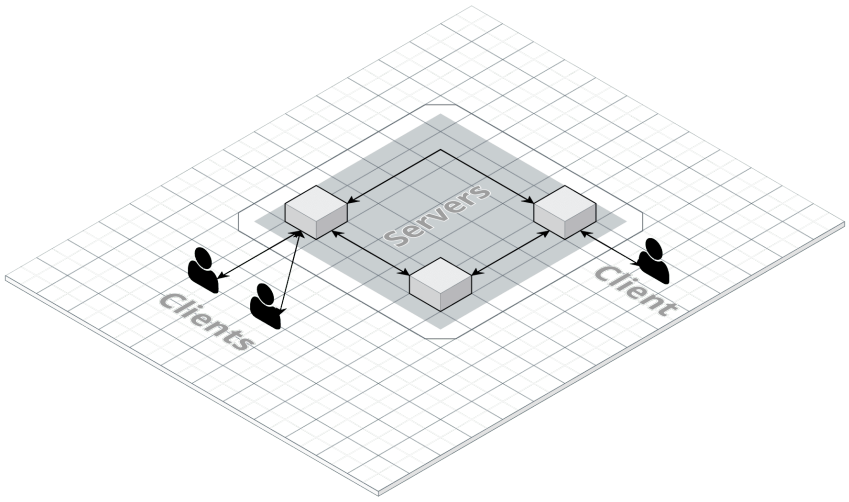

Abbildung 1: Architektur

ten 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 // create chat und so bla :)

1.2 Fehlerbehandlung

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.

1.2.1 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.

1.2.2 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 stehts eine synchronisation durch wodurch diese immer den kompletten Informationsbestand besitzen.

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

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

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

//TODO

2.4 Gruppenchats

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

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

//TODO

2.7 Verschlüsselte serverseitige Speicherung der Chats

//TODO

3 Anhang

```
    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;
        this.handlePacket(packet);
      } 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);
            }
          }
        }

      } catch (SecurityException | IllegalArgumentException |
    IllegalAccessException
```

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

4 TODOS

Das doc schreiben duh :P Konfliktvermeidung ansprechen -¿ Datenbanken vergleich mit UUID als Fremdschlüssel UUIDs erwähnen Code walkthrough + kommentieren Titlepage überarbeiten