Netzwerk Protokoll Spezifikation

Buddler Joe

March 24, 2019

Inhalt

1 Übersicht und Hierarchie

1.1 Protokolltyp

Unser Netzwerkprotokoll basiert auf dem Übertragungssteuerungsprotokoll (engl. TCP). Wir stellen für jeden Client in einem eigenen Thread eine Verbindung mittels der Java Bibliothek *java.net.Socket* her. Die übertragenen Daten sind einfach, von Menschen lesbare Strings. Jede übertragene Nachricht besteht aus einem 5-stelligen Buchstabencode, gefolgt von einem Leerzeichen, gefolgt von einem String, welcher alle Daten enthält welche, bei Bedarf, mit unserem Protokoll-Trennzeichen || verbunden resp. aufgeteilt werden können.

Die Steuerung des Netzwerkprotokolls ist auf 5 Kernklassen verteilt, welche im folgenden beschrieben werden.

1.2 Klassenhierarchie

1.2.1 Client Seite

GameGUI Wird aufgerufen beim Spielstart und fungiert als Interface mit dem Client. Nimmt Port und IP entgegen und erstellt damit eine Instanz der *ClientLogic*. Das GUI bietet ein Konsolen- oder ein Grafisches Interface. Befehle über das Interface werden in protokolleigenen Paketen verarbeitet und dann via die Klasse *ClientLogic* an den Server gsesendet. Für Meilenstein 2 ist das GameGUI ein simples Konsoleninterface, es wird jedoch später natürlich ausgebaut. Die Main Klasse befindet sich im *net* Packet und heisst *StartNetworkOnlyClient*. Später wird die Funktion des GameGUI durch das volle GUI des Spiels übernommen.

ClientLogic Auf die Klasse ClientLogic wird weitgehend statisch zugegriffen. Sie wird vom *GameGUI* einmalig mit IP und Port über den Konstruktor initialisiert und verbindet so zum Server via *java.net.Socket*. Dann setzt sie ihre statischen Variablen für den In- und Output zum Server. Im Konstruktor wird ausserdem ein Thread gestartet, in welchem die Klasse den Socket von Server stetig ausliest und vor-verarbeitet.

Die ClientLogic verwaltet den In- und Output Stream zum Server und stellt Methoden zum Senden von protokolleigenen Paketen an den Server zur Verfügung.

1.2.2 Server Seite

ServerGUI Nimmt einen Port entgegen und initialisiert mit diesem Port die *ServerLogic* Klasse. Falls ein serverseitiges Konsoleninterface gebraucht wird, dann würde dies in dieser Klasse entstehen. Die Main Klasse befindet sich im *net* Packet und heisst *StartServer*.

ServerLogic Auf die Klasse ServerLogic wird weitgehend statisch zugegriffen. Sie wird vom *ServerGUI* einmalig mit dem Port über den Konstruktor initialisiert und erstellt via *java.net.ServerSocket* einen Socket auf welchen die Clients verbinden können. Die ServerLogic verwaltet ausserdem zwei Listen: Eine Liste mit allen Lobbies, welche auf dem Server aktiv sind und eine Liste aller Clients mit ihrem Thread welche zum Server verbunden sind.

In einer Schleife wartet die ServerLogic auf neue Verbindungen von Clients. Verbindet sich ein neuer Client, kriegt dieser Client eine einzigartige clientId, es wird ein neuer *ClientThread* erstellt und schliesslich wird beides in der entsprechenden Liste gespeichert. Nachrichten an einen Client gehen durch die ServerLogic, welche die Nachricht an den ensprechenden ClientThread weiterleitet.

ClientThread Der ClientThread wird für jeden verbundenen Client initialisiert, hat eine eindeutige ID und ist für die Verbindung zu diesem Client verantwortlich. Im ClientThread sind die via *java.net.Socket* erstellten In- und Output Streams zum entsprechenden Client gespeichert. Die Klasse ClientThread befindet sich im Packet *net.playerhandling*.

Jeder ClientThread läuft in einem eigenen Thread und liest in einer Schlaufe auf dem input Socket die vom Client gesendeten Nachrichten. Die Nachrichten werden anhand des Headers an das verantwortliche protokolleigene Paket weitergeleitet. Der ClientThread enthält ebenfalls eine Methode um ein protokolleigenes Paket an den Client zu übermitteln.

2 Struktur der protokolleigenen Pakete

Unser Protokoll funktioniert mit vordefinierten Netzwerkpaketen. Jedes Paket übernimmt genau eine Funktion. Die Pakete sind selber verantwortlich für die Validierung, Verarbeitung und Fehlerbehandlung der eigenen Daten.

2.1 ENUM für PaketTypen

Das ENUM für die PaketTypen befindet sich in der Abstrakten Klasse für alle Pakete und definiert alle Paket-Typen welche das Protokoll unterstützt. Ein PaketTyp ist definiert durch den 5-stelligen Buchstabencode, welcher auch als Header für die Nachrichten dient. Das ENUM weist ausserdem jedem PaketCode eine beschreibende Variable zu, damit der Code deutlich an Leserlichkeit gewinnt. Das Paket mit dem Code "STNMS" zum Beispiel wird so im Code mit PaketTypes.SET_NAME_STATUS referenziert. Ein ENUM bietet uns ausserdem viel Flexibilität und ermöglicht eine einfache Erweiterung um zusätzliche Pakete.

2.2 Abstrake Klasse für alle Pakete

Protokolleigene Pakete haben einen Typ und enthalten einen String mit allen Daten. Falls das Paket von einem Client versendet wurde, wird dies in clientId gespeichert. Ausserdem hat jedes Paket eine Liste mit Fehlern im String (klartext) Format. Ist diese Liste nicht leer, dann können die Fehler mit den entsprechenden Methoden ausgelesen und behandelt werden.

```
private PacketTypes packetType;
private String data;
private int clientId;
private List<String> errors = new ArrayList<>();
```

Die abstrakte Klasse enthält Methoden welche für alle Pakete identisch sind, wie:

- Getter und Setter für data und clientId
- Bool'sche Methode zum überprüfen ob ein Paket Fehler enthält
- Getter und Adder für Fehler sowie eine Methode welche alle Fehler zu einem String verbindet und zurück gibt
- toString: Transformiert Daten zu einem lesbaren String welcher als Nachricht über das Protokoll geschickt wird
- Funktionen zum Senden des Pakets an Client, Lobby oder Server
- Allgemeine Validierungsfunktionen welche von mehreren Paketen genutzt werden

Ausserdem schreibt die abstrakte Klasse vor, dass jedes Paket die folgenden zwei Methoden implementieren muss:

- validate(): Validiert alle Klassenvariablen und erstellt gegebenenfalls Fehler mit aussagekräftigen Fehlermeldungen. Fehler werden im Paket gespeichert wie oben beschrieben. Diese Funktion wird am Ende des Konstruktors aufgerufen und hat **keinen Zugriff** auf Informationen ausserhalb des Pakets! Sprich, die Funktion muss auf Server- und Clientseite gleichermassen funktionieren.

- processData(): Wird nach dem empfangen eines Pakets ausgeführt und enthält einen Grossteil der Logik des Pakets. Diese Funktion kann ebenfalls Fehler zum Paket hinzufügen und hat Zugriff auf Klassen und Informationen auf der Server- und/oder Client Seite. Hier werden oft Antwortpakete erzeugt und versendet.

2.3 Implementierung der Pakete

Pakete werden von der abstrakten Klasse abgeleitet und definieren individuell weitere Klassenvariablen zum speichern und validieren der Daten für welche sie zuständig sind. Jeder PaketTyp ist genau für eine Funktion zuständig wie etwa *Login*, *Login Status Meldung* oder *Disconnect*.

Pakete müssen wenn möglich clientId und data setzten, sowie validate() und processData() implementieren, wie in 2.2 beschrieben.

Pakete sind in Gruppen geordnet, welche jeweils eine Funktionskategorie umfassen. Im nächsten Kapitel wird jede dieser Funktionskategorien mit ihren Paketen erläutert.

2.4 Kontrollfluss der Pakete

Ein Paket geht typischerweise durch die folgenden Schritte:

1. Der Konstruktor wird aufgerufen mit Spieldaten oder Userinput
    - Die übergebenen Variablen werden den entsprechenden Klassenvariablen zugeteilt
    - Die *data* variable wird generiert: Eine Aneinanderreihung der Klassenvariablen, getrennt durch das Protokoll-Trennzeichen
    - Die Klassenvariablen werden mittels der validate() methode validiert. Allfällige Fehler werden dem Paket angehängt
2. Falls das Paket Fehler enthält, können diese vor dem verschicken behandelt werden, indem zum Beispiel die entsprechende Send-Methode überschrieben wird
3. Das Paket wird versendet mittels einer der Send-Methoden welche den Output der toString() Methode verschicken
4. Der String wird auf der anderen Seite empfangen und anhand des Paket-Typ-Codes einem Paket zugewiesen
5. Der Konstruktor dieses Pakets wird aufgerufen mit dem *data-String* und falls vorhanden, der clientId
    - Der *data-String* wird in der *data* Variable gespeichert und dann Mittels dem Protokoll-Trenzeichen in die Klassenvariablen aufgesplittet
    - Die Klassenvariablen werden mittels der validate() methode validiert. Allfällige Fehler werden dem Paket angehängt
6. Die Methode processData() kann aufgerufen werden. Hier Findet die Fehlerbehandlung und die ganze Logik des Pakets Platz. Falls notwendig werden hier Antwortpakete erstellt

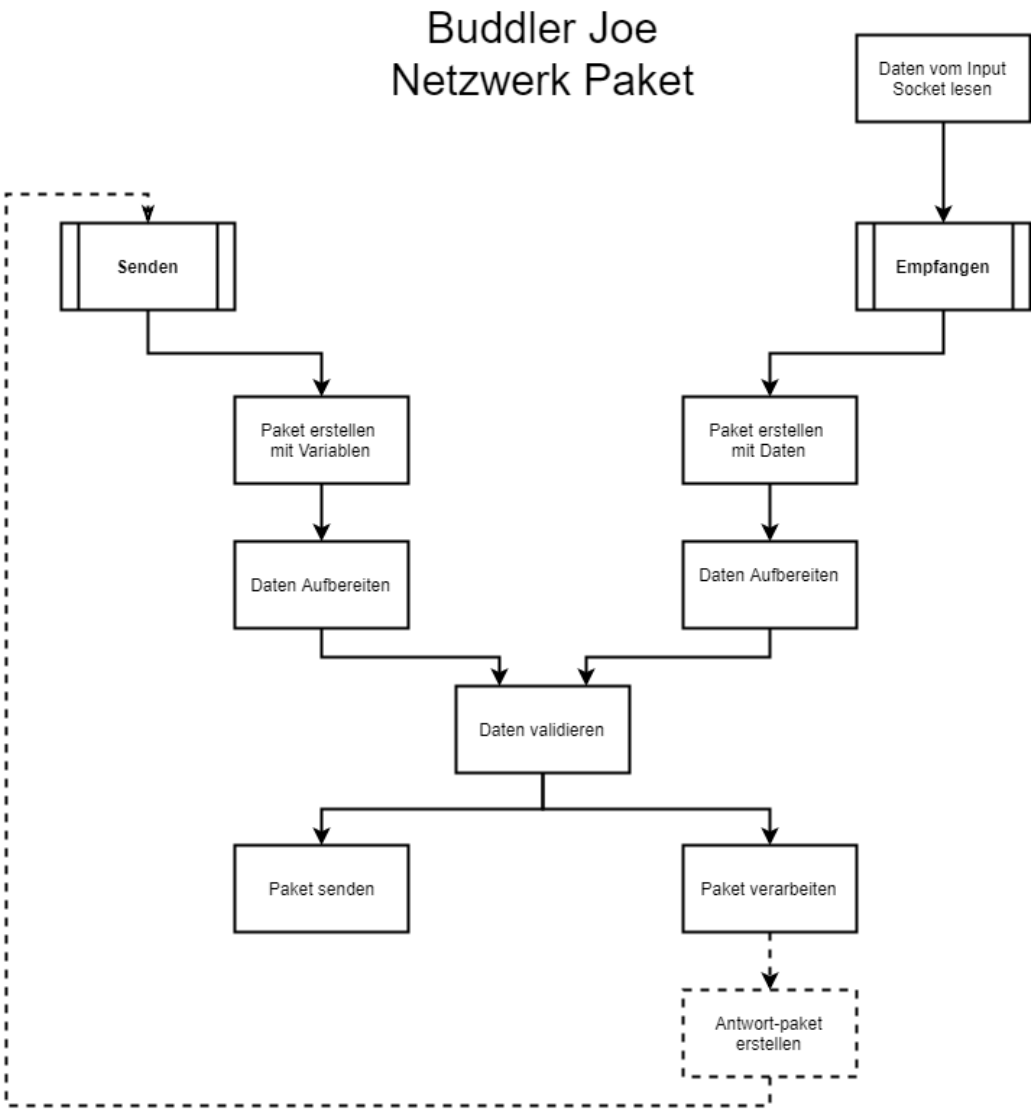

Figure 1: Grafische Darstellung des Paket-Kontrollfluss

3 Funktionskategorien der Pakete

3.1 Login und Logout

Java-Package: net.packets.login_logout

Der Server empfängt den Login eines Clients mit Username. Der Username wird validiert und der Server prüft, ob der Spieler ohne Konflikte erstellt werden kann. Ist dies möglich wird der Spieler erstellt in die Liste der eingeloggten Spieler eingetragen. In jedem Fall wird der Server eine Antwort an den Client senden mit dem Resultat des Logins.

Das Disconnect Packet kann entweder vom User gesendet werden bei einem ordnungsgemässen Logout, oder es wird vom Server generiert, falls die Verbindung zum Client nicht ordnungsgemäss abbricht. Das Disconnect Packet löscht den Spieler aus allen relvanten Listen und sendet die notwendigen Pakete an andere Clients um diese vom Logout zu informieren.

3.1.1 Packet: LOGIN

Allgemein

Code	PLOGI
Klasse	PacketLogin.java
Absender	Client
Empfänger	Server

Klassenvariablen

username

Beschreibung	Gewünschter Benutzername im Spiel
Validierung	Ist vorhanden. Mit Hilfe der checkUsername Methode der Abstrakten Packet Klasse wird überprüft, ob der username nicht zu kurz/zu lang ist, ob er im extended Ascii ist und ob er überhaupt vorhanden ist. Siehe 2.2 für mehr Informationen.

Verarbeitung. Falls der Benutzername bereits existiert auf dem Server, wird eine aufsteigende Zahl an den Namen angehängt, bis der Name einzigartig ist. Danach wird versucht den Spieler zu erstellen sofern dieser noch nicht eingeloggt ist und den Spieler zur Liste der eingeloggten Spieler hinzuzufügen.

In jedem Fall wird ein Login-Status-Packet erstellt und dem Client zurück gesendet.

Beispiel. "PLOGI Joe_Buddler" von Client zu Server erstellt den Spieler "Joe_Buddler" auf dem Server und sendet ein Antwortpaket mit dem Inhalt "OK" zurück.

3.1.2 Packet: LOGIN_STATUS

Allgemein

Code	PLOGS
Klasse	PacketLoginStatus.java
Absender	Server
Empfänger	Client

Klassenvariablen

status

Beschreibung	Resultat des Login-Versuchs
Validierung	Nur extended ASCII und nicht leer

Verarbeitung. Falls der Login erfolgreich war, wird status = "OK" an den Client gesendet. Ansonsten wird eine Liste von Fehlern gesendet, welche der Client individuell anzeigt oder verarbeitet. Der Benuzter wird etwa informiert, wenn der Server den Namen angepasst hat.

Beispiel. "PLOGS OK peter" von Server zu Client signalisiert einen erfolgreichen Login und zeigt beim Client den Text: <"Login Successful, your username is: peter"> an.

3.1.3 Packet: DISCONNECT

Allgemein

Code	DISCP
Klasse	PacketDisconnect.java
Absender	Client oder Server
Empfänger	Server

Klassenvariablen

keine

Verarbeitung. Der Server ruft die Methode removePlayer in der Classe ServerLogic auf mit der SpielerID als Parameter. Diese Methode prüft ob der dazugehörender Spieler in der Spielerliste vorhanden ist. Anschliessend wird überprüft, ob sich der Spieler in einer Lobby befindet. Wenn ja, wird der Spieler aus der Lobby geworfen. Der Spieler Thread wird geschlossen und der Spieler wird von der Spielerliste genommen. Dann wird eine Nachricht an die Spieler in der Lobby geschickt, dass der entsprechende Spieler das Spiel verlassen hat. Zuletzt wird der Lobbystatus an alle in der Lobby gesendet, damit sie den Überblick über die Lobby haben.

Beispiel. Ein Spieler, welcher in einer Lobby ist, schreibt in die Konsole <disconnect>. Der Spieler wird dann aus der Lobby und von der Spielerliste genommen. Die restlichen Spieler in der Lobby bekommen dann eine Nachricht vom Server das der Spieler die Lobby verlassen hat. Danach noch zusätzlich den aktuellen Stand der Lobby.

3.2 Name

Java-Package: net.packets.name

Dieses Package enthält alle Paket-Klassen die mit dem Name setting/getting zu tun haben. Dazu gehört zum Beispiel eine setName Klasse, welche dem Spieler ermötlicht den Namen zu ändern oder auch eine getName, welche einem Spieler ermöglich von jedem anderen Spieler und sich selber den username zu erhalten.

3.2.1 Packet: SET_NAME

Allgemein

Code	SETNM
Klasse	PacketSetName.java
Absender	Client
Empfänger	Server

Klassenvariablen

username

Beschreibung	Ein String, der den username enthält, welcher der player gerne setzen möchte.
Validierung	Ist vorhanden. Mit Hilfe der checkUsername Methode wird überprüft, ob der username nicht zu kurz/zu lang ist, ob er im extended Ascii ist und ob er überhaupt vorhanden ist.

Verarbeitung. Falls in der validierungsmethode ein Fehler gefunden wird, wird eine Fehler Nachricht erstellt und als status an ein PacketSetNameStatus weitergegeben und an den client zurückgeschickt. Falls der Name schon in der serverPlayerList vorhanden ist, wird mit Hilfe eines Zählers so lange der username variiert, bis der Name einmalig ist und dann geändert. Sobald der Name einmalig ist, wird eine Statusmeldung erstellt und auch einer PacketSetNameStatus Klasse übergeben und dem Client geschickt. . Falls der Name schon von Anfang an einmalig ist, wird einfach dieser neue Name gesetzt und ein Status mit einer PacketSetNameStatus Klasse an den Client zurückgeschickt.

Beispiel. "SETNM peter" von Client zu Server. Angenommen der username peter ist schon vorhanden, so wird so lange mit dem Counter hochgezählt, bis peter_ + Counter einmalig ist. Nehmen wir an, dies sei bei peter_1 der Fall, so würde der username vom Client auf peter_1 geändert und ein Status "Changed to: peter_1. Because your chosen name is already in use." mit einem PacketSetNameStatus an den Client zurückgeschickt.

3.2.2 Packet: SET_NAME_STATUS

Allgemein

Code	STNMS
Klasse	PacketSetNameStatus.java
Absender	Server
Empfänger	Client

Klassenvariablen

status

Beschreibung	Ein String, der den Status des Name settings enthält
Validierung	Ist vorhanden. Wird geprüft, ob der Status vorhanden ist und ob der Status extended Ascii ist.

Verarbeitung. Falls in der Validierungsmethode ein Fehler gefunden wird, wird eine Fehler Nachricht erstellt und dem Client ausgegeben. Falls der Status mit "Successfully" beginnt, war das name setting erfolgreich, falls der Status mit "Changed" beginnt, wurde der Name noch abgeändert und es wird der neue Username angezeit.

Beispiel. "STNMS Successfully changed the name to: peter" vom Server zum Client. In diesem Fall war das Name setting erfolgreich und der neue username des Clients ist peter. Diese Nachricht wird dann auch dem Spieler auf der Konsole ausgegeben.

3.2.3 Packet: GET_NAME

Allgemein

Code	GETNM
Klasse	PacketGetName.java
Absender	Client
Empfänger	Server

Klassenvariablen

playerId

Beschreibung	Ein String, der die clientId des zu suchenden client enthalten sollte. Sollte also ein Integer beinhalten.
Validierung	Ist vorhanden. Wird geprüft, ob es sich wirklich um ein Integer handelt, ob der Spieler überhaupt in der Liste ist und ob playerId überhaupt vorhanden ist.

Verarbeitung. Falls in der Validierungsmethode ein Fehler gefunden wird, wird eine Fehler Nachricht erstellt und als status an ein PacketSendName weitergegeben und an den client zurückgeschickt. Wenn der Spieler in der playerList gefunden wurde, wird der gesuchte username zusammen mit "OK" an PacketSendName übergeben und an den client geschickt.

Beispiel. "GETNM 1" von Client zu Server. Angenommen der Spieler mit der clientId 1 ist in der playerList vorhanden, dann wird der gesuchte username via ein PacketSendName an den Spieler zurückgeschickt.

3.2.4 Packet: SEND_NAME

Allgemein

Code	SENDN
Klasse	PacketSendName.java
Absender	Server
Empfänger	Client

Klassenvariablen

name

Beschreibung	Ein String, der den username des Spielers, der zuvor mit dem PacketGetName gesucht wurde.
Validierung	Ist vorhanden. Wird geprüft, ob der username vorhanden ist und ob der username extended Ascii ist.

Verarbeitung. Falls in der validierungsmethode ein Fehler gefunden wird, wird eine Fehler Nachricht erstellt und dem client angezeigt. Falls der username mit "OK" beginnt, wird dem client der gefundene username angezeigt. Falls jedoch der username aus einer Fehlernachricht aus der PacketGetName Klasse besteht, wird diese ausgegeben.

Beispiel. "SENDN OKpeter" vom Server zum Client. Angenommen das SendName Packet beginnt mit "OK", dann wird der zuvor gesuchte username, in unserem Beispiel peter dem Client ausgegeben.

3.3 Lobby

Java-Package: net.packets.lobby

Dieses Packet enthält alle Packet-Klassen die mit dem Lobbysystem etwas zu tun haben. Dazu gehören Beispielsweise Pakete, die dazu verwendet werden um dem Benutzer eine Übersicht über verfügbare Lobbys zu geben, Pakete für das Erstellen neuer Lobbys oder auch Pakete, die es dem Benutzer erlauben einer Lobby beizutreten, beziehungsweise sie wieder zu verlassen.

3.3.1 Packet: GET_LOBBIES

Allgemein

Code	LOBGE
Klasse	PacketGetLobbies.java
Absender	Client
Empfänger	Server

Klassenvariablen

keine

Verarbeitung. Falls der Benutzer, der das Packet an den Server gesendet hat, noch nicht angemeldet ist wird eine Fehlermeldung zurück gegeben. Ansonsten wird eine Auflistung von maximal zehn Lobbys erstellt, welche nicht voll sind (Maximale Anzahl an Lobbymitgliedern ist noch nicht definitiv festgelegt). Die Auflistung wird in einem String gespeichert, mit welchem dann ein LOBBY_OVERVIEW Paket erstellt wird. Falls Fehler auftraten enthält das neu erstellte Packet die Fehlermeldung. Ansonsten beginnt der String mit "OK".

Beispiel. "LOBGE" von Client zu Server. Angenommen es gäbe eine Lobby namens myLobby, mit der lobbyId 1 und darin befänden sich 3 Spieler. Es wird eine Auflistung der verfügbaren Lobbys erstellt und ein LOBBY_OVERVIEW Paket mit dem inhalt "OK|| Name: ones, LobbyId: 1, Spieler: 3|| " an den Benutzer geschickt, der "LOBGE" gesendet hat.

3.3.2 Packet: LOBBY_OVERVIEW

Allgemein

Code	LOBOV
Klasse	PacketLobbyOverview.java
Absender	Server
Empfänger	Client

Klassenvariablen

in[]

Beschreibung	Das Array enthält an jedem Index einen String der folgendermassen aufgebaut ist: "Name: lobbyName, LobbyId: lobbyId, Spieler: AmountOfPlayers"
Validierung	Ist vorhanden. Nur extended ASCII wird akzeptiert.

Verarbeitung. Es wird überprüft ob die empfangenen Daten mit "OK" beginnen. Ist dies der Fall, werden alle Strings die im Array in enthalten sind ausgegeben. Ansonsten wird nur die erste Stelle ausgegeben, welche dann die Fehlermeldungen enthält.

Beispiel. Der Server erhielt ein "LOBGE" von einem Benutzer. Es gibt jedoch noch keine Lobbys auf dem Server. Es wird also ein LOBBY_-OVERVIEW-Paket mit dem String "OK || No Lobbies online" erstellt und an den Benutzer zurück geschickt. Dieser zeigt daraufhin die Lobbyanasicht an. In welcher die Information "No Lobbies online" steht.

3.3.3 Packet: CREATE_LOBBY

Allgemein

Code	LOBCR
Klasse	PacketCreateLobby.java
Absender	Client
Empfänger	Server

Klassenvariablen

lobbyname

Beschreibung	Gewünschter Lobbyname.
Validierung	Ist vorhanden. Sollte mindestens 4 Zeichen und höchstens 16 Zeichen lang sein. Nur extended ASCII wird akzeptiert.

Verarbeitung. Es wird geprüft, ob der Benutzer, der das Packet gesendet hat, eingeloggt ist und ob er sich bereits in einer Lobby befindet. (Nicht eingeloggt oder bereits in einer Lobby wären hier ein Fehler). Wenn keine Fehler existieren, wird eine neue Lobby mit dem gewünschten Lobbynamen erstellt und der Liste, die der Server über die Lobbys führt, hinzugefügt. Falls das erstellen erfolgreich war, sendet der Server nun an alle Benutzer die noch nicht in einer Lobby sind ein LOBBY_OVERVIEW-Paket um sie über die neu Lobby zu informieren. Auf jeden Fall wird aber ein CREATE_LOBBY_STATUS-Packet an den Benutzer geschickt der den Befehl zum neu erstellen einer Lobby gesendet hat. Dieses enthält beim Erstellen einen String, welcher allfällige Fehlermeldungen oder die Information über denn Erfolg des Lobbyerstellen enthält.

Beispiel. Der Client sendet "LOBCR myLobby" an den Server. Dieser erstellt eine Lobby mit dem Namen myLobby auf dem Server und schickt uns ein CREATE_LOBBY_STATUS-Packet mit dem Inhalt "OK" zurück. An alle Benutzer die nicht in einer Lobby sind (auch wir selber) wird ausserdem ein LOBBY_OVERVIEW-Paket gesendet.

3.3.4 Packet: CREATE_LOBBY_STATUS

Allgemein

Code	LOBCS
Klasse	PacketCreateLobbyStatus.java
Absender	Server
Empfänger	Client

Klassenvariablen

status

Beschreibung	Ein String der Informationen darüber enthält ob das Erstellen der Lobby funktioniert hat. Wenn nicht Ist eine Fehlerbeschreibung enthalten, sonst "OK".
Validierung	Ist vorhanden. Nur extended ASCII wird akzeptiert.

Verarbeitung. Es wird überprüft ob die empfangenen Daten mit "OK" beginnen. Ist dies der Fall wird "Lobby-Creation Successful" ausgegeben, andernfalls die Fehlermeldung die in status enthalten ist oder eine Auflistung der Fehler, die bei der Validierung von status auftraten.

Beispiel. Der Server erhielt "LOBCR myLobby" und hat erfolgreich eine neue Lobby namens myLobby erstellt. Nun erstellt er ein CREATE_LOBBY_STATUS-Packet mit dem String "OK" ("OK" wurde von ServerLogic.getLobbyList().addLobby(lobby) zurückgegeben als die neue Lobby der Lobbyliste des Servers hinzugefügt wurde). Dieses Packet wird nun an den Client gesendet welcher darauf "Lobby-Creation Successful" ausgiebt.

3.3.5 Packet: JOIN_LOBBY

Allgemein

Code	LOBJO
Klasse	PacketJoinLobby.java
Absender	Client
Empfänger	Server

Klassenvariablen

lobbyname

Beschreibung	Name der Lobby der beigetreten werden soll.
Validierung	Nur extended ASCII.

Verarbeitung.Es wird überprüft ob eine Lobby existiert, die nach dem angegebenen Lobbynamen benannt ist, ob der Benutzer, der einer Lobby beitreten will auf dem Server angemeldet ist und ob der Benutzer bereits in einer Lobby ist. (Fehler wären: Es existiert keine entsprechende Lobby, der Benutzer ist noch nicht auf dem Server angemeldet oder er ist bereits in einer Lobby.). Sind keine Fehler festgestellt worden, wird der Benutzer der Lobby hinzugefügt. Danach wird an alle Benutzer die sich in der Lobby befinden der beigetreten wurde ein CUR_-LOBBY_INFO-Paket geschickt. Damit werden sie über denn neuen Nutzer informiert. Zudem wird an alle Benutzer die sich aktuell nicht in einer Lobby befinden ein LOBBY_OVERVIEW-Paket geschickt. Dadurch werden sie darüber informiert dass sich die Spielerzahl in der behandelten Lobby geändert hat. In jedem Fall wird aber ein JOIN_-LOBBY_STATUS-Paket an den Benutzer zurück geschickt der einer Lobby beitreten wollte. Dieses enthält dann entweder "OK", im Falle eines Erfolges und andernfalls eine entsprechende Fehlermeldung als String.

Beispiel. Wir haben uns auf dem Server eingeloggt und wollen nun der Lobby myLobby beitreten, die bereits existiert. Es wird also "LOBJO myLobby" an den Server gesendet. Da wir auch noch nicht in einer Lobby sind, fügt der Server uns der gewählten Lobby hinzu und schickt ein JOIN_LOBBY_STATUS-Paket an uns zurück. Alle anderen Benutzer werden zusätzlich auch über die Änderung informiert.

3.3.6 Packet: JOIN_LOBBY_STATUS

Allgemein

Code	LOBJS
Klasse	PacketJoinLobbyStatus.java
Absender	Server
Empfänger	Client

Klassenvariablen

status

Beschreibung	Ein String der Informationen darüber enthält, ob das Beitreten in die gewünschte Lobby funktioniert hat. Wenn nicht, Ist eine Fehlerbeschreibung enthalten.
Validierung	Ist vorhanden. Nur extended ASCII.

Verarbeitung.Es wird überprüft, ob die empfangenen Daten mit «OK» beginnen. Ist dies der Fall, wird "Successfully joined lobby" ausgegeben, andernfalls wird die Fehlermeldung, die in status enthalten ist oder eine Auflistung der Fehler, die bei der Validierung von status auftraten ausgegeben.

Beispiel. Nach dem wir „LOBJO myLobby“ an den Server schickten, hat dieser uns erfolgreich der Lobby myLobby hinzugefügt und schickt uns ein JOIN_LOBBY_STATUS-Paket zurück. Da dieses den Status „OK“ enthält. Wird bei uns „Successfully joined lobby“ angezeigt.

3.3.7 Packet: GET_LOBBY_INFO

Allgemein

Code	LOBGI
Klasse	PacketGetLobbyInfo.java
Absender	Client
Empfänger	Server

Klassenvariablen

keine

Verarbeitung.Es wird überprüft ob der Benutzer der das Packet geschickt hat, eingeloggt ist und sich in einer Lobby befindet. Falls dies nicht so wäre, würde ein Fehler vermerkt. Sind jedoch keine Fehler aufgetreten wird ein CUR_LOBBY_INFO-Paket erstellt. Dieses Beinhaltet in diesem Fall Informationen über die Lobby in der sich der Benutzer befindet (Momentan die Namen aller Benutzer der Lobby). Falls Fehler auftraten enthält das Antwort Packet entsprechende Fehlermeldungen.

Beispiel. Wir haben uns als Benutzer auf dem Server eingeloggt und schicken „LOBGI“. Der Server stellt fest, dass wir uns nicht in einer Lobby befinden und schickt ein CUR_LOBBY_INFO-Paket mit entsprechender Fehlermeldung an uns zurück. In diesem Fall wäre diese "Not in a lobby."

3.3.8 Packet: CUR_LOBBY_INFO

Allgemein

Code	LOBCI
Klasse	PacketCurrLobbyInfo.java
Absender	Server
Empfänger	Client

Klassenvariablen

info

Beschreibung	Ein String der Informationen über die Lobby enthält in der sich der Empfänger gerade befindet. Dies sind momentan die Namen aller Benutzer, die sich in dieser Lobby befinden. Er hätte beispielsweise die Struktur „OK||nameOne||nameTwo||. Falls es zuvor zu Fehlern kam, beispielsweise bei einem „LOBGI“ enthält der String entsprechende Fehlermeldungen.
Validierung	Ist vorhanden. Es wird geprüft, ob der String extended Ascii ist und ob er vorhanden ist.

infoArray

Beschreibung	Ein String-Array welches alle Namen die in info aufgelistet sind enthält. Das Array wird im konstruktor des Paketes gefüllt. Dabei wird info an den Stellen „||“ gespalten. Falls info eine Fehlermeldung enthält, wird diese in infoArray[0] gespeichert.
Validierung	Es wird jedes Element von infoArray darauf überprüft, ob es nur extendet ASCII Zeichen enthält.

Verarbeitung. Falls es in der Validierung zu Fehlern kam, wird eine entsprechende Fehlermeldung ausgegeben. Sonst wird geprüft ob infoArray[0] „OK“ entspricht, ist dies der Fall werden die restlichen Elemente von infoArray ausgegeben. Falls infoArray [0] ungleich „OK“ ist, wird nur infoArray [0], welches eine Fehlermeldung enthält, ausgegeben.

Beispiel. Wir haben uns als Benutzer auf dem Server eingeloggt und

schicken „LOBGI“. Der Server stellt fest, dass wir uns nicht in einer Lobby befinden und schickt ein CUR_LOBBY_INFO-Paket mit entsprechender Fehlermeldung an uns zurück. In diesem Fall wäre diese "Not in a lobby." Da infoArray[0] in diesem Fall nicht „OK“ enthält. Wird nur infoArray[0] also die Fehlermeldung ausgegeben.

3.3.9 Packet: LEAVE_LOBBY

Allgemein

Code	LOBLE
Klasse	PacketLeaveLobby.java
Absender	Client
Empfänger	Server

Klassenvariablen

keine

Verarbeitung.Es wird überprüft, ob der Benutzer der das Packet geschickt hat, eingeloggt ist und sich in einer Lobby befindet. Falls eines der beiden nicht gegeben ist, wird ein Fehler vermerkt. Sind keine Fehler vorhanden, entfernt der Server den Benutzer, der das Packet geschickt hat, aus seiner aktuellen Lobby. Danach wird an alle Benutzer, die sich in der Lobby befinden, aus der der Benutzer entfernt wurde, ein CUR_LOBBY_INFO-Paket geschickt. Damit werden sie über das Verlassen des Nutzers informiert. Zudem wird an alle Benutzer, die sich aktuell nicht in einer Lobby befinden (dazu zählt auch der Benutzer der jetzt gerade eine Lobby verlassen hat), ein LOBBY_OVERVIEW-Paket geschickt. Dadurch werden sie darüber informiert, dass sich die Spielerzahl in der behandelten Lobby geändert hat. In jedem Fall wird ein LEAVE_LOBBY_STATUS-Paket an den Benutzer zurück geschickt der die Lobby verlassen wollte. Dieses enthält dann entweder "OK", im Falle eines Erfolges und andernfalls eine entsprechende Fehlermeldung als String.

Beispiel. Wir sind als Benutzer in einer Lobby und wollen sie verlassen. Dazu senden wir „LOBLE“ an den Server. Da dieser keine Fehler feststellt, entfernt er uns aus der Lobby und informiert uns mit einem LEAVE_LOBBY_STATUS-Paket darüber, dass wir die Lobby verlassen konnten. Zudem erhalten wir auch gleich ein LOBBY_OVERVIEW-Paket, um wieder eine Übersicht über die verfügbaren Lobbys zu haben. Alle anderen Benutzer werden zusätzlich auch über die Änderung in der Lobby informiert.

3.3.10 Packet: LEAVE_LOBBY_STATUS

Allgemein

Code	LOBLS
Klasse	PacketLeaveLobbyStatus.java
Absender	Server
Empfänger	Client

Klassenvariablen

status

Beschreibung	Ein String der Informationen darüber enthält ob das Verlassen der Lobby funktioniert hat. Ist dies der Fall ist der String „OK“. Sonst enthält er eine entsprechende Fehlerbeschreibung.
Validierung	Ist vorhanden. Nur extended ASCII.

Verarbeitung.Es wird überprüft ob die empfangenen Daten mit «OK» beginnen. Ist dies der Fall wird "Successfully left lobby" ausgegeben, andernfalls die Fehlermeldung die in status enthalten ist oder eine Auflistung der Fehler, die bei der Validierung von status auftraten.

Beispiel. Wir befinden uns in einer Lobby. Nach dem wir „LOBLE“ an den Server schicken, entfernt dieser uns erfolgreich aus der Lobby und schickt uns ein LEAVE_LOBBY_STATUS-Paket zurück. Da dieses den Status „OK“ enthält. Wird bei uns „Successfully left lobby“ angezeigt.

3.4 Ping und Pong

Java-Package: net.packets.pingpong

Dieses Package enthält die Ping und die Pong Klasse, welche eine wichtige Rolle für den Erreichbarkeitstest von Server und Clients haben. Ein Ping Paket wird mit seiner Erstellungszeit von der Server wie auch von der Client Seite verschickt. Als Antwort darauf sollte ein Pong Paket von der Gegenseite erstellt und mit dem selben Inhalt zurückgeschickt werden, worauf der Absender das Pong Paket erhält und sich die Zeit merkt. Nun wird die Zeitdifferenz berechnet. An der Zeitdifferenz kann abgelesen werden, wie lange der Transport von Packeten hin zum Adressaten und zurück zum Absender braucht. Die Pingpakete werden im Hintergrund automatisiert verschickt und können vom Client selbst ausgelöst werden.

3.4.1 Packet: PING

Allgemein

Code	UPING
Klasse	PacketPing.java
Absender	Client und Server
Empfänger	Server und Client

Klassenvariablen

keine

Verarbeitung. Falls sich keine Fehler in der im Pingpaket übergebenen Uhrzeit verstecken, wird ein Antwort- bzw. ein Pongpaket erstellt, welchem abhängig von der Art des eingetroffenen Pingpakets zum Server oder zum Client verschickt wird. Enthielt das Pingpaket eine Nummer, welche den Client identifiziert, so wird das Pongpaket zum Client verschickt, anderenfalls zum Server.

Beispiel. Möchte ein Client die Erreichbarkeit testen, kann er ein Pingpaket mit dem Befehl "ping" erstellten und dies dem Server schicken. Der Server verarbeitet anschliessend das erhaltene Paket.

3.4.2 Packet: PONG

Allgemein

Code	PONGU
Klasse	PacketPong.java
Absender	Server und Client
Empfänger	Client und Server

Klassenvariablen

keine

Verarbeitung. Falls sich keine Fehler in der vom Pingpaket übergebenen Uhrzeit verstecken, so erfolgt die Berechnung der Zeitdifferenz. Hier wird von der aktuellen Uhrzeit die übergebene Zeit abgezogen und somit die Zeitdifferenz berechnet.

Beispiel. Man könnte in der Konsole den Befehl "PONGU" eingeben, jedoch würde er keine relevante Aufgabe erfüllen.

3.5 Chat

Java-Package: net.packets.chat

Dieses Package enthält alle Paket-Klassen die mit dem Nachrichtenaustausch unterhalb den Spielern zu tun haben. Dazu gehört ein Paket, welches dem Spieler ermöglicht Nachrichten an den Server zu versenden, ein Paket welches der Server dem Spieler zurückschickt als Bestätigung das seine Nachricht angekommen ist und ein Paket welches der Server benutzen kann um Nachrichten an die Spieler zu versenden.

3.5.1 Packet: CHAT_MESSAGE_TO_SERVER

Allgemein

Code	CHATS
Klasse	PacketChatMessageToServer.java
Absender	Client
Empfänger	Server

Klassenvariablen

chatmsg

Beschreibung	Ein String der die Nachtricht des Spieler enthält welche gesendet werden soll.
Validierung	Ist vorhanden. Die Nachricht darf nicht leer sein und darf höchstens 100 Zeichen enthalte.

timestamp

Beschreibung	Ein String der die Zeit abspeichert wann die Nachricht erstellt wurde.
Validierung	Ist vorhanden. Der String darf nicht leer sein und muss extended Ascii sein.

receiver

Beschreibung	Ein String der den Empfänger speichert falls ein Spieler nur ein einen Spieler eine Nachricht senden möchte.
Validierung	Ist vorhanden. Der String darf nicht leer sein und muss extended Ascii sein. Will der Spieler an alle Spieler in der Lobby eine Nachricht senden, wird der String auf "0" gesetzt.

Verarbeitung. Zuerst wird geprüft ob sich der Spieler in der Spielerliste vorhanden ist. Dann wird geprüft ob der Spieler sich in einer Lobby befindet. Ist alles erfüllt nimmt der Server das Paket mit der Nachricht und der Zeit und speichert die fertige Nachricht in der Stringvariablen fullmessage. Danach wird ein neues CHAT_MESSAGE_TO_CLIENT Paket erstellt welches dann mittels LobbyID vom spieler oder receivervariable verschickt wird.

Beispiel. Möchte ein Spieler eine Nachricht an andere Spieler senden zum Beispiel "hallo" muss er in der Konsole <C hallo> schreiben. Das

Paket mit der Nachricht wird an den Server geschickt. Der Server nimmt die Informationen von Spielername, Zeit und Nachricht welche anschliesend in soch einen String verpackt wird<['Spielername'-'Zeit'] hallo> , der mittels CHAT_MESSAGE_TO_CLIENT Paket weiterverschickt wird an die Spieler.

3.5.2 Packet: CHAT_MESSAGE_TO_CLIENT

Allgemein

Code	CHATC
Klasse	PacketChatMessageToClient.java
Absender	Server
Empfänger	Client

Klassenvariablen

chatmsg

Beschreibung	Ein String der die endgültige Nachtricht enthält welcher vom Server Verschickt wird un beim Spieler nur noch auf der Konsole ausgegeben werden muss
Validierung	Ist vorhanden. Die Nachricht darf nicht leer sein und darf höchstens 130 Zeichen enthalte. 100 Zeichen für die Nachricht selbst und 30 Zeichen für den Benutzernamen und die Zeit.

Verarbeitung. Der Server erstellt das Paket mit dem endgültigen String. Dieses Paket wird an den Spieler geschickt und wird dann in der Konsole ausgegeben.

Beispiel. Der Server erhält ein CHAT_MESSAGE_TO_SERVER Paket welche keine Fehler aufweist. Daraufhin wird ein CHAT_MESSAGE_-TO_CLIENT Paket erstellt mir dem String <['Spielername'-'Zeit'] hallo> welcher dann an die jeweiligen Spieler geschickt wird. Das Paket muss den String entnehmen und nochmals prüfen ob der ganze String nicht zulange ist. Dannach wird der String in der Konsole des Spieles ausgegeben.

3.5.3 Packet: CHAT_MESSAGE_STATUS

Allgemein

Code	CHATN
Klasse	PacketChatMessageStatus.java
Absender	Server
Empfänger	Client

Klassenvariablen

status

Beschreibung	Ein String der Informationen darüber enthält ob das Verlassen der Lobby funktioniert hat. Ist dies der Fall ist der String „OK“. Sonst enthält er eine entsprechende Fehlerbeschreibung.
Validierung	Ist vorhanden. Nur extended ASCII.

Verarbeitung.Es wird überprüft ob die empfangenen Daten mit «OK» beginnen. Ist dies nicht der Fall, wird die Fehlermeldung die in status enthalten ist oder eine Auflistung der Fehler, die bei der Validierung von status auftraten ausgegeben.

Beispiel. Ein Spieler wollte eine Nachricht an seine Mitspieler in der Lobby senden. Er schickt die Nachricht ab. Kurz darauf wird in seiner eigenen Konsole ein Fehler angezeigt, dass die Nachricht nicht erfolgreich verschickt werden konnte mit dem entsprechenden Fehlermeldung. Der Spieler kann dann nochmals eine Nachricht schicken. Falls keine Fehlermeldung erscheint, weiss der Spieler, das seine Mitspieler die Nachricht bekommen haben.

4 Aussicht

4.1 Geplante Packete

Weiter, spezifisch für unser Spiel, geplante Packete sind:

1. GAME_START
    - Startet ein Spiel einer Lobby und startet die Game-Logic
    - Wird von einem Spieler der Lobby ausgelöst
    - Wird vom Server verarbeitet und danach die Informationen an die Spieler der Lobby geschickt.
2. MOVE
    - Überträgt die aktuellen koordinaten vom Spieler an den Server
    - Der Server validiert dann die Koordinaten und überträgt sie an die anderen Spieler in der Lobby
3. HIT_BLOCK
    - Wird vom Client an den Server geschickt und überträgt Informationen über das Buddel verhalten des Spielers
    - Der Server validiert dann die Aktion und überträgt die Blockinformationen an die anderen Spieler in der Lobby
    - Der Server teilt dem Spieler auch mit, ob der Block eine Fragezeichenbox, Gold oder welche Art des Steines es ist.
4. MISTERY_ITEM
    - Informationen vom Server an den Client wenn der Client eine Fragezeichen Box geöffnet hat. Dazu gehören zum Beispiel Informationen über wie lange die Spieler eingefroren sind, wie viele Herzen ein Spieler bekommen hat und so weiter.
    - Der Spieler verarbeitet dies dann und setzt die Information grafisch um.

5. HEART_STATUS

    - Statusanfrage vom Spieler an den Server über den aktuellen Stand der Herzen anzufragen.
    - Der Server antwortet mit einem Status Paket und lässt den Spieler den Herzstand wissen.

6. PLAYER_LOST

    - Information vom Server an alle Spieler, dass ein Spieler ausgeschieden ist.

7. PLAYER_POINTS

    - Anfrage vom Client über den aktuellen Punktestand aller Spieler in der Lobby, inklusive des eigenen.
    - Der Server erstellt ein Antwortpacket mit der Punktzahl aller Spieler in der Lobby.

8. ENDGAME

    - Anfrage vom Client um ein Spiel zu beenden. Ausserdem ein Informationspacket vom Server an den Spieler um mitzuteilen, dass das Spiel vorüber ist.
    - Der Server beendet danach das Spiel und verarbeitet alle Informationen sowie einen Highscore.

9. BROADCAST

    - Befehl, um an alle Spieler des Servers eine Nachricht zu senden.
    - Der Server versendet dann diese Nachricht an alle Spieler.

List of Figures