Hexxagon Netzwerkstandard

Standardisierungsdokument zum Einzelprojekt **Hexxagon** im Softwaregrundprojekt 2019/2020.

Version 0.6.1 vom 03.09.2019

Inhaltsverzeichnis

Einleitung

Bei diesem Dokument handelt es sich um den Netzwerkstandard für das rundenbasierte Mehrspielerspiel Hexxagon, welches im Rahmen des Softwaregrundprojekts 2019/2020 als Einzelabgabe implementiert werden muss. Der Netzwerkstandard definiert hierbei wie die Kommunikation zwischen den Clients und dem Server ablaufen muss.

Der Netzwerkstandard stellt eine Erweiterung des Lastenhefts dar und stellt sicher, dass alle Clients, welche standardkonform implementiert sind, Spielpartien mit dem zur Verfügung gestellten Server durchführen können.

Definitionen

Einige Begriffe welche in diesem Dokument verwendet werden, haben in einem anderen Kontext eventuell eine andere Bedeutung. Deshalb sollen nachfolgend alle Begriffe, welche eine spezielle Bedeutung innerhalb dieses Dokuments haben, definiert werden.

Begriff	Definition
Client	Ein Client ist ein Programm. Ein Client kann eine Verbindung zu einem Server aufbauen. Ein Client kann Nachrichten an einen Server senden und Nachrichten von einem Server empfangen. Ein Client stellt das Spiel über eine graphische Oberfläche dar. Ein Client gehört im Rahmen dieses Dokuments zu einem Spieler, sobald eine Spielpartie gestartet wird.
Server	Ein Server kommuniziert mit einem Client. Ein Server kommuniziert mit 1-n Clients gleichzeitig. Ein Server verwaltet Client Verbindungen, Lobbies und Spielpartien.
Lobby	Eine Lobby stellt die Vorstufe zu einer Spielpartie dar. Clients können sich mit einer Lobby verbinden um eine Spielpartie zu starten. Im Kontext dieses Dokuments können sich maximal zwei Clients mit einer Lobby verbinden. Clients die durch das Starten einer Spielpartie die Lobby verlassen, werden als Spieler bezeichnet, siehe Definition Client / Spieler.
Teilnehmer	Ein Teilnehmer ist ein Client der mit einer Lobby interagiert. Dieser kann einer Lobby beitreten, sich innerhalb einer Lobby befinden und die Lobby verlassen.
Spieler	Ein Spieler nimmt an einer Spielpartie teil.
Spielpartie	Bei einer Spielpartie handelt es sich um die Durchführung des Spiels Hexxagon. Dies bedeutet es spielen zwei Spieler solange nach den Spielregeln des Spiels Hexxagon, bis ein Spieler die Spielpartie gewinnt.
Mehrspielerspiel	Bei einem Mehrspielerspiel handelt es sich um ein Spiel welches mehr als einen Spieler hat. Im Fall von Hexxagon hat eine Spielpartie genau zwei Spieler.
rundenbasiert	Bei Hexxagon handelt es sich um ein rundenbasiertes Mehrspielerspiel. Dies bedeutet, dass ein Spieler nach dem anderen Aktionen in einer Spielpartie ausführt. Das Gegenteil eines rundenbasierten Spieles wäre zum Beispiel ein Mehrspieler-Rennspiel, dort führen Spieler innerhalb einer Spielpartie gleichzeitig Aktionen aus.
WebSocket-Protokoll	Das WebSocket Protokoll ermöglicht die Kommunikation zwischen Client und Server über ein Netzwerk, zum Beispiel LAN (Local Area Network) oder über das Internet. Das WebSocket Protokoll wurde in RFC 6455 spezifiziert. Für weitere Informationen zum WebSocket Protkoll wird hier auf WebSocket - Wikipedia verwiesen. Für die Programmiersprache Java sind diverse Implementierungen des WebSocket Protokolls verfügbar. Im Rahmen dieses Dokuments wird sich in Beispielen auf die TooTallNate - Java-WebSocket Implementierung bezogen.

Begriff	Definition
JSON	Bei JSON handelt es sich um ein textbasiertes Format um Daten plattformunabhängig zu übertragen. Alle Nachrichten welche in diesem Dokument spezifiziert werden sind im JSON Format spezifiziert und werden als String über eine WebSocket Verbindung übertragen. Für weitere Informationen zum JSON Format wird auf JavaScript Object Notation - Wikipedia verwiesen.
Nachricht	Eine Nachricht wird im JSON Format als String vom Client an den Server oder vom Server an 1-n Clients gesendet.
Strike	Unter einem Strike wird das Senden einer ungültigen Nachricht verstanden. Eine Nachricht ist genau dann ungültig, wenn die Nachricht: - semantisch inkorrekt ist (Beispiel: es fehlt das Attribut messageType innerhalb der Nachricht) - zum gesendeten Zeitpunkt nicht zulässig ist (Beispiel: Client sendet CreateNewLobby Nachricht nachdem der Client bereits einer Lobby beigetreten ist) Wenn ein Client zu viele Strikes bekommt, dann wird die Verbindung zum Client getrennt.
MAX_STRIKE_COUNT	Die maximale Anzahl an Strikes die ein Client bekommen darf, bevor der Server die Verbindung zum Client trennt.

Nachrichten

Ein Client kommuniziert mit einem Server über das WebSocket-Protokoll. Die Nachrichten, die zwischen Client und Server geschickt werden, sind im JSON Format. Dieses Kapitel spezifiziert welche Nachrichtenarten es gibt, wann und von wem die Nachrichten gesendet werden und wie die Nachrichten strukturiert sind.

Nachrichtenarten

Server-Nachrichten

Server-Nachrichten, sind die Nachrichten, welche vom Server an den Client gesendet werden.

- Welcome
- AvailableLobbies
- LobbyCreated
- LobbyJoined
- LobbyStatus
- GameStarted
- GameStatus
- Strike

Client-Nachrichten

Client-Nachrichten, sind die Nachrichten, welche vom Client an den Server gesendet werden.

- GetAvailableLobbies
- CreateNewLobby
- JoinLobby
- LeaveLobby
- StartGame
- GameMove
- LeaveGame

Spezifikation der Datentypen

MessageType

Jede Nachricht die vom Client zum Server, bzw. vom Server zum Client, geschickt wird, muss ein Attribut vom Typ MessageType besitzen. Der Datentyp MessageTyp ist ein Enum, welcher alle Nachrichtentypen aufführt.

```
public enum MessageType {
    Welcome,
    GetAvailableLobbies,
    AvailableLobbies,
    CreateNewLobby,
    LobbyCreated,
    JoinLobby,
    LobbyJoined,
    LobbyStatus,
    LeaveLobby,
    StartGame,
    GameStarted,
    GameStatus,
    GameMove,
    LeaveGame,
    Strike
}
```

UUID

Beim Datentyp UUID handelt es sich um ein java.util.UUID Objekt.

Eine UUID besitzt als String beispielsweise folgende Form:

52adce64-cffc-4df9-acc2-78877fed78fd

String

Beim Datentyp String handelt es sich um ein java.lang.String Objekt.

Integer

Beim Datentyp Integer handelt es sich um ein java.lang.Integer Objekt.

Lobby

Beim Datentyp Lobby handelt es sich um folgende Java Klasse:

```
public class Lobby {

    public UUID lobbyId;
```

```
    public String lobbyName;

    public UUID playerOne;
    public UUID playerTwo;

    public String playerOneUserName;
    public String playerTwoUserName;

    public Date creationDate;
    public Boolean isClosed;

}
```

Das Attribut lobbyId ist vom Typ java.util.UUID und hält die eindeutige Indentifikationsnummer der Lobby.
Das Attribut lobbyName ist vom Typ java.lang.String und hält den Namen der Lobby.
Das Attribut playerOne ist vom Typ java.util.UUID und hält die eindeutige Identifikationsnummer von einem Client welcher mit der Lobby verbunden ist.
Das Attribut playerTwo ist vom Typ java.util.UUID und hält die eindeutige Identifikationsnummer von einem Client welcher mit der Lobby verbunden ist.
Das Attribut playerOneUserName ist vom Typ java.lang.String und hält den Benutzername des Clients welcher in playerOne eingetragen ist.
Das Attribut playerTwoUserName ist vom Typ java.lang.String und hält den Benutzername des Clients welcher in playerTwo eingetragen ist.
Das Attribut creationDate ist vom Typ java.util.Date und hält den Zeitstempel vom Zeitpunkt der Erstellung der Lobby.
Das Attribut isClosed ist vom Typ java.lang.Boolean und besitzt den Wert false solange die Lobby existiert, sobald der Wert true ist, wird damit signalisiert, dass die Lobby nicht mehr vorhanden ist.

TileStateEnum

Jedes Feld auf dem Spielbrett kann einen von vier Zuständen besitzen:

- Das Feld ist frei
- Das Feld ist durch einen Spielstein von Spieler 1 besetzt
- Das Feld ist durch einen Spielstein von Spieler 2 besetzt
- Das Feld ist blockiert, was bedeutet, es kann kein Spielstein auf dieses Feld gesetzt werden

Der Datentyp TileStateEnum bildet diese Zustände als Enum ab und ist wie folgt spezifiziert:

```
public enum TileStateEnum {
    FREE,
    PLAYERONE,
    PLAYERTWO,
    BLOCKED
}
```

TileEnum

```
public enum TileEnum {
    TILE_1,
    TILE_2,
    TILE_3,
    ...
    TILE_59,
    TILE_60,
    TILE_61
}
```

Es wurde eine verkürzte Abbildung des Enums gewählt, da es trivial sein sollte das Schema fortzusetzen.

Board

Um das Spielfeld abzubilden wird der Datentyp Board verwendet. Das Spielbrett besitzt 61 Felder, welche jeweils einen von vier Zuständen annehmen können. Die Zustände sind in TileStateEnum definiert.

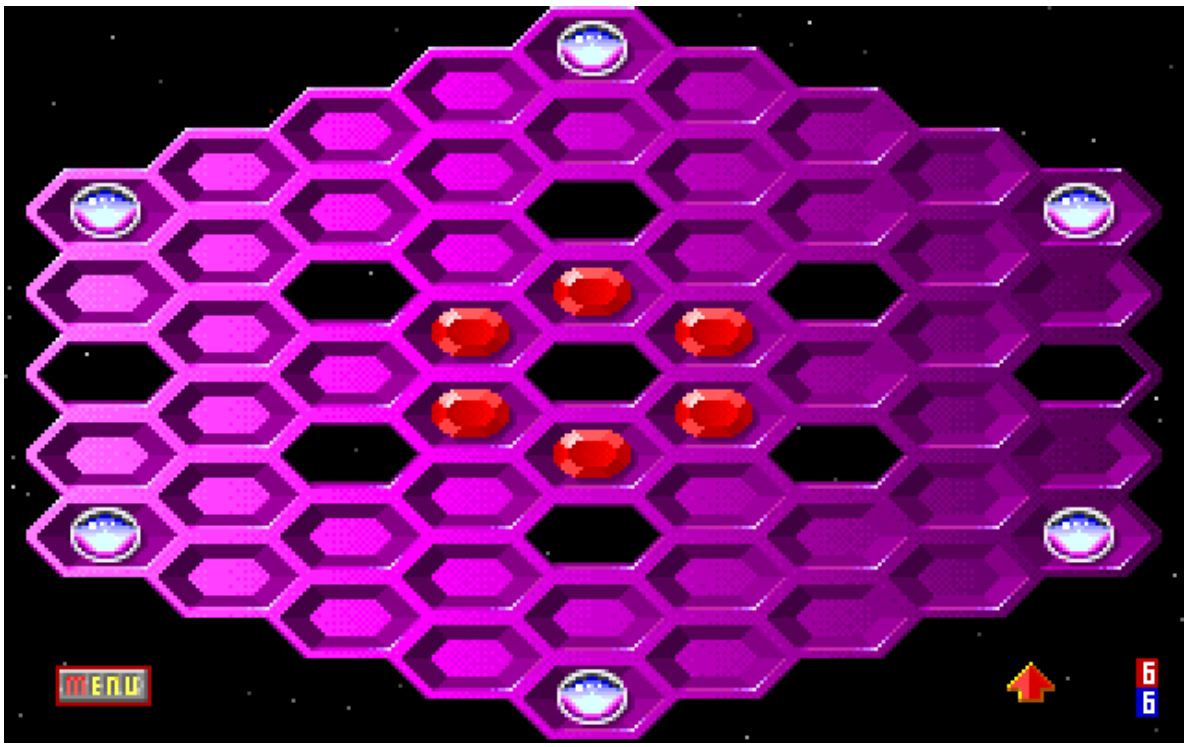
Das Spielfeld wird durch die Java Klasse Board abgebildet und ist wie folgt spezifiziert:

```
public class Board {

    public HashMap<TileEnum, TileStateEnum> tiles;

}
```

Der Startzustand des Spielbretts sieht wie folgt aus:



Die entsprechende JSON Darstellung der Board Klasse ist die folgende:

```
{
  "tiles": {
    "TILE_1": "PLAYERTWO",
```

```
    "TILE_2": "FREE",
    "TILE_3": "BLOCKED",
    "TILE_4": "FREE",
    "TILE_5": "PLAYERTWO",
    "TILE_6": "FREE",
    "TILE_7": "FREE",
    "TILE_8": "FREE",
    "TILE_9": "FREE",
    "TILE_10": "FREE",
    "TILE_11": "FREE",
    "TILE_12": "FREE",
    "TILE_13": "FREE",
    "TILE_14": "BLOCKED",
    "TILE_15": "FREE",
    "TILE_16": "BLOCKED",
    "TILE_17": "FREE",
    "TILE_18": "FREE",
    "TILE_19": "FREE",
    "TILE_20": "FREE",
    "TILE_21": "FREE",
    "TILE_22": "PLAYERONE",
    "TILE_23": "PLAYERONE",
    "TILE_24": "FREE",
    "TILE_25": "FREE",
    "TILE_26": "FREE",
    "TILE_27": "PLAYERTWO",
    "TILE_28": "FREE",
    "TILE_29": "BLOCKED",
    "TILE_30": "PLAYERONE",
    "TILE_31": "BLOCKED",
    "TILE_32": "PLAYERONE",
    "TILE_33": "BLOCKED",
    "TILE_34": "FREE",
    "TILE_35": "PLAYERTWO",
    "TILE_36": "FREE",
    "TILE_37": "FREE",
    "TILE_38": "FREE",
    "TILE_39": "PLAYERONE",
    "TILE_40": "PLAYERONE",
    "TILE_41": "FREE",
    "TILE_42": "FREE",
    "TILE_43": "FREE",
    "TILE_44": "FREE",
    "TILE_45": "FREE",
    "TILE_46": "BLOCKED",
    "TILE_47": "FREE",
    "TILE_48": "BLOCKED",
    "TILE_49": "FREE",
    "TILE_50": "FREE",
    "TILE_51": "FREE",
    "TILE_52": "FREE",
    "TILE_53": "FREE",
    "TILE_54": "FREE",
    "TILE_55": "FREE",
    "TILE_56": "FREE",
    "TILE_57": "PLAYERTWO",
    "TILE_58": "FREE",
    "TILE_59": "BLOCKED",
```

```
    "TILE_60": "FREE",
    "TILE_61": "PLAYERTWO"
  }
}
```

Hinweis: Es wird eine java.util.HashMap vom Server verwendet, um die einzelnen Kacheln und deren Zustand zu speichern. Eine HashMap ist nicht geordnet, deshalb ist es sehr wahrscheinlich, dass die Reihenfolge, von *TILE_1* bis *TILE_61*, nicht wie in der obigen Auflistung ist.

Spezifikation der Nachrichten

Welcome

Nachdem ein Client eine WebSocket Verbindung zum Server aufgebaut hat, wird dem Client eine Welcome Nachricht vom Server geschickt. Die Welcome Nachricht ist die erste Nachricht die gesendet wird. Jeder Client bekommt eine eindeute Identifikationsnummer als UUID vom Server zugeordnet, diese Identifikationsnummer ist Teil der Welcome Nachricht und muss bei jeder weiteren Nachricht vom Client an den Server, zur Identifikation, mitgeschickt werden.

```
{
    messageType: MessageType.Welcome,
    userId: UUID,
    welcomeMessage: String
}
```

GetAvailableLobbies

Um als Client eine Übersicht aller aktuell verfügbaren Lobbies zu bekommen, muss der Client eine GetAvailableLobbies Nachricht an den Server schicken, dieser antwortet, nach Erhalt der GetAvailableLobbies Nachricht, mit einer AvailableLobbies Nachricht.

Die GetAvailableLobbies Nachricht ist wie folgt spezifiziert:

```
{
    messageType: MessageType.GetAvailableLobbies,
    userId: UUID
}
```

AvailableLobbies

Nachdem der Server eine GetAvailableLobbies Nachricht empfangen hat, sendet er dem Client, welcher die Nachricht gesendet hat, eine AvailableLobbies Nachricht. Die AvailableLobbies Nachricht beinhaltet alle aktuell, auf dem Server, verfügbaren Lobbies.

```
{
    messageType: MessageType.AvailableLobbies,
    userId: UUID,
    availableLobbies: Lobby[]
}
```

Beim Attribut availableLobbies handelt es sich um ein Array vom Datentyp Lobby, siehe Abschnitt: Spezifikation der Datentypen.

Eine beispielhafte AvailableLobbies Nachricht könnte wie folgt aussehen:

```
{
    "messageType": "AvailableLobbies",
    "userId": "301a0c7d-3cba-4642-8b52-53a6f189a81b",
    "availableLobbies": [
        {"lobbyId": "f07bac68-9a43-4d9d-887c-475e19e7c4f9",
         "lobbyName": "Max Mustermann Lobby",
         "playerOne": "293076c0-bbed-4bfd-8cb2-24b5654eb0d6",
         "playerOneUserName": "Max Mustermann",
         "playerTwo": "131773ee-d789-4901-8be4-022e8e97e2e6",
         "playerTwoUserName": "Melanie Musterfrau",
         "creationDate": "Jul 15, 2019 1:46:54 PM",
         "isClosed": false},
        {...}
    ]
}
```

CreateNewLobby

Um eine neue Lobby zu erstellen, sendet der Client dem Server eine CreateNewLobby Nachricht, welche wie folgt spezifieziert ist:

```
{
    messageType: MessageType.CreateNewLobby,
    userId: UUID,
    lobbyName: String
}
```

LobbyCreated

Wenn der Server eine CreateNewLobby Nachricht von einem Client empfangen hat, dann sendet er diesem Client eine LobbyCreated Nachricht mit der Identifikationsnummer der neu erstellten Lobby zu. Die LobbyCreated Nachricht ist wie folgt spezifiziert:

```
{
    messageType: MessageType.LobbyCreated,
    userId: UUID,
    lobbyId: UUID
    successfullyCreated: Boolean
}
```

Das Attribut successfullyCreated ist vom Typ java.lang.Boolean und hat den Wert true, wenn die Lobby erfolgreich erstellt wurde, andernfalls ist der Wert false.

JoinLobby

Um einer Lobby beizutreten muss der Client dem Server eine JoinLobby Nachricht an den Server senden. Diese Nachricht muss die Identifikationsnummer des Clients, die Identifikationsnummer der Lobby und einen frei wählbaren Benutzername des Spielers beinhalten. Die JoinLobby Nachricht ist wie folgt spezifiziert:

```
{
    messageType: MessageType.JoinLobby,
    userId: UUID,
    lobbyId: UUID,
    userName: String
}
```

LobbyJoined

Nachdem ein Client eine JoinLobby Nachricht gesendet hat, versucht der Server den Client zu einer Lobby hinzuzufügen. Egal ob das Hinzufügen zur Lobby erfolgreich war oder nicht, wird dem Client eine LobbyJoined Nachricht gesendet. In der LobbyJoined Nachricht wird die Identifikationsnummer der Lobby und ein boolean gesendet, der boolean bestitzt den Wert true, wenn der Client erfolgreich zur Lobby hinzugefügt wurde, andernfalls wird false gesendet. Die LobbyJoined Nachricht ist wie folgt spezifiziert:

```
{
    messageType: MessageType.LobbyJoined,
    userId: UUID,
    lobbyId: UUID,
    successfullyJoined: boolean
}
```

LobbyStatus

Immer wenn sich am Zustand einer Lobby etwas ändert (Client tritt Lobby bei, Client verlässt Lobby, Lobby wurde vom Server geschlossen), wird vom Server eine LobbyStatus Nachricht an alle mit der Lobby verbundenen Clients gesendet. Die LobbyStatus Nachricht ist wie folgt spezifiziert:

```
{
    messageType: MessageType.LobbyStatus,
    userId: UUID,
    lobbyId: UUID,
    lobby: Lobby
}
```

Anmerkung: Falls die Lobby vom Server geschlossen wird, da eine Spielpartie gestartet wurde, wird keine LobbyStatus Nachricht gesendet.

LeaveLobby

Ein Client kann die Verbindung zu einer Lobby trennen, hierfür sendet er eine LeaveLobby Nachricht an den Server. Die LeaveLobby Nachricht ist wie folgt spezifiziert:

```
{
    messageType: MessageType.LeaveLobby,
    userId: UUID,
    lobbyId: UUID
}
```

StartGame

Der Client, der in der Lobby als playerOne eingetragen ist, kann das Spiel starten, insofern die Lobby über zwei verbundenen Clients verfügt. Hierzu sendet der Client, der in der Lobby als playerOne eingetragen ist, eine StartGame Nachricht. Die StartGame Nachricht ist wie folgt spezifiziert:

```
{
    messageType: MessageType.StartGame,
    userId: UUID,
    lobbyId: UUID
}
```

GameStarted

Nachdem der Server für eine Lobby eine GameStart Nachricht erhalten hat, wird vom Server eine neue Spielpartie erstellt. Nachdem der Server diese Spielpartie erstellt hat sendet er an alle, der zur Spielpartie gehörenden Lobby, verbundenen Clients eine GameStarted Nachricht. Im Anschluss wird die Lobby geschlossen.

Die GameStarted Nachricht ist wie folgt spezifiziert:

```
{
    messageType: MessageType.GameStarted,
    userId: UUID,
    gameId: UUID,
    creationDate: Date
}
```

GameStatus

Die GameStatus Nachricht wird vom Server an alle in der aktuellen Spielpartie teilnehmenden Clients gesendet. Die GameStatus Nachricht hält den aktuellen Spielzustand und wird immer gesendet wenn sich etwas am Spielzustand ändert.

Änderungen des Spielzustands sind:

- Das Spiel wurde frisch gestartet
- Ein Spieler (Client) hat einen Spielzug gemacht
- Ein Spieler hat das Spiel gewonnen
- Das Spiel wurde beendet

Die GameStatus Nachricht ist wie folgt spezifiziert:

```
{
    messageType: MessageType.GameStatus,
    userId: UUID,
    gameId: UUID,
    playerOne: UUID,
    playerTwo: UUID,
    playerOneUserName: String,
```

```
    playerTwoUserName: String,
    playerOneLeft: Boolean,
    playerTwoLeft: Boolean,
    playerOnePoints: Integer,
    playerTwoPoints: Integer,
    board: Board,
    turn: Integer,
    lastMoveFrom: TileEnum,
    lastMoveTo: TileEnum,
    creationDate: Date,
    actionDate: Date,
    activePlayer: UUID,
    tie: Boolean,
    winner: UUID,
    isClosed: Boolean
}
```

Das Attribut playerOneLeft ist vom Datentyp java.lang.Boolean und gibt an, ob playerOne die Spielpartie verlassen hat.
Das Attribut playerTwoLeft ist vom Datentyp java.lang.Boolean und gibt an, ob playerTwo die Spielpartie verlassen hat.
Das Attribut playerOnePoints ist vom Datentyp java.lang.Integer und gibt an, wie viele Punkte (Spielsteine) playerOne besitzt.
Das Attribut playerTwoPoints ist vom Datentyp java.lang.Integer und gibt an, wie viele Punkte (Spielsteine) playerTwo besitzt.
Das Attribut board ist vom Datentyp Board und hält den aktuellen Zustand des Spielbretts.
Das Attribut turn ist vom Datentyp java.lang.Integer und gibt die aktuelle Runde an.
Das Attribut lastMoveFrom ist vom Datentyp TileEnum und ist zu Beginn einer neuen Spielpartie null.
Das Attribut lastMoveTo ist vom Datentyp TileEnum und ist zu Beginn einer neuen Spielpartie null.
Das Attribut creationDate ist vom Datentyp java.util.Date und hält den Zeitstempel der Erstellung der Spielpartie.
Das Attribut actionDate ist vom Datentyp java.util.Date und hält den Zeitstempel der letzten Aktion in der Spielpartie.
Das Attribut activePlayer ist vom Datentyp java.util.UUID und hält die Identifikationsnummer des Clients welcher den nächsten Spielzug ausführen darf.
Das Attribut tie ist vom Datentyp java.util.Boolean und gibt an, ob die Spielpartie mit einem Unentschieden beendet wurde.
Das Attribut winner ist vom Datentyp java.util.UUID und hält die Identifikationsnummer des Clients welcher die Spielpartie gewonnen hat. Insofern die Spielpartie noch nicht durch einen Spieler (Client) gewonnen wurde ist der Wert null.
Das Attribut isClosed ist vom Datentyp java.lang.Boolean und besitzt den Wert false, solange die Partie läuft. Wenn der Wert true ist, dann ist die Spielpartie vorbei. Ein Spielende kann mehrere Gründe haben:

- Ein Spieler (Client) hat die Partie gewonnen
- Ein Spieler (Client) hat die Verbindung geschlossen
- Ein Spieler (Client) hat mehr als für MAX_STRIKE_COUNT eingestellte ungültige Nachrichten gesendet und die Client Verbindung wurde vom Server geschlossen
- Der Server hat die Partie vorzeitig beendet, zum Beispiel durch das Herunterfahren des Servers oder einen internen Serverfehler.

GameMove

Die GameMove Nachricht wird vom Client an den Server gesendet. Wenn ein Spieler seinen Zug ausgeführt hat, so sendet er den Spielzug mit einer GameMove Nachricht an den Server. Der Server validiert den Zug des Spielers und sendet dann an alle Clients innerhalb der Spielpartie eine GameStatus Nachricht.

Die GameMove Nachricht ist wie folgt spezifiziert:

```
{
    messageType: MessageType.GameMove,
    userId: UUID,
    gameId: UUID,
    moveFrom: TileEnum,
    moveTo: TileEnum
}
```

LeaveGame

Die LeaveGame Nachricht wird vom Client an den Server gesendet um eine laufende Spielpartie zu verlassen.

Die LeaveGame Nachricht ist wie folgt spezifiziert:

```
{
    messageType: MessageType.LeaveGame,
    userId: UUID,
    gameId: UUID
}
```

Strike

Der Server sendet dem Client eine Strike Nachricht, wenn der Client eine Nachricht gesendet hat, welche:

- semantisch nicht korrekt ist (Beispiel: es fehlt das Attribut messageType)
- zum Zeitpunkt des Sendens der Nachricht nicht zulässig ist (Beispiel: Client sendet eine CreateNewLobby Nachricht während der Client mit einer Lobby verbunden ist)

Bekommt ein Client mehr als die in MAX_STRIKE_COUNT festgelegte Anzahl an Strikes, so wird die Vvalueerbindung zum Client getrennt.

Die Strike Nachricht ist wie folgt spezifiziert:

```
{
    messageType: MessageType.Strike,
    userId: UUID,
    strikeCount: Integer,
    maxStrikeCount: Integer
}
```

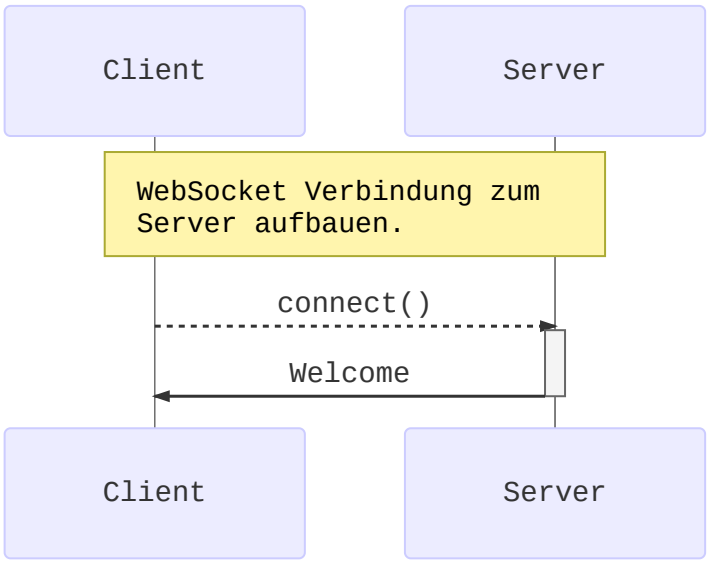
Das Attribut strikeCount vom Datentyp java.lang.Integer gibt an wie viele Strikes der Client bereits bekommen hat.
Das Attribut maxStrikeCount vom Datentyp java.lang.Integer gibt an wie viele Strikes der Client maximal bekommen darf, bis der Server die Verbindung zum Client trennt.

Beispiel: Angenommen der Wert in maxStrikeCount ist 10, dann trennt der Server die Verbindung zum Client, sobald dieser den zehnten Strike bekommen hat und nicht erst beim elften Strike.

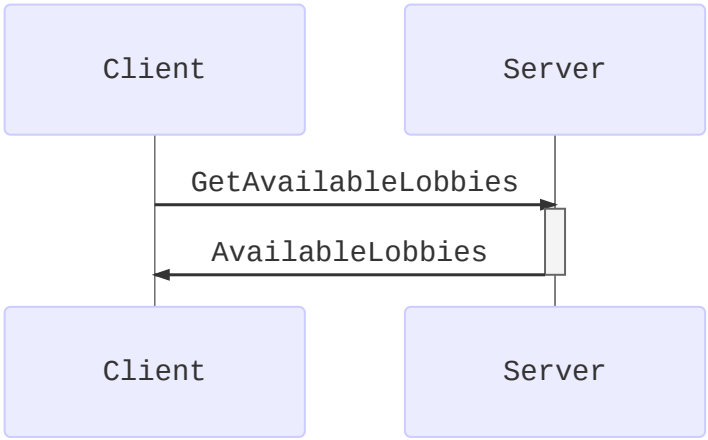
Sequenzdiagramme

Die nachfolgenden Sequenzdiagramme beschreiben, welche Nachrichten in welcher Reihenfolge geschickt werden dürfen.

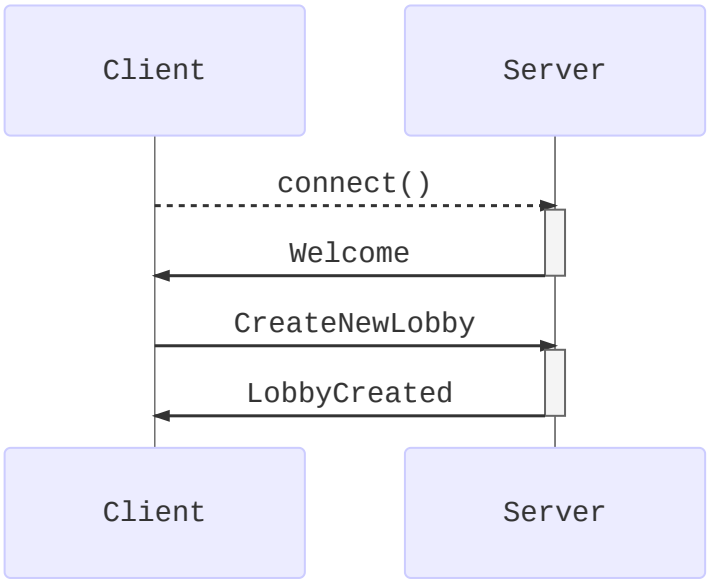
Verbindung zum Server aufbauen



Lobby Übersicht anfordern



Erstellen einer Lobby

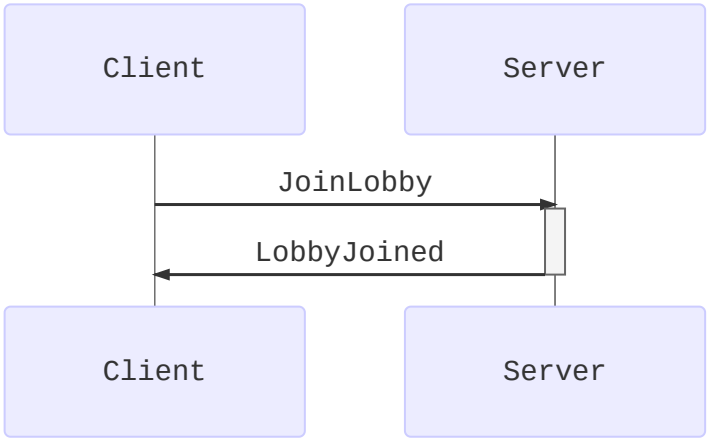


Einer Lobby beitreten

Im folgenden werden alle Szenarien abgebildet, welche beim Beitritt zu einer Lobby auftreten können.

Einer Lobby ohne Teilnehmer beitreten

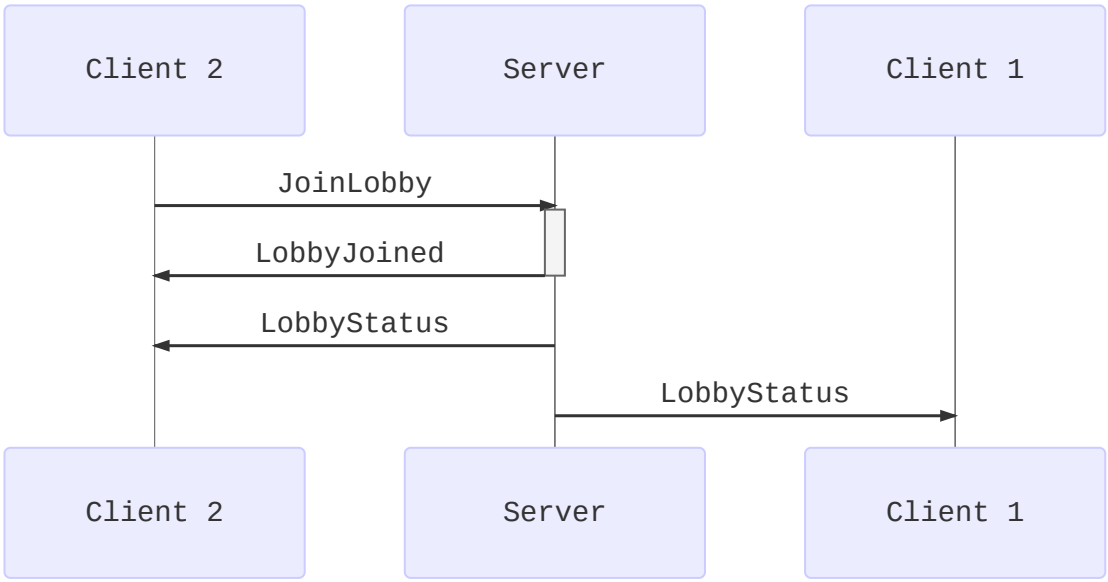
Um einer Lobby ohne Teilnehmer beizutreten müssen folgende Nachrichten gesendet werden:



Eine Lobby kann leer sein, dies ist der Fall wenn die Lobby gerade erst erstellt wurde. Verlassen alle Teilnehmer einer Lobby die Lobby, dann wird die Lobby zerstört und nicht leer.

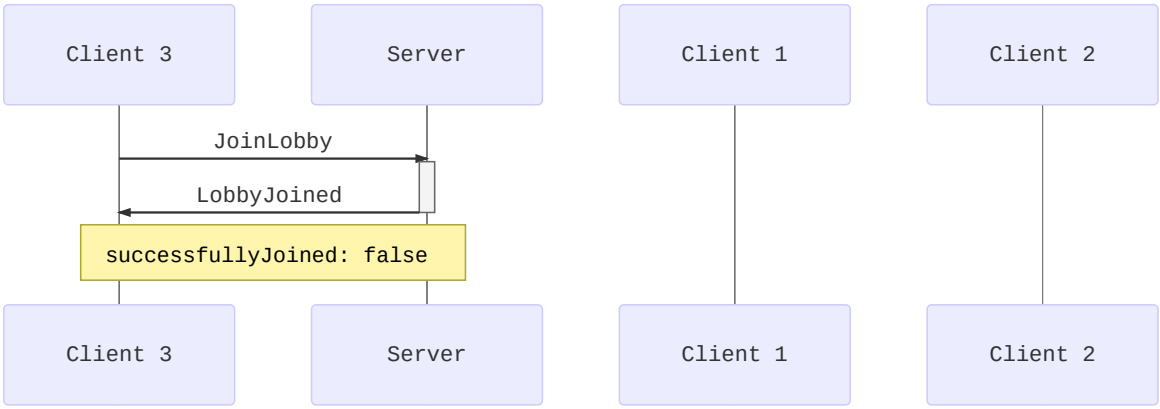
Einer Lobby mit einem Teilnehmer beitreten

Um einer Lobby mit einem Teilnehmer beizutreten müssen folgende Nachrichten gesendet werden:



Einer Lobby mit zwei Teilnehmern beitreten

Beim Versuch einer Lobby mit zwei Teilnehmern beizutreten werden folgende Nachrichten gesendet:



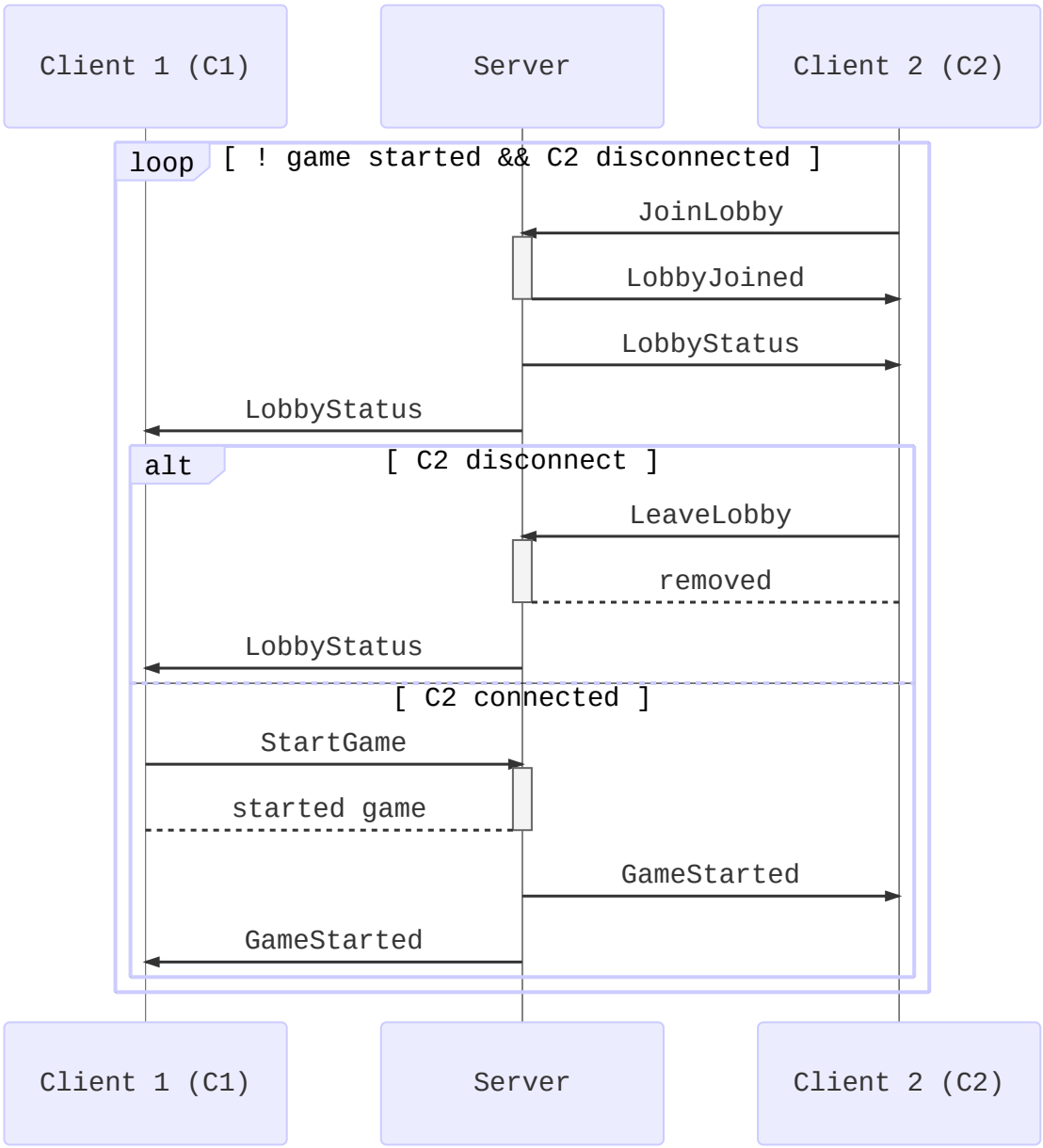
Ablauf innerhalb einer Lobby

Man ist der erste Teilnehmer in der Lobby

Wenn man der erste Teilnehmer in einer Lobby ist, muss gewartet werden, bis ein zweiter Teilnehmer der Lobby beitritt. Sobald ein zweiter Teilnehmer der Lobby beigetreten ist, kann das Spiel gestartet werden.

Der zweite Teilnehmer kann die Lobby allerdings jederzeit verlassen, hierfür sendet er eine LeaveLobby Nachricht an den Server. In diesem Fall muss erneut auf den Beitritt eines zweiten Teilnehmers gewartet werden.

Man kann ein Spiel nur starten, wenn man der erste Teilnehmer in einer Lobby ist.



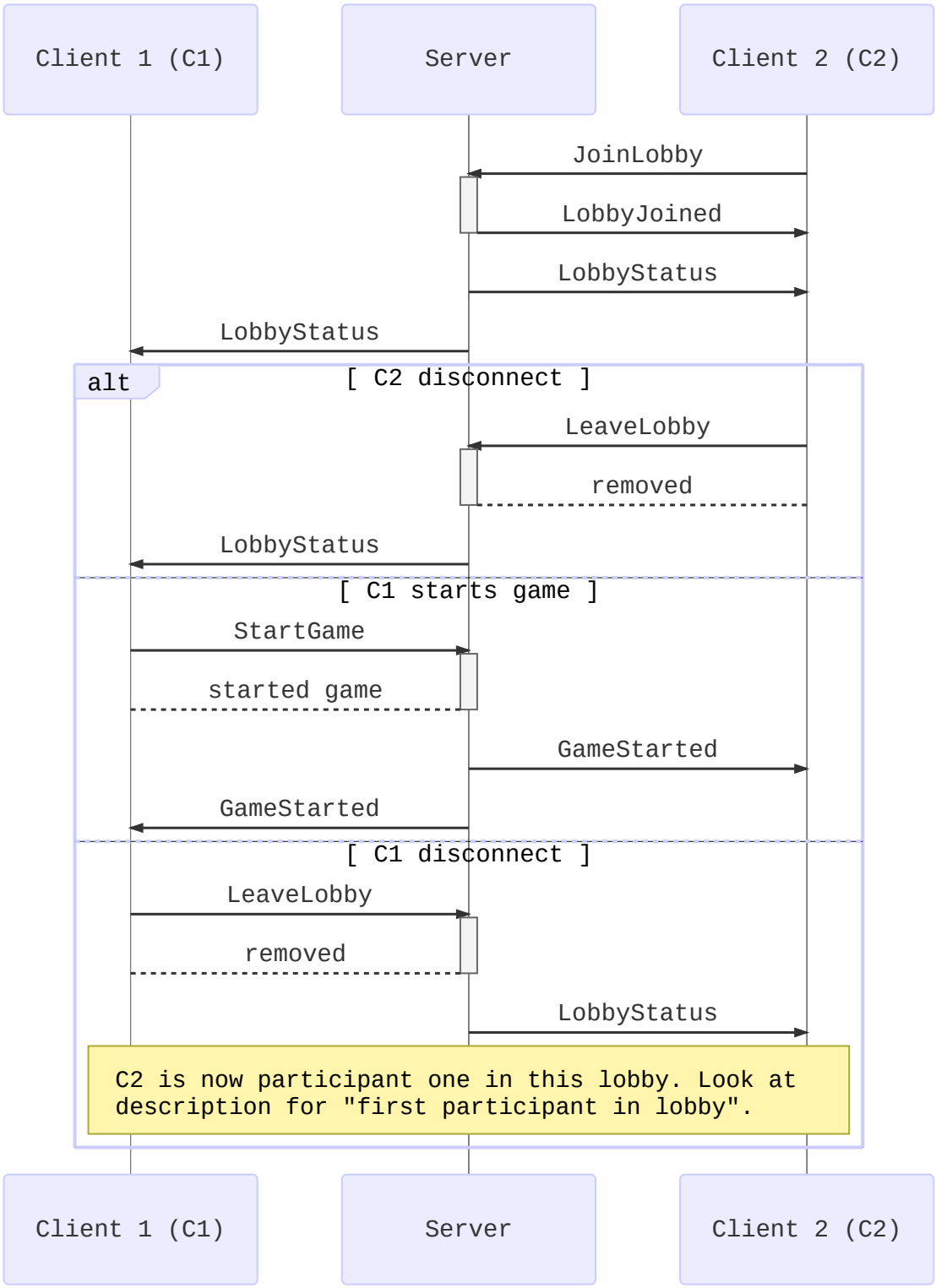
Man ist der zweite Teilnehmer einer Lobby

Wenn man der zweite Teilnehmer in einer Lobby ist, so muss gewartet werden bis der erste Teilnehmer der Lobby das Spiel startet.

Als zweiter Teilnehmer einer Lobby kann man die Lobby verlassen, solange keine StartGame Nachricht vom ersten Teilnehmer gesendet wurde.

Sendet der erste Teilnehmer eine StartGame Nachricht, so beginnt die Spielpartie.

Es ist möglich, dass der erste Teilnehmer die Lobby verlässt während man als zweiter Teilnehmer mit der Lobby verbunden ist. In diesem Fall wird man zum ersten Teilnehmer der Lobby. Betrachten Sie hierfür das Sequenzdiagramm für "Man ist der erste Teilnehmer einer Lobby".



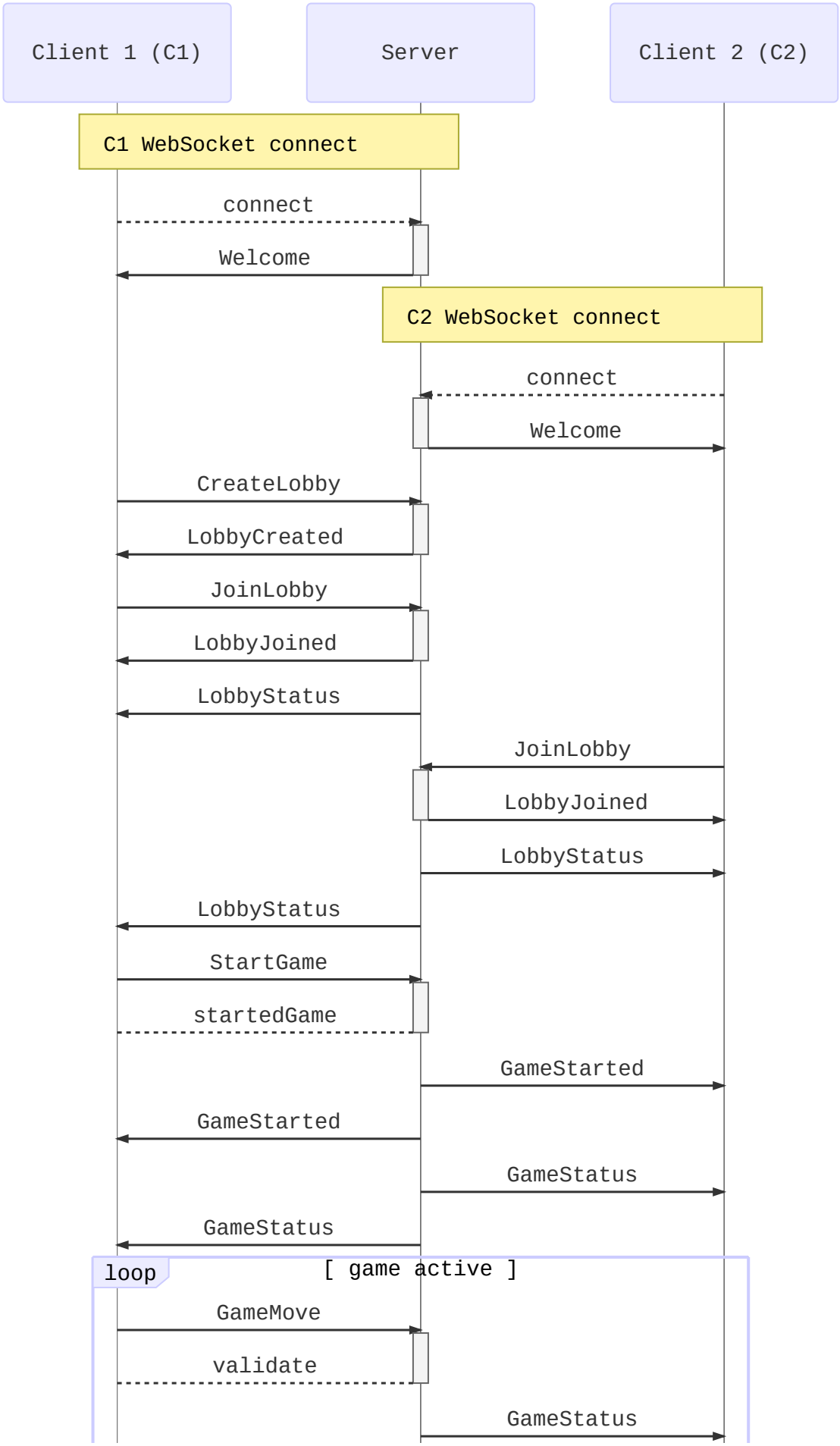
Beispiele zum Spielablauf

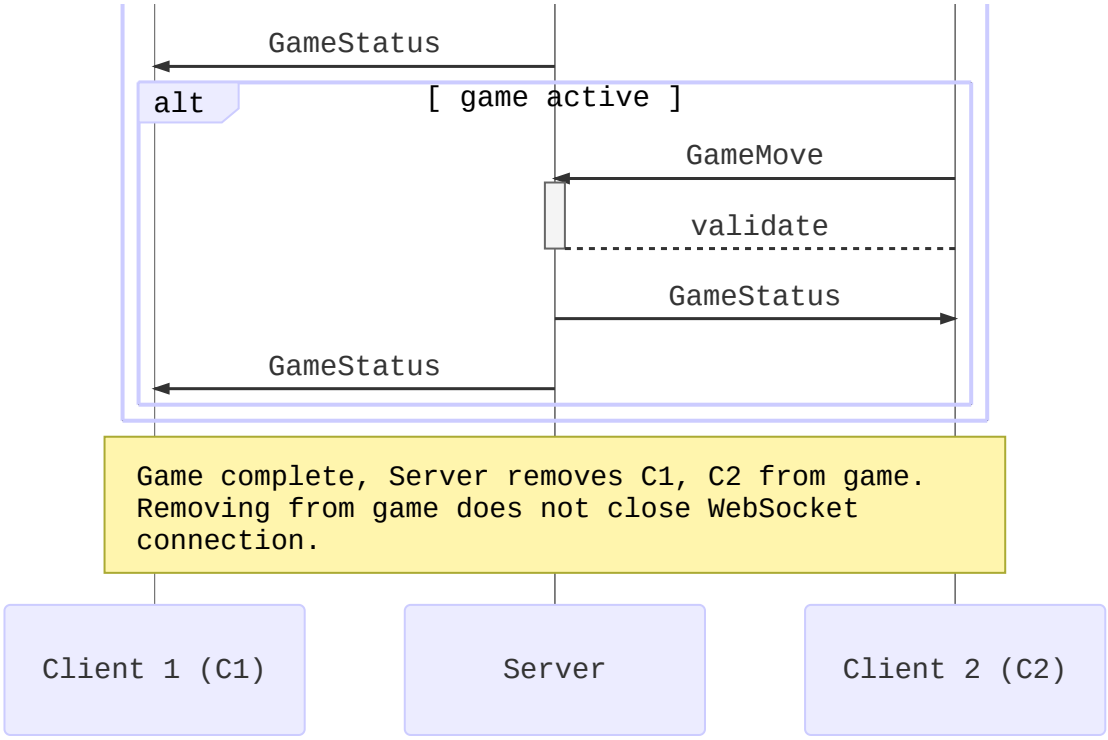
Optimalfall

Das nachfolgende Sequenzdiagramm stellt einen optimalen Spielablauf dar. Das optimal bezieht sich darauf, dass keine unvorhersehbaren Ereignisse eintreten, wie zum Beispiel:

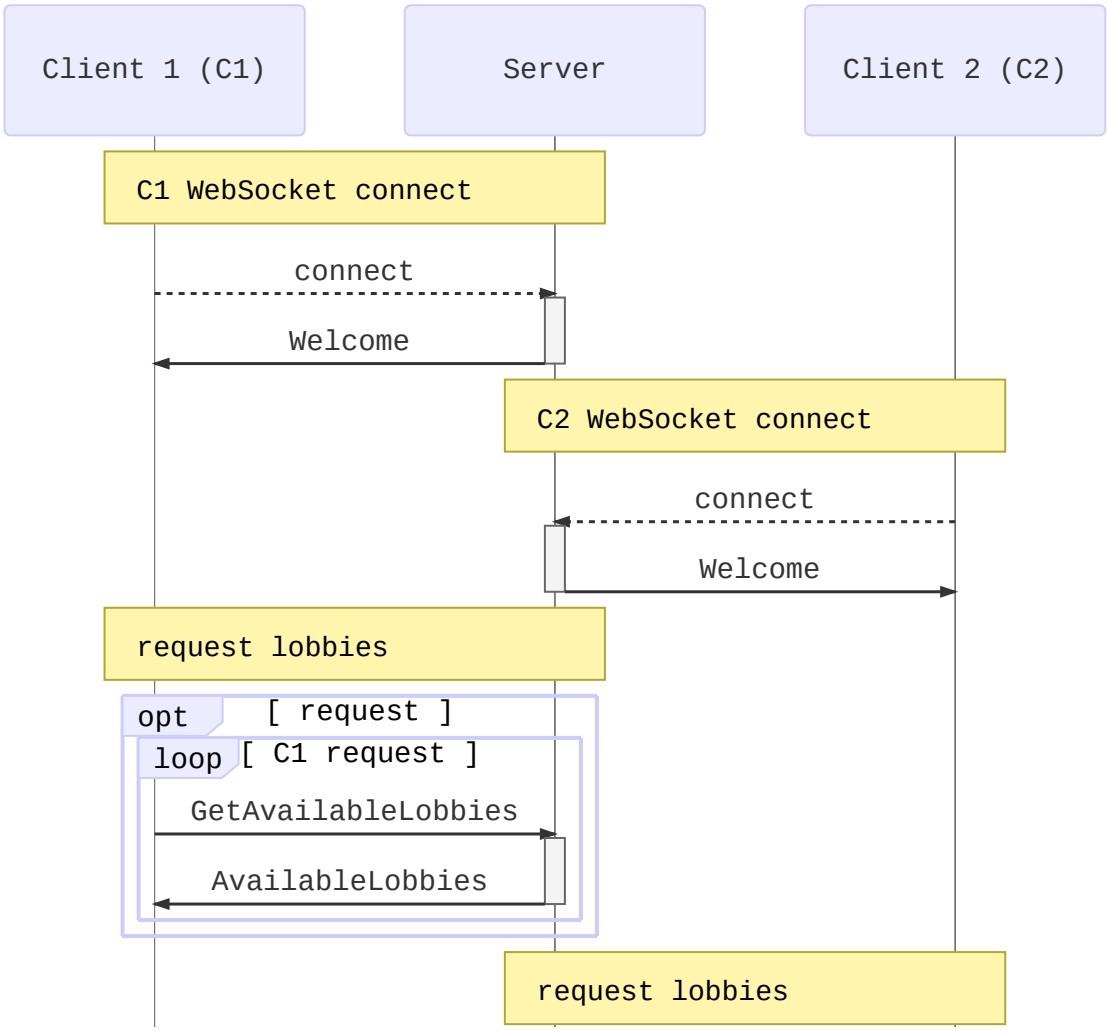
- Verbindung zum Client bricht ab
- Client verlässt Lobby nachdem er beigetreten ist

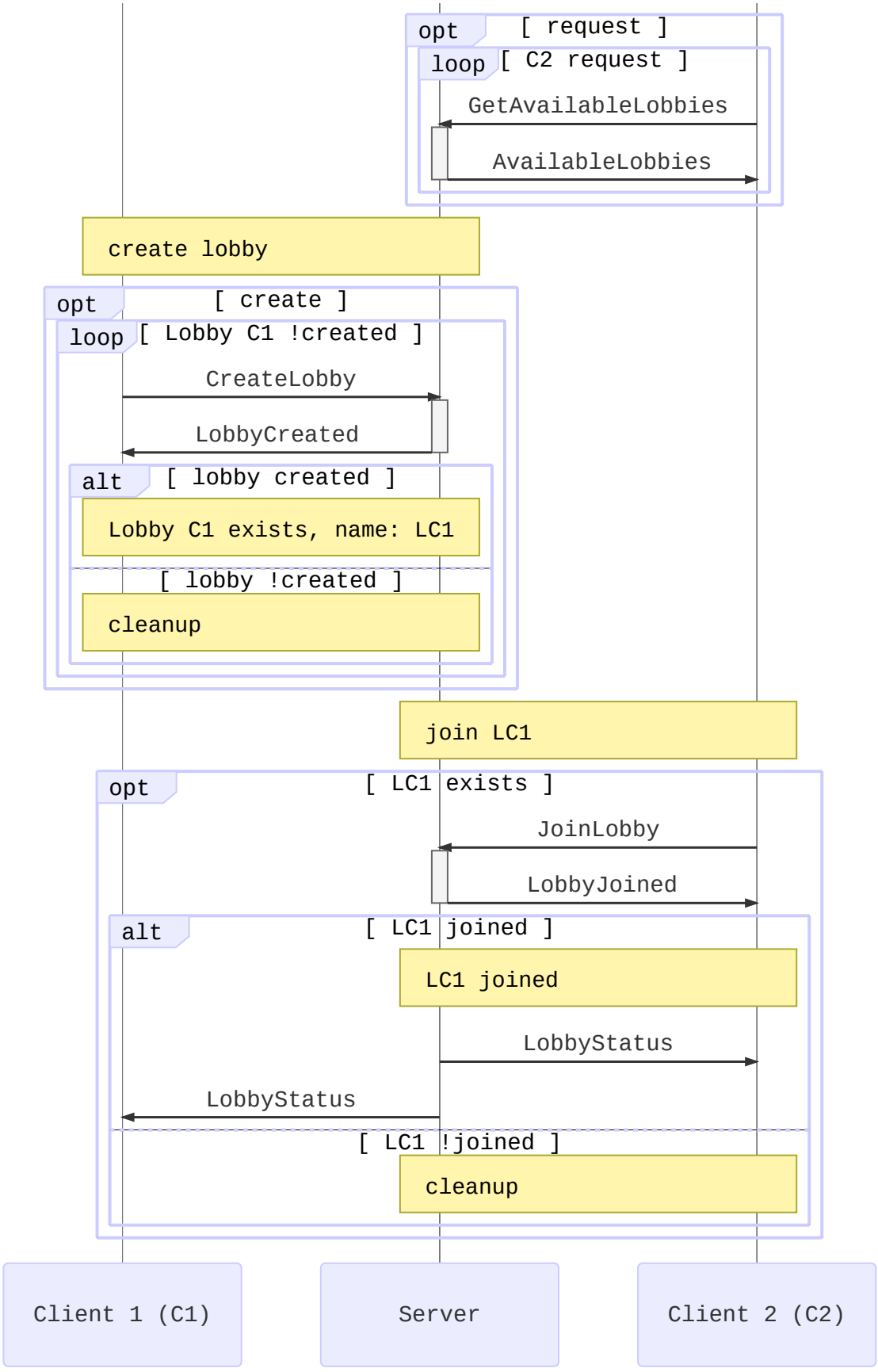
- Client verlässt laufende Spielpartie
- Client sendet unzulässige Nachrichten
- Interne Serverfehler





Komplexer Spielablauf




opt
[request]
loop
[C2 request]
GetAvailableLobbies
AvailableLobbies
create lobby
opt
[create]
loop
[Lobby C1 !created]
CreateLobby
LobbyCreated
alt
[lobby created]
Lobby C1 exists, name: LC1
[lobby !created]
cleanup
join LC1
opt
[LC1 exists]
JoinLobby
LobbyJoined
alt
[LC1 joined]
LC1 joined
LobbyStatus
LobbyStatus
[LC1 !joined]
cleanup
Client 1 (C1)
Server
Client 2 (C2)