VSY Konzept
Ligretto

Rolf Koch, Sandro Ropelato, Michael Schwarz,
Andreas Ruckstuhl, Christof Würmli

24. April 2011

Inhaltsverzeichnis

1 Aufgabenstellung

Der Verlauf eines Ligretto-Spiels soll simuliert werden können. Um echten Zufall ins Spiel zu bringen, soll das Spiel über mehrere Maschinen verteilt werden.

- Auf dem **Eingangsserver** lassen Sie einen Webserver laufen. Ein POST-Request auf den Server initiiert eine Ligretto-Simulation und liefert einen Link über den das Resultat per Polling abgholt werden kann. Das Resultat der Simulation wird unter diesem Link als XHMTL-Seite präsentiert. Während der Berechnung wird eine Seite mit dem Text *in Bearbeitung* angezeigt.
- Eingangsparameter: Ausgangslage des Spiels (Verteilung der Karten) und implizit die Anzahl Spieler. Rückgabewert: Punkteverteilung für die Spieler
- Dieser Eingangsserver sei bereits komplett definiert. Sie erhalten die Daten aus dem POST-Request in der von Ihnen gewünschten Datenstruktur zur Weiterverarbeitung.
- Um das **Mischeln und Verteilen** der Karten müssen Sie sich nicht kümmern. Der Eingangsserver liefert ja bereits die Spielsituation genau vor Beginn des Spiels. Zu diesem Zeitpunkt sind die Karten verteilt und Spezialregeln (neue Karten bei >= 30 Punkte) sind bereits erfüllt. Aber Sie müssen dafür Sorgen, dass die **Clients die nötigen Informationen vom Eingangsserver** erhalten. Dies ist Abhängig von der von Ihnen gewählen Implementation.
- Ihnen stehen für die Simulation eine bestimmte **Anzahl Rechner** zur Verfügung. Die Rechner haben sich vorgängig beim Eingangsserver gemeldet, so dass er eine Liste der verwendbaren Clients hat und damit die Anzahl der verwendbaren Rechner kennt. Sie brauchen sich nicht darum zukümmern wie sich die Clients beim Server anmelden.
- Die **CPU-Leistung** ist nicht zu optimieren.
- Die **Berechnungszeit** ist nicht zu optimieren.
- Beschreiben Sie den **Algorithmus** vollständig (Ax im Bewertungsmassstab). Sie beginnen ab dem Moment wo Sie die Daten vom Eingangsserver erhalten (in der von Ihnen definierten Datenstruktur). Das erste wird wohl die Verteilung der Daten auf die Client-Rechner seind.
- Beschreiben Sie alle verwendeten **Datenstrukturen** inklusive der Deklaration in Java oder C/C++.
- Beschreiben Sie die **Schnittstellen** als RPC x-File oder RMI-Interface. Nur die Schnittstellen die während dem Spiel benutzt werden sind hier verlangt.
- Die Anzahl Ziffern (1-10) und Frontfarben (4 verschiedene) sind unveränderliche Konstanten
- Wie **skaliert** das System mit Zunahme der Anzahl Spieler?

- Die restlichen Bewertungskriterien (wie Rechnerausfall etc.) können Sie weglassen
- Die **Spielstrategie** können Sie an eine Funktion KI auslagern. Wenn Sie also aus spieltechnischen Ueberlegungen eine Karte nicht legen wollen, obwohl es möglich wäre, so kann diese Entscheidung durch diese KI-Funktione ermittelt werden. Sie müssen allerdings die für die Entscheidung nötigen Informationen dieser Funktion zur Verfügung stellen. D.h. Sie müssen sich überlegen welche Informationen nötig sind und woher diese Informationen besorgt werden. Aber um die Implementation der Spieletheorie müssen Sie sich nicht kümmern.

2 Spielbeschreibung

Abbildung 1: Ligretto

Ligretto ist ein Spiel, bei dem es darum geht, seinen eigenen Ligretto Stapel abzubauen, bevor dies ein Gegner erreicht. Der Ligretto Stapel besteht aus 10 verdeckten zufälligen Karten aus dem Kartenvorrat des Spielers. Der Kartenvorrat eines Spielers (auch Deck genannt), besteht aus 40 Karten. Die Karten sind von 1 bis 10 nummeriert und haben die folgenden vier Farben: Rot, Gelb, Blau und Grün (Abbildung 2). Es gibt keine doppelten Karten, das heisst, jede Karte kommt nur genau einmal im Deck des Spielers sowie im ganzen Spiel vor.

Abbildung 2: Ligretto Karten

Um ein neues Spiel vorzubereiten, mischt jeder Spieler seinen Kartenstapel. Jeder Spieler zählt dann 10 Karten ab und legt diese verdeckt vor sich hin. Dies ist sein eigenes Ligretto Deck. Danach legt er 4 Karten von seinen restlichen Handkarten offen neben den Stapel. Sobald alle Spieler soweit sind, geht es los.

Der Spielablauf funktioniert nun für jeden Spieler wie folgt: Der Spieler

1. schaut die offen vor sich liegenden Karten an
2. prüft die Stapel an Karten in der Mitte des Spielfelds und
3. versucht, eine Karte zu finden, die um 1 höher ist als die oberste Karte eines Stapels sowie die selbe Farbe besitzt. Ist dies der Fall, so darf er seine Karte nehmen und oben auf diesen Stapel legen.
4. Wenn diese gelegte Karte eine der vier offen vor sich liegenden Karten gewesen ist, dann darf der Spieler die oberste Karte vom Ligretto Stapel aufdecken und anstelle der soeben gelegten Karte vor sich hin legen.
5. Wenn eine 1er Karte verfügbar ist, darf damit einen neuen Stapel in der Mite des Spielfelds eröffnet werden.
6. Wenn mit den offenen Karten keine Aktionen durchgeführt werden können, darf der Spieler von seinen restlichen Handkarten 3 Karten verdeckt abzählen, diese umdrehen und vor sich auf den Ablagestapel legen. Existiert noch kein Ablagestapel, so legt er die 3 Karten neben seine bereits vor sich liegenden Karten und eröffnet somit seinen Ablagestapel. Beim Ablagestapel darf nun jeweils die oberste Karte ebenfalls gespielt werden.
7. Sollte weiterhin keine Karte gelegt werden können, so wiederholt der Spieler Punkt 6 und legt immer 3 Karten oben auf den Ablagestapel, bis er keine Handkarten mehr in der Hand hält. Ist dies der Fall, so nimmt der Spieler den Ablagestapel wieder als Handkarten auf und fängt von vorne an.

Sobald ein Spieler die letzte Karte seines Ligretto Stapels aufgedeckt hat, ruft er *Ligretto Stop!*, das Spiel wird beendet und es wird **abgerechnet**.

Bei der Abrechnung bekommt jeder Spieler Punkte für sich selber. Dies läuft wie folgt ab:

1. Jeder Spieler zählt die noch vorhandenen Karten auf dem eigenen Ligretto Stapel. Für jede Karte auf dem Ligretto Stapel bekommt er **zwei Minuspunkte**.
2. Die Karten auf dem Spielfeld werden nun wieder nach Farbe auf der Rückseite sortiert. Jeder Spieler erhält seine Karten zurück und zählt diese. Für jede gelegte Karte bekommt der Spieler **einen Pluspunkt**.
3. Die Punkte werden nun notiert und jeder Spieler bereitet sich für die nächste Runde vor.

3 Problemanalyse

Als Aufgabe gilt es den durch die Spielregeln festgelegten Ablauf anhand folgender Gesichtspunkte in einen Algorithmus umzusetzen:

1. Der Spielablauf muss den Spielregeln entsprechen und die KI-Algorithmen, welche die Mitspieler simulieren, sollten nur an Informationen über den Spielablauf gelangen, welche auch ein menschlicher Spieler gelangen würde.
2. Der Spielablauf sollte möglichst fair sein. D.h. bis auf den Zufall, der durch das Mischen der Karten ins Spiel gelangt, sollten alle Mitspieler gleichgestellt sein.
3. Die verwendeten Algorithmen und Schnittstellen sollten möglichst gut mit der Anzahl Mitspieler skalieren, unter Betrachtung der benötigten Rechenzeit und Netzwerkkapazität.

Folgende Punkte werden bei der Herleitung des Algorithmus nicht betrachtet:

1. Alle Mitspieler müssen sich an die Spielregeln halten, insbesondere in Teilen des Ablaufs, die im Spiel mit menschlichen Spielern nicht überwacht werden können. Dazu gehört z.B. auch das regelkonforme Ablegen der Handkarten beim suchen einer passenden Karte.
2. Die beiligten Geräte versenden nur Nachrichten an Geräte, mit denen sie laut dem Alogrithmus kommunizieren dürfen.
3. Die eingesetzte Soft- und Hardware darf keine Fehler aufweisen und der Netzerkverkehr darf nicht gestört oder unterbrochen werden (z.B. dürfen keine Pakete verloren gehen).

3.1 Systemgrenzen

Damit die Laufzeit eines Spiels nicht (oder nur schwach) mit der Anzahl Spieler wächst, muss die Anzahl verwendeter Computer linear mit der Anzahl Mitspieler wachsen können. Deshalb ist es notwendig und sinnvoll, für jeden Mitspieler mindestens einen separaten Prozess oder Thread zu verwenden.

Da ein einzelner Spieler jedoch zu jeder Zeit immer nur eine Aktion ausführen darf (gemäss Spielregeln), ist eine Aufteilung eines einzelenen Spielers in mehrere Prozesse oder Threads nicht sinnvoll.

Zur Überwachung des gesammten Spiels sowie dessen Aufbau und Abbau wird ein separater Prozess verwendet. Da der überwachende Prozess nur vor Spielbeginn und nach Beendigung des Spiels eine Aufgabe hat, dessen Komplexität mit der Anzahl Mitspieler wächst, ist eine Aufteilung dieser Aufgaben in mehrere Prozesse oder Threads zwar erwägenswert, aber nicht von oberster Priorität.

Zusammengefasst:

1. Ein Prozess zur Überwachung des Spiels (Aufbau und Abbau).
2. Ein Prozess pro Mitspieler, welche alle Aktionen des jeweiligen Mitspielers durchführt.

3.2 Verteilung

Bei der Betrachtung der Verteilung des Zustandes gibt es drei Arten von Objekten:

1. **Zustands-Objekte** verkörpern den veränderlichen Zustand eines Spiels. Die Eigenschaften dieser Objekte sind somit mutierbar. Für diese Objekte ist immer genau ein Prozess zuständig und diese Objekte werden auf andere Prozesse verschoben oder kopiert.
2. **Token-Objekte** werden zur Kommunikation verwendet und sind nicht veränderbar. Token-Objekte können zwischen den Prozessen verschoben werden, jedoch ist immer nur genau ein Prozess für ein jeweiliges Token-Objekt verantwortlich.
3. **Transiente Objekte** werden zu Kommunikationszwecken und zur Unterstützung der Algorithmen erstellt und aufbewahrt. Wenn ein transientes Objekt an einen anderen Prozess versendet wird, dann besitzen beide Prozess eine eigene Kopie des Objektes.

Die folgenden Token-Objekte können identifiziert werden:

1. Spielkaren: Zu beginn des Spieles wird für jede Spielkarte, welche in einen realen Ligretto-Spiel verwerdet würde, ein Spielkarten-Objekt erstellt.

Folgende Zustands-Objekte können identifiziert werden:

1. Handkarten-Stapel
2. Ausgelegte Karten vor dem Mitspieler
3. Stapel auf dem Spieltisch
4. Mitspieler
5. Überwachender Prozess

Zur Kommunikation werden folgende Transiente Objekte erstellt:

1. Start- und Stop-Signale des überwachenden Prozesses
2. Stop-Signal eines Mitspielers
3. Stapel mit gemischten Karten, welche vom überwachenden Prozess an die Mitspieler verteilt werden

3.3 Concurrency-Problem

3.3.1 Gleichzeitiges Legen einer Karte

Es kommt vor, dass zwei Spieler gleichzeitig versuchen, die selbe Karte auf den selben Stapel zu legen. Wie im echten Spiel gilt auch in der Spielsimlulation: **First come first served**. Dabei werden alle RMI-Methoden mit dem *synchronized*-Schlüsselwort synchronisiert.

3.3.2 Spielstop währenddem eine Karte gelegt wird

Wird das Spiel während einem Versuch, eine Karte auf den Stapel zu legen, beendet, darf die Karte gemäss Spielregeln nicht mehr gelegt werden. Dies kann erreicht werden, indem man die Erfolgsmeldung mit dem Spielstatus verknüpft. Ist das Spiel beim Aufruf der *legeKarte*-Methode beendet, wird in jedem Fall *false* zurückgegeben, beziehungsweise der legende Spieler interpretiert es als *false*.

3.4 Zuverlässigkeit

3.4.1 Verlorenes Netzwerkpaket

Da wir uns für den Einsatz von RMI entschieden haben, brauchen wir uns um Paketverlust, Korrektheit der Daten, etc. keine Gedanken zu machen. RMI benutzt TCP zur Kommunikation und stellt somit die korrekte Übermittlung einer Nachricht sicher.

3.4.2 Absturz eines Spielers

Stürzt ein Spieler während des Spiels ab, so beeinträchtigt dies den Verlauf des Spiels nicht. Um eine lange Antwortzeit beim Legen einer Karte zu vermeiden, werden ihn seine Nachbarn für weitere Anfrageversuche ignorieren.

3.4.3 Absturz des Servers

Da der Server für die Übermittlung der Ligretto-Stop-Nachricht verantwortlich ist, hätte ein Serverabsturz zur Folge, dass alle Spieler spielen, bis sie ihren Ligrettostapel abgearbeitet haben, und dann beim Aufruf der *ligrettoStop*-Methode erfahren, dass der Server keine Antwort mehr gibt. Der Spielausgang wäre damit verfälscht und nicht brauchbar.

3.5 Skalierbarkeit

Gemäss Spielregeln braucht es mindestens zwei Spieler für ein Ligrettospiel. Mit nur zwei Spielern ist die Chance jedoch hoch, dass das Spiel nicht zu Ende gespielt werden kann, da die Karten ungünstig liegen und damit ein Deadlock entsteht. Gegen oben existieren keine Grenzen. Da nur eine Kommunikation mit einer konstanten Anzahl Nachbarn erfolgt, bleibt die Netzwerkauslastung und der Rechenaufwand pro Spieler auch bei steigender Anzahl Gegner konstant. Je nach Implementation können Nachrichten, welche an alle Spieler gehen (*spielStart* und *spielEnde*) durch Grenzen der Netzwerkgeschwindigkeit etwas verspätet eintreffen.

4 Lösungsansatz

Client 1 hat vom Server Client 2 bis Client 7 als Nachbarn zugewiesen bekommen (Abbildung 3). Als erstes selektiert der Client 1 eine Karte und versucht diese bei sich selbst Legen. Gelingt dies nicht, versucht er die Karte bei einem seiner 6 Nachbarn zu legen, indem er ihnen die Karte übergibt und, falls sie gelegt werden konnte, eine Erfolgsmeldung (*true*) oder ansonsten eine Misserfolgsmeldung (*false*) zurückbekommt. Konnte er die selektierte Karte legen, wird diese gelöscht und eine neue Karte wird selektiert, mit welcher wieder von vorne begonnen wird. Das Beispiel von Client 1 ist representativ für alle Clients.

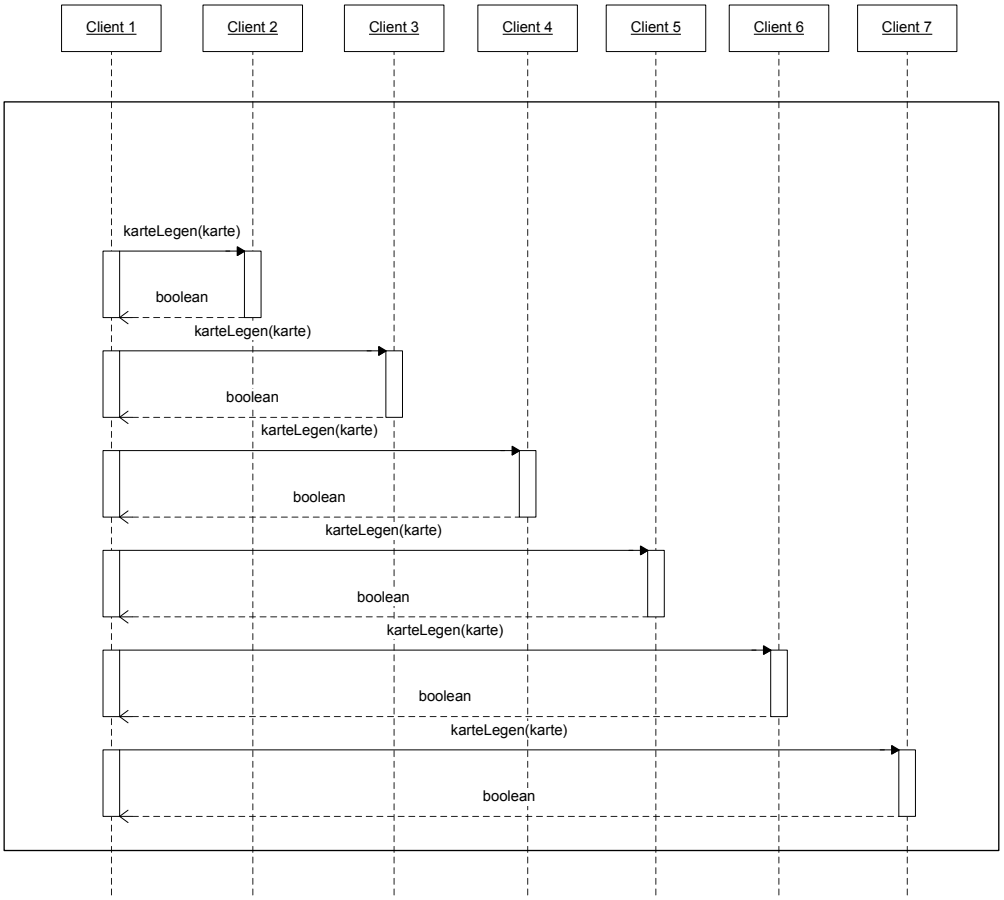

Abbildung 3: Sequenzdiagramm: Kartenlegen

Dadurch, dass nicht zuerst der Status der Spiel-Stapel der Nachbarn abgefragt werden muss und danach versucht wirde eine Karte zu lege, ist die Kommunikation auf den Versuch einfach eine Karte zu legen minimiert. Es wird also nur eine Karte übertragen und entweder "true"für den Erfolg oder "false"für den Misserfolg zurückgegeben. Zudem verwaltet jeder Spieler (Client) seine eigenen Spiel-Stapel vor sich. Diejenigen die er selbst

erstellt hat mit einer Eins. Der Spieler muss also nur maximal 4 Spiel-Stapel verwalten auf die andere Spieler (Clients) Zugriff haben. Auch ist die anzahl Spieler die auf diese Spiel-Stapel Zugriff haben beschränkt, indem jeder Spieler nur 6 Nachbarn bekommt, mit denen er spielen kann. Wenn diese Nachbarn Sinnvoll verteilt sind, also das heisst solche Spieler die Lokalisationsmässig nahe beieinander liegen auch Nachbarn sind, findet die Kommunikation nur lokal statt und somit wirde ebenfalls die Kommunikation über das globale Netz minimiert.

5 Spiel-Algorithmus

5.1 Strategie

Wir haben uns dazu entschieden das Spiel zu beginn durch einen Server starten zu lassen. Der Server übergibt alle nötigen Informationen den Playern und überlässt dann das Spiel den Playern unter sich.

Die Player kennen immer nur eine Limitierte Zahl von Nachbarn. Wir haben uns entschieden diese Nachbarn Zahl auf 6 Nachbarn zu begrenzen. Jeder Player kennt also sich selber und 6 seiner Nachbarn.

Die Organisation welcher Player welche Nachbarn kennt übernimmt der Server.

Er holt sich regelmässig den Player Zustand der andern Spieler um dort ebenfalls Karten ablegen zu können.

Sobald ein Spieler fertig ist meldet er Ligretto-Stop zurück an den Server

5.2 KI

Ist nicht direkt Teil unseres Konzepts, aber wir müssen der KI gewisse Informationen und Möglichkeiten zur Verfügung stellen damit sie überhaupt operieren kann.

Dazu gehören:

- Spiel-Start Signal
- Stapel auf denen sie Karten ablegen könnte
- Die Handkarten des Spielers
- Das Legen einer Karte auf einen Stapel (Erfolgreich oder nicht)
- Zusätzliche Informationen über die Sichtbaren Karten der Nachbarn.
- Spiel-Ende Signal

5.3 Spielablauf

Abbildung 4 zeigt das Verhalten des Servers gegenüber den Clients.

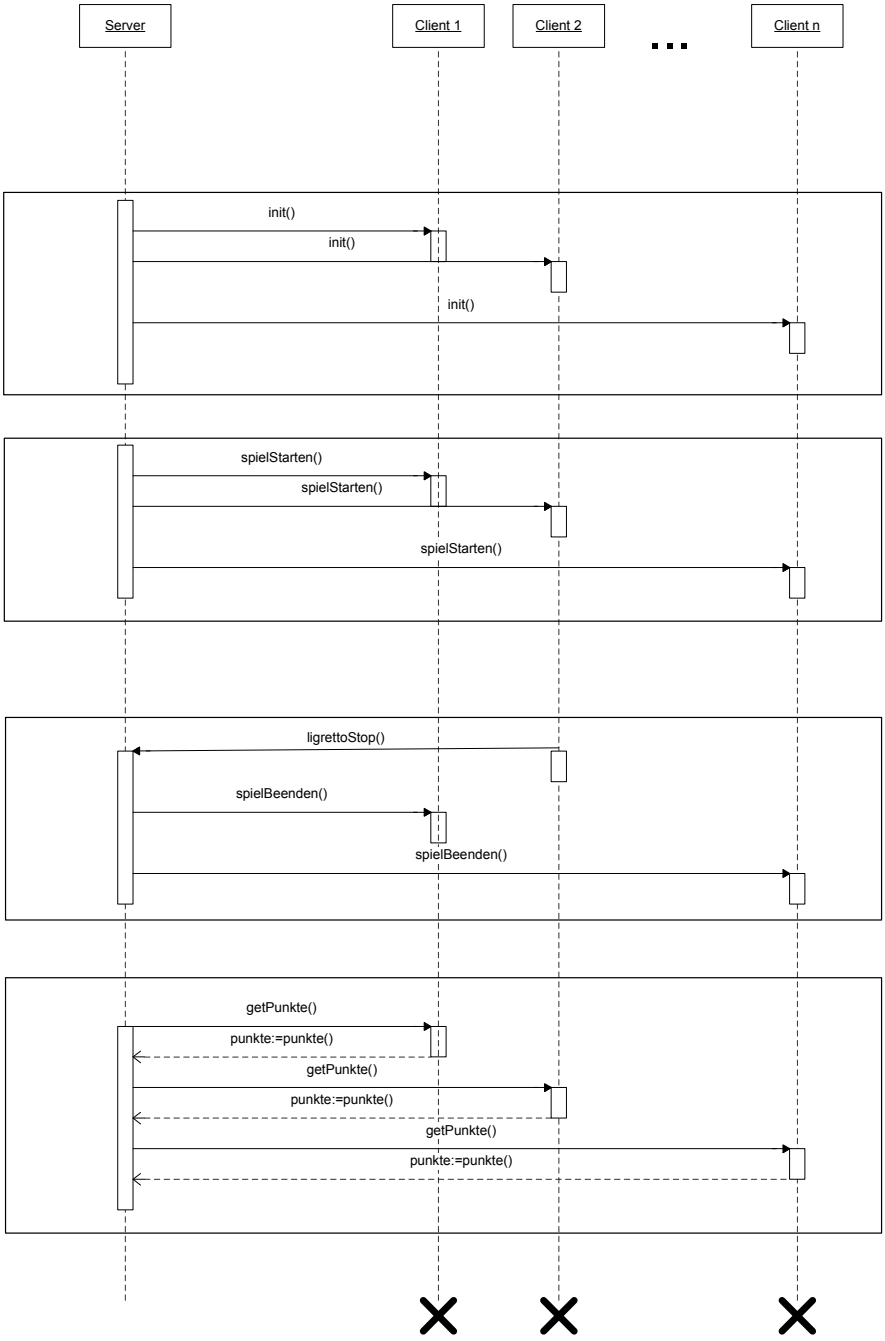

Abbildung 4: Sequenzdiagramm für den Spielablauf aus Serversicht

1. Der Server wird nur für den Start und das Ende des Spieles benötigt. Das Spiel selber läuft via Peer to Peer ab.
2. Der Server stellt zu beginn eine Verbindung zu allen Playern her. Dieser Verbindungsaufbau ist nicht Teil dieses Konzepts.
3. Der Server bekommt ein Set aus bereits gemischten Karten Sets à 40 Karten und verteilt davon jeweils ein Set an einen Player.
4. Der Server teilt jedem Player zudem mit, wer seine Nachbarn sind. Die Anzahl an Nachbarn haben wir auf Maximum 6, und Minimum 1 definiert. Sie wächst oder schrumpft je nach Anzahl Playern.
5. Sobald der Client Karten hat und seine Nachbarn kennt bereitet er sich für das Spiel vor. Sobald er bereit ist meldet er dem Server, dass er Ready ist.
6. Wenn der Server von allen Clients das Ready-Signal erhalten hat sendet er das Game-Start Signal an alle Players und zieht sich aus dem Spielgeschehen zurück und wartet bis der erste Client das Ligretto-Stop-Signal an ihn sendet.
7. Wenn das Spiel startet holt sich jeder Spieler bei seinen Nachbarn den aktuellen Zustand und fängt an zu spielen. Mit einer 1er-Karte eröffnet er bei sich selber einen neuen Stapel und er holt sich regelmässig bei allen bekannten Nachbarn deren aktuellen sichtbaren Spielzustand. Er holt dazu zum einen die vier offenen Karten neben dem Ligretto-Stapel, die oberste Karte des Ablagestapels und alle Stapel des jeweiligen Spielers. Er prüft dann für jeden Stapel des Nachbarn ob er eine Karte besitzt, die er darauf legen könnte.
8. Wenn der Player eine mögliche Option zum Karte ablegen gefunden hat versucht er diese auf diesen Stapel zu legen. Wenn er zu langsam war, bekommt er eine Too-Slow Fehlermeldung. Wenn die Karte gar nicht hätte gelegt werden dürfen bekommt er eine Illegal Karte Fehlermeldung.
9. Die KI zu programmieren ist wiederum nicht unsere Aufgabe in diesem Konzept. Wir liefern der KI nur die Möglichkeit sich Informationen von andern Playern zu holen.
10. Sobald ein Player die letzte Karte seines Ligretto-Stapels umdrehen konnte, meldet er dem Server Ligretto Stop.
11. Der Server meldet dieses Ligretto-Stop Signal weiter an alle teilnehmenden Players.
12. Wenn ein Player noch eine Karten lege Operation am laufen hat darf er diese noch zu ende führen. Danach stellt er seine Aktionen ein und errechnet seine erreichte Punktezahl. Er zählt dafür seinen Ligretto-Stapel und zählt dann alle noch nicht gelegten Karten und subtrahiert diese von ursprünglichen Deck-Grösse.
13. Er errechnet dann seine Punktezahl und meldet diese dem Server.

5.3.1 Karte legen

Abbildung 5 zeigt das Verhalten des Clients gegenüber seinen Nachbarn.

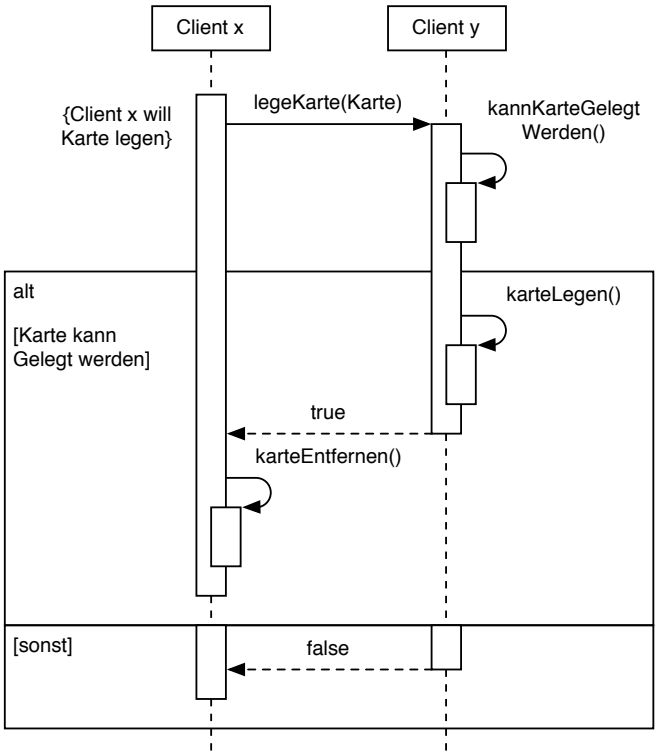

Abbildung 5: Sequenzdiagramm zum Legen einer Karte

5.4 Kommunikation

Bei der definition der Kommunikation muss gleichzeitig die Korrektheit und die Performance beachtet werden.

Der Ablauf lässt sich in folgende Abschnitte unterteilen:

1. Während dem **Aufbau** werden die Spielkarten durch den Server vorbereitet und an die registrierten Clients verteilt
2. Stop-Signal eines Mitspielers
3. Stapel mit gemischten Karten, welche vom Überwachenden Prozess an die Mitspieler verteilt werden

5.4.1 Vor Spielbeginn

Ausgangslage bei Spielbeginn ist folgende:

- Der Server kennt alle Spieler, die an dieser Spielrunde teilnehmen.
- Dem Server stehen eine entsprechende Anzahl gut gemischter Kartendecks zur Verfügung.
- Kommunikationsgrundlagen (Netzwerk, RMIRegistry, etc...) sind gegeben.

5.4.2 Spielinitialisierung

Zuerst initialisiert der Server alle ihm bekannten Spieler, indem er über RMI deren *init*-Methode aufruft. Dabei werden sämtliche Informationen, die ein Spieler kennen muss, übermittelt (Referenz zum Server und zu den Mitspielern, die gemischten Karten).

5.4.3 Spielstart

Nach der Initialisierung werden alle Spieler über den Spielstart informiert. Dies geschieht über den Remoteaufruf der Methode *spielStart*.

5.4.4 Während des Spiels

Sobald ein Spieler über den Spielstart informiert wurde, versucht er der Reihe nach, die Karten aus seinen Spielslots oder jene aus seinem Handstapel auf einen eigenen oder den Stapel eines Nachbarn zu legen. Für einen Versuch auf dem eigenen Stapel ist kein Remoteaufruf nötig. Um die Karte bei einem Nachbarn zu platzieren, wird diesem die Karte mittels Aufruf der Methode *legeKarte* übergeben. Hat die Karte auf einem Stapel platz gefunden, so bestätigt dies der Empfänger der Karte mit *true*, ansonsten gibt er *false* zurück und die Karte geht wieder zurück in den Slot bzw. in den Handkartenstapel.

5.4.5 Ligretto Stop!

Ist der Ligrettostapel eines Spielers leer, so ruft er die *ligrettoStop*-Methode des Servers auf (Abbildung 6). Dieser übermittelt diese Botschaft allen Spielern, indem er deren *spielEnde*-Methode aufruft.

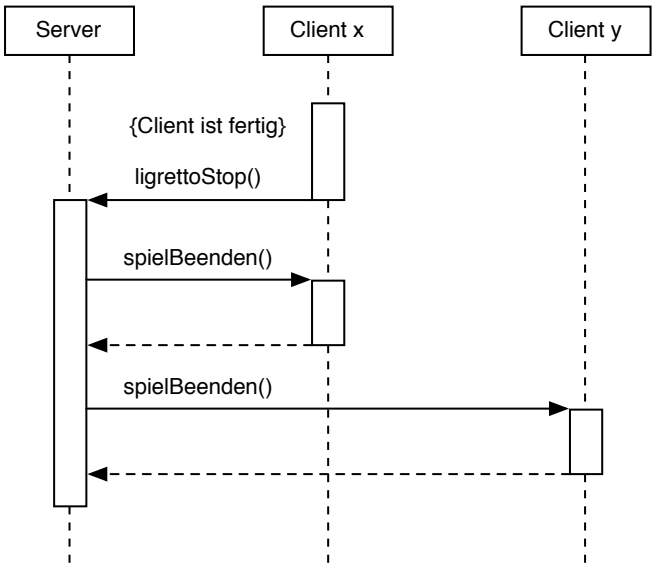

Abbildung 6: Sequenzdiagramm zum auslösen des Ligretto-Stop-Signals

Durch die Parallelität des Systems ist es möglich, dass während dem legen einer Karte das Spiel beendet wird (Abbildung 7, Abbildung 8). Im folgenden Beispiel will der Client y auf einem Stapel des Clients y eine Karte ablegen. Falls der Client y die die Nachricht zum Spielende bekommt während er noch auf eine Antwort des Clients x wartet, so muss diser warten, bis er eine Antwort bekommt.

Beim Client x wird entschieden, welche Nachricht zu erst eingetroffen ist.

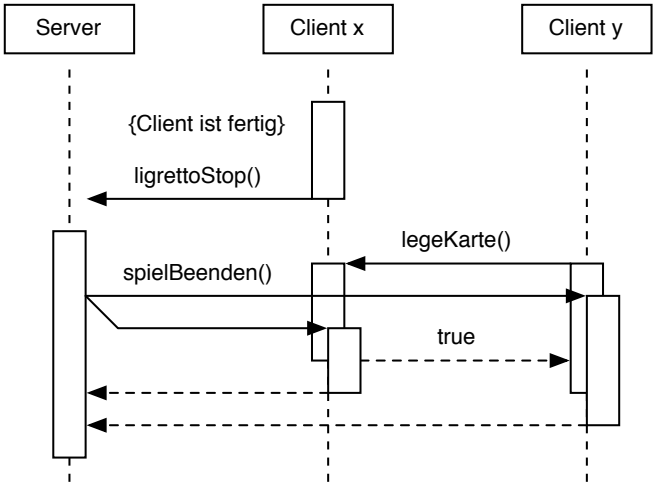

Abbildung 7: Race-Condition beim legen einer Karte, der legende Client gewinnt

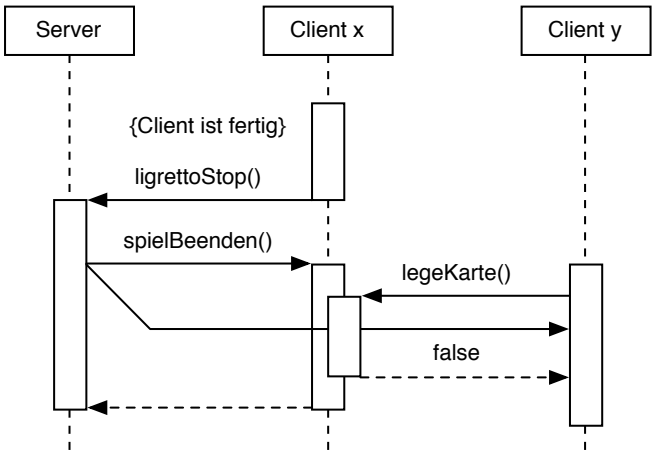

Abbildung 8: Race-Condition beim legen einer Karte, der legende Client verliert

5.4.6 Punkte zählen

Nach Spielende ruft der Server von allen Spielern den Punktestand ab. Dies geschieht über dem RMI-Aufruf der Methode *getPunkte*.

6 Datenstrukturen und Schnittstellen

Abbildung 9 bildet die Klassen und Interfaces und alle Daten und Funktionen ab , die zum Durchführen eines Ligrettospiels notwendig sind.

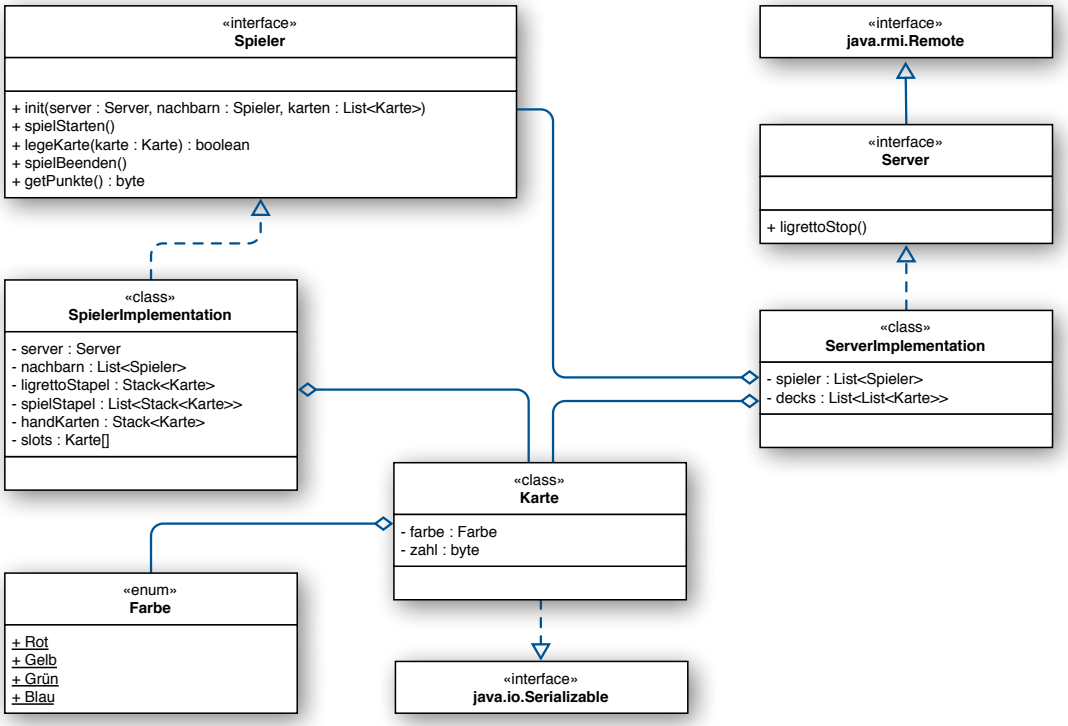

Abbildung 9: Klassendiagramm

6.1 Karte

Die Karte hat eine **Farbe** (Rot, Gelb, Grün oder Blau) und eine **Zahl** (zwischen eins und zehn). Sie ist ein serialisierbares Objekt, welches zwischen dem Server und verschiedenen Spielern ausgetauscht werden kann.

```
import java.io.Serializable;

public class Karte implements Serializable {

    // Instanzvariablen
    public Farbe farbe;
    public byte zahl;

    // Die vier Farben als Enumeration
    public enum Farbe {
        Rot, Gelb, Grün, Blau;
    }
}
```

listings/Karte.java

6.2 Server

Das **Serverinterface** definiert eine Methode *ligrettoStop()*, welche von dem Spieler, der als erster alle Karten seines Ligrettostapels verspielt hat, aufgerufen wird.

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Server extends Remote {

    // Ligretto Stop Event (von Spieler)
    public void ligrettoStop() throws RemoteException;
}
```

listings/Server.java

Die Klasse *ServerImplementation* besitzt zwei Listen; *spieler*, worin alle am Spiel teilnehmenden Spieler enthalten sind und *decks*, welche bereits gemischte Kartenstapel à 40 Karten für die Spieler enthält.

```
import java.rmi.RemoteException;
import java.util.List;

public class ServerImplementation implements Server {

    // Instanzvariablen
    private List<Spieler> spieler;
    private List<List<Karte>> decks;

    // Methodenimplementation
    public static void ligrettoStop() throws RemoteException { ...
       }
}
```

listings/ServerImplementation.java

6.3 Spieler

Ein **Spieler** kennt vier Methoden. Mit dem Aufruf von *init()* erhält er alle Informationen, die er zum Spiel kennen muss. Wo ist der Server? Wo sind meine Nachbarn? Was sind meine Karten?

Dabei werden die 40 Karten gemäss den Spielregeln auf Ligretto-, Spiel- und Handkartenstapel sowie auf die vier Slots aufgeteilt. Das Spiel wird mit *spielStarten()* begonnen. Ab diesem Aufruf spielt der Spieler, bis er (oder ein Gegner) das Spiel beendet. Die Methode *legeKarte()* kann vom Spieler selbst oder von einem seiner Nachbarn aufgerufen werden. Dabei wird eine Karte übergeben und (wenn sie auf einen der Spielstapel passt) niedergelegt. Im Erfolgsfall wird dabei *true* zurückgegeben, ansonsten *false*. Ein Aufruf von *spielBeenden()* beendet das Spiel. Zuletzt wird *getPunkte()* vom Server aufgerufen um den Sieger der aktuellen Spielrunde zu ermitteln.

```
import java.rmi.Remote;
import java.rmi.RemoteException;
import java.util.List;

public interface Spieler extends Remote {

    // Spieler mit Karten auszurüsten (vom Server)
    public void init(Server server, List<Spieler> nachbarn,
        List<Karte> karten) throws RemoteException;

    // Ligretto Start Event (vom Server)
    public void spielStarten() throws RemoteException;

    // Karte legen (von anderen Spielern oder selber)
    public boolean legeKarte(Karte karte);

    // Ligretto Stop Event (vom Server)
    public void spielBeenden() throws Remote Exception;

    // Punktestand ermitteln (vom Server)
    public byte getPunkte();
}
```

listings/Spieler.java

SpielerImplementation implementiert das oben beschriebene Spielerinterface. Dabei werden folgende Attribute definiert:

1. *server*: die Serverinstanz, die das Spiel kontrolliert
2. *nachbarn*: eine Liste aller Nachbarn
3. *ligrettoStapel*: ein Stapel mit anfänglich zehn Karten, die möglichst rasch verspielt werden müssen
4. *spielStapel*: Die (maximal vier) Stapel, auf die alle Spieler ihre Karten legen können
5. *handKarten*: der Stapel mit den Karten, die der Spieler in der Hand hält
6. *slots*: die vier offen liegenden Karten, die neben dem Ligrettostapel platziert werden

```
import java.rmi.RemoteException;
import java.util.List;
import java.util.Stack;

public class SpielerImplementation implements Spieler {

    // Instanzvariablen
    private Server server;
    private List<Spieler> nachbarn;
    private Stack<Karte> ligrettoStapel;
    private List<Stack<Karte>> spielStapel;
    private Stack<Karte> handKarten;
    private Karte[] slots;

    // Methodenimplementation
    public static void init(Server server, List<Spieler> nachbarn,
       List<Karte> karten) throws RemoteException { ... }
    public static void spielStarten() throws RemoteException { ...
       }
    public static boolean legeKarte(Karte karte) throws
       RemoteException { ... }
    public static void spielBeenden() throws Remote Exception {
       ... }
    public static byte getPunkte() throws RemoteException { ... }
}
```

listings/SpielerImplementation.java

7 Erwartetes Ergebnis

Unser geplantes Spiellayout sieht vor, dass jeder Client eigentlich nur bis zu 6 seiner eigenen Nachbarn kennt. Der Rest des Spiels ist ihm nicht direkt bekannt. Dadurch wollen wir erreichen, dass der Workload lokal in einer kleinen Gruppe bleibt und sich nicht auf das komplette System exponentiell auswirkt.

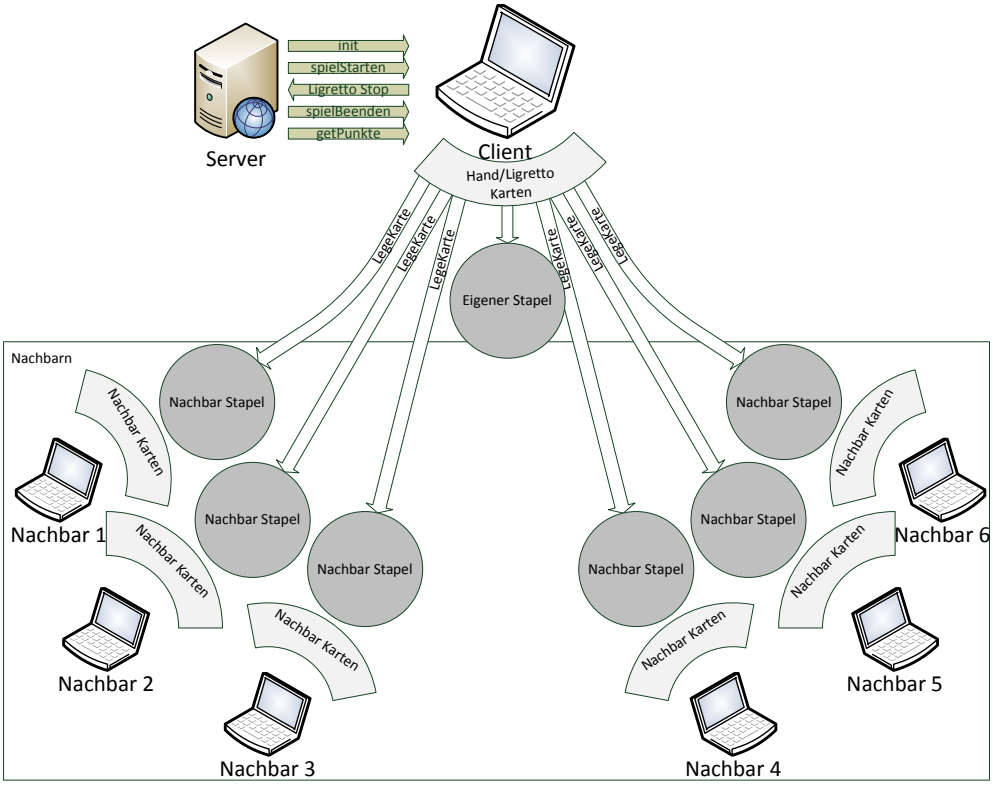

Abbildung 10: Spiel aus Client-Sicht

Abbildung 10 zeigt, wie sich der Client verhält und was er wahrnimmt. Er bekommt vom Server alle nötigen Inputs und versucht, auf den eigenen und den Stapeln seiner Nachbarn seine Karten zu legen. Gelingt ihm dies nicht, geht er weiter zur nächsten Karte, bis es ihm gelingt.

Wir hatten uns überlegt, zuerst zu prüfen, ob auf einem Nachbar Stapel eine Karte gelegt werden könnte und es dann zu tun, sind aber von diesem Ansatz wieder abgekommen, weil dem aufrufenden Client in der Zwischenzeit sehr wahrscheinlich bereits andere Clients zuvorkommen würden und damit viele Versuche (eine Karte zu legen) sowieso misslingen würden. Ob der Client ständig versucht, eine Karte zu legen oder ob eine Karte legbar ist und danach versucht, die Karte zu legen, kommt fast auf das Selbe hinaus. Wir haben uns aber dafür entschieden, nur die Karten zu legen, da damit die

RMI Calls minimiert werden.

Generell hat unsere verteilte Lösung ein Problem, sobald ein beliebiger Client ausfällt. Sollte ein Nachbar ausfallen, so wartet der Client unter Umständen sehr lange auf einen Timeout, bis er weiter machen kann. Vermutlich sind in der Zeit andere Clients bereits mit ihrem Spiel fertig und alle Nachbarn des ausgefallenen Clients währen mit dem Warten auf das Timeout "beschäftigt".

Um dieses Risiko zu vermindern, muss garantiert werden, dass kein Client ausfällt. Ansonsten muss das Spiel nochmals wiederholt werden.

7.1 Auswirkungen bei starker Skalierung

Aufgrund unserer geplanten Limitierung des Clients auf seine 6 Nachbarn wird der Workload innerhalb des Spiels linear verteilt.

Jeder Client hat (egal wie gross das Spiel ist) immer nur maximal 6 Nachbarn, die versuchen, bei ihm Karten auf die Stapel zu legen. Damit erhöht sich der Workload für die Clients nicht, egal wie viele Clients am Spiel teilnehmen.

Der Knackpunkt ist jedoch die Verteilung des Start- und Endsignals.

7.1.1 Start- und Endsignal ohne Timestamp

Der grösste Knackpunkt ist die zeitliche Verteilung des *SpielStart* und *SpielBeenden* Signals **ohne** Timestamps.

In einem 100 Mbit/s LAN ist die durchschnittliche Latenz bei ca. 60 µs.

Dadurch ergibt sich pro Client ein Zeitzuwachs für das Verteilen des *SpielStart* und *SpielBeenden* Signals.

Abbildung 11 zeigt, wie sich die Zeitdauer mit erhöhter Client Anzahl verlängert.

Problem:

Jeder Client hat immer 4 Karten die er legen muss plus eine Karte aus seinen Handkarten, die er optional auch noch legen könnte. Er versucht, jede Karte bei 6 Nachbarn zu legen. Wir gehen von einer Fehlerrate von ca. 98% aus. Das heisst, nur eine aus 50 Karten kann überhaupt gelegt werden. Tendenziell geht diese Fehlerquote sogar eher noch höher.

Im Best Case legt der Gewinner des Spiels genau 10 Karten, im Worst Case legt er 36 Karten. Im Durchschnitt legt der Sieger eines Spiels (reales Spiel mit Karten) ca. 20 Karten.

Best Case: Um 10 Karten zu legen (bei einer 98% Fehlerrate) muss der Client 500 Versuche starten, um eine Karte zu legen. $500 * 60\ \mu s = 30000\ \mu s = 30\ ms$.

Average Case: Um 20 Karten zu legen (bei einer 98% Fehlerrate) muss der Client 1000

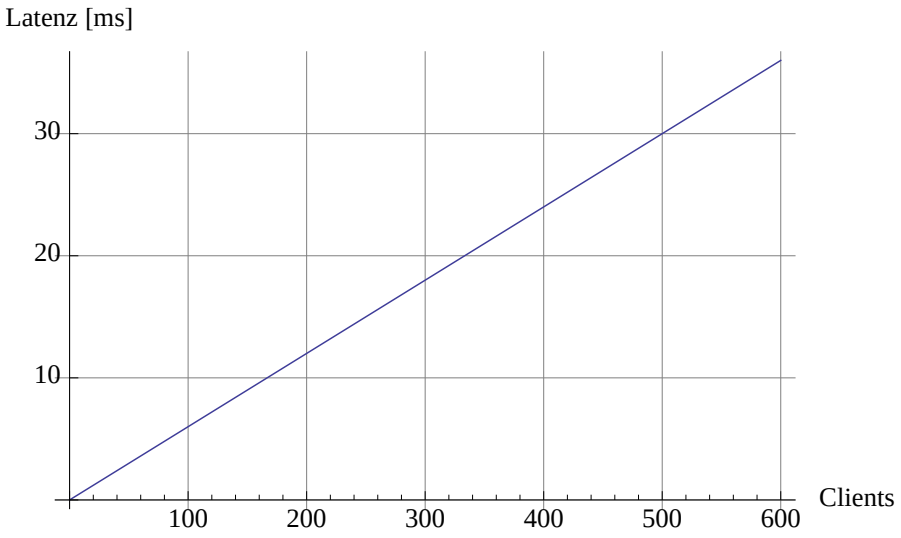

Abbildung 11: Zeit für das Verteilen des Start- oder Endsignals

Versuche starten, um eine Karte zu legen. $1000 * 60\ \mu s = 60000\ \mu s = 60\ ms$.

Worst Case: Um 34 Karten zu legen (bei einer 98% Fehlerrate) muss der Client 1700 Versuche starten, um eine Karte zu legen. $1700 * 60\ \mu s = 102000\ \mu s = 102\ ms$.

Fazit:

Bereits ab einer Spielerzahl von ca. 100 Clients hat der erste Spieler im Best-Case bereits gewonnen, bevor der letzte das Startsignal erhält.

Ab einer Spielerzahl von 200 hat bereits im Average-Case der erste Spieler gewonnen, ab einer Spielerzahl von 350 tritt sogar der Worst-Case ein.

Man kann also sagen, dass das Spielresultat voraussichtlich nur bis etwa 10 Spieler einigermassen zufällig ist. Von 10 bis 50 ist der Vorteil bereits klar bei den ersten 10 Spielern; alle Spieler von 50-100 haben keine reale Chance mehr zu gewinnen, da der Gewinner sicher unter den ersten 50 sein wird.

7.1.2 Start- und Endsignal als Timestamp

Als alternative zu der oben beschriebenen Problematik währe folgender Lösungsvorschlag machbar: Alle Signale werden in Form eines Timestamps übertragen. Dies setzt jedoch hohe anforderungen an die Zeitsynchronisation und an die KI.

Werden Timestamps benutzt, um die Start- und Endsignale zu übertragen, stellt dies zwar grössere Anforderungen an die Zeitsynchronisation und die KI, aber die Möglichkeit, dass ein Client gewinnt, ist dabei absolut zufällig.

Der erste Knackpunkt ist die Verteilung der Karten sowie das Startsignal. Der Server

benötigt eine gewisse Zeit, um die Karten zu verteilen. Anschliessend benötigt er eine gewisse Zeit, allen Teilnehmern das Startsignal zu übermitteln. Um sicherzustellen, dass alle zur gleichen Zeit starten, könnte man das Startsignal in Form eines Startzeitpunkts ein paar Sekunden in der Zukunft zu definieren. Damit kann sichergestellt werden, dass alle Clients exakt zum gleichen Zeitpunkt mit dem Spiel starten können.

Der zweite Knackpunkt in unserem System mit sehr vielen Clients ist der Zeitpunkt, zu dem ein Client *Ligretto-Stop!* aufruft. In diesem Augenblick muss der Server bei allen Clients das Spiel beenden. Dies kann bei sehr vielen Clients eine gewisse Zeit dauern, bis alle Clients das Stopsignal bekommen haben. Man könnte mit einem Timestamp sicherstellen, dass das Stopsignal auch rückwirkend gültig ist. Es stellt aber erweiterte Anforderungen an die KI, ihre Schritte rückgängig zu machen, sollte das Stopsignal erst einige Zeit nach dem eigentlichen Stopzeitpunkt ankommen.

Der dritte (jedoch unkritische) Knackpunkt ist das Abholen der Punkte. Der Server hat dafür jedoch keinen Zeitdruck, da alle Clients bereits gestoppt sind. Es kann aber eine gewisse Zeit dauern, bis alle Clients ihre Punkte abgegeben haben, da der Server jeden einzelnen anfragen muss.