

# Simulation eines Ligretto-Spiels in einem verteilten System

Peter Herter <hertepet@zhwin.ch>  
Pascal Iacoviello <iacovpas@zhwin.ch>  
Tobias Klauser <klaustob@zhwin.ch>

18. Januar 2007

## **Zusammenfassung**

Das Kartenspiel Ligretto soll mittels eines verteilten Systems simuliert werden. Um den Spielverlauf möglichst realistisch nachzubilden, werden gezielt die in der Netzwerkkommunikation auftretenden Verzögerungen eingesetzt. Die einzelnen Komponenten des Spiels werden in unabhängige Software-Komponenten unterteilt. Diese werden anschliessend zur Simulation des Spiels zufällig (unter Einhaltung gewisser Randbedingungen) auf den Rechnern im verteilten System angeordnet. Die damit erzeugten Resultate sind damit zufällig und fallen auch bei identischen Eingangsparametern unterschiedlich aus.

# Inhaltsverzeichnis

<b>1 Aufgabenstellung</b>	<b>3</b>
<b>2 Algorithmus</b>	<b>3</b>
2.1 Grobbeschreibung	3
2.2 Ausgangslage	4
2.3 Initialisierung des Systems	4
2.4 Auftrag ins System speisen	5
2.4.1 Verteilung der Teilaufgaben	5
2.4.2 Verteilung der Spieldaten	7
2.5 Verarbeitung des Auftrags	8
2.6 Ende der Simulation	8
2.7 Zusammenführen des Schlussresultates	8
2.8 Resultat aus dem System entnehmen	8
<b>3 Datenbeschreibung</b>	<b>9</b>
<b>4 Schnittstellenbeschreibung</b>	<b>10</b>
4.1 Schnittstellen für Spielverlauf	10
4.2 Schnittstellen für Initialisierung	13
<b>5 Analyse</b>	<b>15</b>
5.1 CPU-Auslastung	15
5.2 Speicherbedarf	15
5.3 Netzwerk-Traffic	15
5.3.1 Berechnung	16
<b>6 Mengengerüst</b>	<b>17</b>
6.1 Minimalanforderungen	17
6.2 Maximale Anzahl Rechenclients	17
6.3 Skalierbarkeit	17
<b>7 Anhang</b>	<b>18</b>

# 1 Aufgabenstellung

Der Verlauf einer vollständigen Partie Ligretto<sup>1</sup> soll in einem über Netzwerk gekoppelten, verteilten System simuliert werden. Um echten Zufall ins Spiel zu bringen, soll das Spiel über mehrere Maschinen verteilt werden. Um die Verarbeitung der Eingangsparameter, sowie das Mischeln und Verteilen der Karten muss sich die vorgeschlagene Lösung nicht kümmern. Die Daten werden bereits aufbereitet vom Eingangsserver (Webserver) zur Verfügung gestellt und können unverändert übernommen werden.

## 2 Algorithmus

### 2.1 Grobbeschreibung

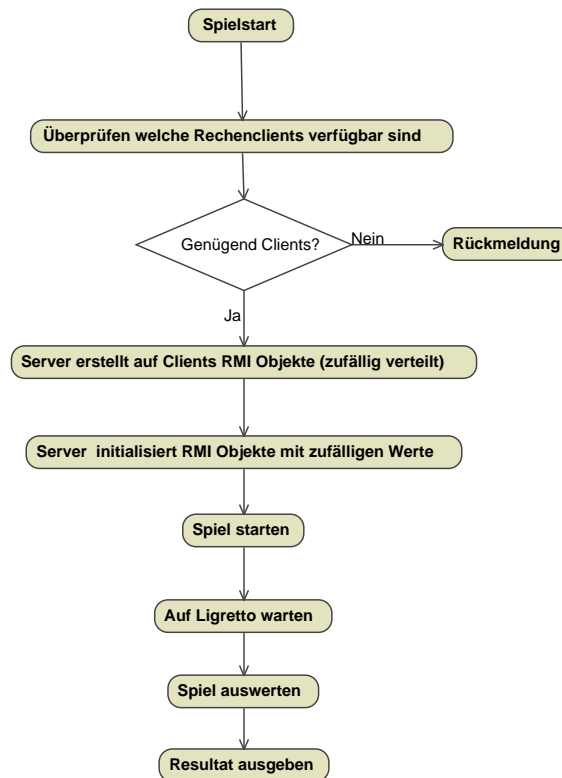


Abbildung 1: Ablauf der Simulation

Im vorliegenden Algorithmus wird der Ansatz verfolgt, die Komponenten bzw. deren Kommunikation untereinander so realitätsnahe wie möglich zu gestalten. Das heisst, Verzögerungen, Zufall und Wartezeiten aus dem realen Spiel werden auf die im verteilten System nicht deterministischen Laufzeiten der Nachrichten abgebildet, um das Element des Zufalls in die Simulation zu bringen. Die Simulation des eigentlichen Spiels wird auf die folgenden Komponenten verteilt:

- Spieler
- Ligretto-Karten

<sup>1</sup><http://de.wikipedia.org/wiki/Ligretto>

- Handkarten
- Teiltische

Pro Spielteilnehmer existiert je eine dieser Komponenten. Die Komponenten verwalten ihre Daten unabhängig voneinander und kommunizieren nur über definierte Schnittstellen. Diese Schnittstellen entsprechen soweit wie möglich den im realen Spiel möglichen Aktionen. Da jede Komponente grundsätzlich auf einem beliebigen Rechner liegen kann, wird jede dieser Nachrichten über das Netzwerk übertragen und hat damit eine nicht voraussagbare Laufzeit. Es ist also zum Beispiel möglich, dass sich die Handkarten eines Spielers auf einem anderen Rechner befinden als der Spieler selbst und die Ligretto-Karten wiederum auf einem anderen Rechner.

Zur Initialisierung und Auswertung der obigen Komponenten wird ausserdem ein Spielleiter benötigt, der Einfachheit halber läuft dieser auf dem Webserver, über den die Anfrage eingeht. Sie hat aber für die eigentliche Simulation keine Bedeutung.

Jeder Spieler hat die selbe Künstliche Intelligenz (KI) eingebaut. Der Spielverlauf (Fertigkeit, Reaktionsgeschwindigkeit etc. eines Spielers) wird nur durch Laufzeiten der Kommunikation im Netzwerk bestimmt. Dadurch ist der Spielverlauf auch bei mehrmaligem Wiederholen bei identischer Ausgangslage nicht deterministisch.

Das Ende der Simulation erfolgt, wenn einer der Spieler Ligretto erreicht, das heisst, wenn er alle seine Ligretto-Karten abgelegt hat. Dies teilt er dem Spielleiter und der anderen Spielern mit. Sobald alle Spieler ihre Karten ausgezählt und dem Spielleiter übermittelt haben, erstellt dieser die Rangliste mit den erreichten Punkten und liefert diese an den Benutzer aus.

Die Abbildung 2.1 zeigt den groben Ablauf der Simulation auf.

## 2.2 Ausgangslage

Die Anfrage, ein Ligretto-Spiel zu simulieren ist über den Webserver bereits eingegangen. Dieser Server übernimmt während des gesamten Ablaufs der Simulation die Rolle des Koordinators (im Folgenden Spielleiter) genannt. Die Spielparameter sind durch die Eingabe des Auftrags durch den Benutzer bereits bekannt. Anhand dieser Informationen wird das Spiel im Folgenden simuliert.

## 2.3 Initialisierung des Systems

Dem zu simulierenden Spiel wird zuerst eine eindeutige Nummer zugewiesen (z.B. eine fortlaufende Nummer), unter der das Spiel danach in der Auswertung (vgl. 2.8) für den Benutzer zugänglich ist. Diese Nummer sollte dem Benutzer mitgeteilt werden.

Die Adresse des Spielleiters ist allen Clients beim Start der Software bekannt (siehe Listing 1), wird ihnen also z.B. via Kommandozeilenparameter oder Konfigurationsdatei übergeben. Des weiteren implementiert die Client-Software mit Ausnahme von `GameLeader` alle Schnittstellen-Interfaces (vgl. Abschnitt 4) in konkreten Implementierungen. Die Spielleiter-Klasse muss das Interface `GameLeader` implementieren. Es wird ebenfalls davon ausgegangen, dass `rmiregistry` auf dem Server und allen Clients korrekt konfiguriert ist (Security Policy etc.) und vor dem Aufruf der eigentlichen Software bereits als Daemon läuft.

Listing 1: Starten der Client-Software

```
$ rmiregistry &
$ java LigrettoClient 192.168.1.10 -Djava.security.policy=this.policy
Starte Ligretto Client...
Bereit... (Beenden mit Ctrl-C)
```

Wird nun die Client-Software gestartet, holt sie sich vom Spielleiter via RMI ein Objekt, welches das Interface `WebServerCommunication` implementiert. Anschliessend erstellt sie ein Objekt der Klasse, welche das `Communication` Interface implementiert und übergibt dieses mittels der Methode `signOn()` dem Spielleiter.

## Listing 2: Anmelden beim Server

```
...
WebServerCommunication srv = (WebServerCommunication) Naming.lookup("
    rmi://" + host + "/" + service);
Communication com = new CommunicationImpl(params);
srv.signon(com);
...
```

Nach dem Eintreffen der `signon()` Nachricht, behält der Server das übergebene `Communication`-Objekt in einer internen Datenstruktur (z.B. einer Linked List). Damit sind dem Spielleiter alle verfügbaren Clients bekannt und er kann mit dem Verteilen der Aufgaben und damit der eigentlichen Verarbeitung des Auftrags beginnen.

## 2.4 Auftrag ins System speisen

Auf Grund der Eingangsparameter erzeugt der Spielleiter nun die Objekte, in welchen die eigentliche Spiellogik und die Verwaltungsinformationen abgelegt sind. Für die Ligretto-Spieler wird aus jeder der folgenden Interfaces eine Klasse implementiert (diese befinden sich auf den Clients).

- `Player` (Listing 8)
- `LigrettoCards` (Listing 9)
- `HandCards` (Listing 10)
- `TablePart` (Listing 11)

Diese Interfaces entsprechen logischerweise auch den Komponenten der Software.

### 2.4.1 Verteilung der Teilaufgaben

**Spieler** Dies ist die zentrale Komponente für jeden simulierten Ligretto-Spieler. Sie hält einerseits Referenzen auf die eigenen Ligretto- und Handkarten (siehe unten), andererseits enthält sie die KI Funktionalität, welche das Verhalten des Spielers steuert. Insbesondere müssen *allen* Spielern auch *alle* Teiltische bekannt sein, damit diese abgefragt und bei Bedarf Karten darauf abgelegt werden können. Der Spieler muss während Simulation regelmässig alle Tischeile bzw. die Kartenstapel darauf abfragen, um zu überprüfen, ob er eine seiner Karten ablegen kann.

**Ligretto-Karten** Die Verwaltung der Ligretto-Karten erfolgt in dieser Komponente. Die vier obenauf liegenden Karten können mit der Methode `getTopCards()` abgefragt werden und davon eine mit der Methode `popCard()` geholt werden. Die Komponente sorgt selbstständig dafür, dass nach dem Holen einer Karte die nächste Karte aufgedeckt wird.

**Handkarten** Alle Karten, welche sich nicht im Ligretto-Stapel befinden, landen in den Handkarten. Diese Komponente sorgt für das Umdrehen der Handkarten (Methode `rotate()`) und liefert über die Methode `getTopCard()` jeweils die oberste Karte zurück.

**(Teil-)Tische mit Kartenstapeln** : Statt den Spieltisch mit den Kartenstapeln auf einem einzigen Rechner abzulegen, welcher dadurch ausserordentlich oft angefragt würde, wird der Tisch in Teile aufgeteilt. Es liegt nahe, genau so viele Tischeile anzulegen, wie Spieler beteiligt sind. Die Idee dahinter ist, dass in einem realen Ligretto-Spiel gewisse Stapel (oder eben Tischeile) näher bei einem Spieler liegen als andere und der Spieler auf diese Stapel im Allgemeinen schneller ablegen kann. Dieser Sachverhalt wird durch diese Aufteilung der Tischeile nachgebildet. Die Komponente stellt Methoden zur Verfügung, um die IDs aller

Stapel zu bekommen (`getStackIds()`), die oberste Karte eines jeweiligen Stapels zu bekommen (`getTopCardOfStack()`) und Karten auf einen Stapel abzulegen. Ausserdem steht eine Methode zur Verfügung, welche die Stapel bei Erreichen von Ligretto für weiteren Zugriff sperren (`lock()`) und eine Methode `examine()`, die zur Auswertung der Kartenstapel bei Spielende (siehe Abschnitt 2.8) dient.

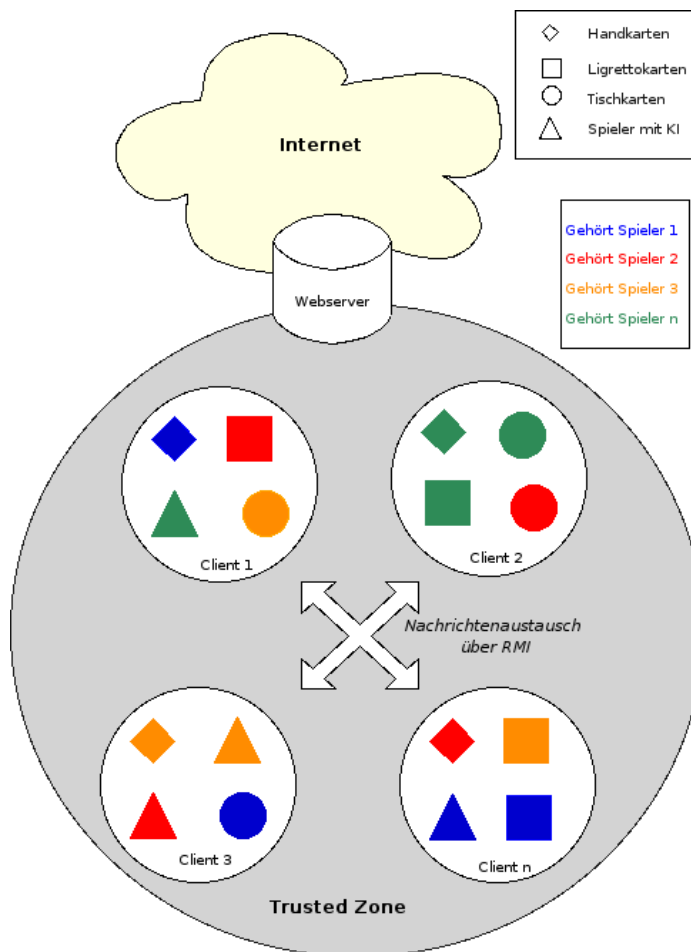


Abbildung 2: Aufteilung der Software-Komponenten auf einzelne Clients

Aus den konkreten Klassen erstellt der Spielleiter nun via RMI-Aufruf auf dem bei der Anmeldung übergebenen `Communication`-Objekt pro Ligretto-Spieler eine Instanz. Diese kann sich grundsätzlich auf einem *beliebigen* Client befinden. Es ist also z.B. möglich, dass ein Spieler (Interface `Player`) seine Handkarten (Interface `HandCards`) auf einem anderen Rechner vorfindet, als er selbst läuft. Der in Listing 3 abgebildete Java-Code zeigt diesen Ablauf im Wesentlichen auf.

Listing 3: Erstellen der Objekte auf dem Server

```

...
/*
 * Liste der Communication-Objekte
 * Pro Client-Rechner ein Objekt
 */
LinkedList<Communication> clients[];
/*
 * Zufallszahlengenerator
 */
Random rand;
...
int numClients = clients.size();
for (int i = 0; i < numPlayers; i++) {
    // Liefert eine Zufallszahl zwischen 0 und numClients
    int rand = rand.nextInt(numClients);
    Player play = clients.get(rand).createPlayer();
    rand = rand.nextInt(numClients);
    LigrettoCards lig = clients.get(rand).createLigrettoCards();
    rand = rand.nextInt(numClients);
    HandCards hand = clients.get(rand).createHandCards();
    rand = rand.nextInt(numClients);
    TablePart part = clients.get(rand).createTablePart();
    ...
    // Objekte initialisieren
}
...

```

Es ist also durchaus möglich, dass auf dem einen Client 10 Objekte erstellt werden, auf einem anderen aber nur 2 Objekte. In einer konkreten Implementierung der Software müssten noch zusätzliche Prüfungen durchgeführt werden, um zu vermeiden, dass z.B. alle Objekte auf dem selben Client erzeugt werden oder einige Clients nicht beschäftigt werden.

Sollten wesentlich mehr Rechner vorhanden sein, als Aufgaben zu verteilen sind, gehen einzelne Rechner leer aus, sie werden nicht in die Simulation mit einbezogen. Dies bedeutet zwar eine Verschwendung von Ressourcen, tritt aber nur in Fällen auf, wo ein Spiel mit sehr wenigen Spielern und Karten simuliert wird. Eine feinere Aufteilung der Software-Komponenten würde gleichzeitig auch eine Erhöhung der Komplexität in der Kommunikation bedeuten (und damit zu einer Erhöhung der potentiellen Fehlerquellen führen). Dies kann sich ab einem gewissen Punkt wieder negativ auf die realitätsnahe Simulation auswirken. Da ausserdem die Berechnungszeit und die CPU-Auslastung nach unseren Anforderungen nicht optimiert werden müssen, ist dieser Aspekt vernachlässigbar.

#### 2.4.2 Verteilung der Spieldaten

Nach der Erstellung der für die Simulation nötigen Objekte, muss dafür gesorgt werden, dass die nötigen Objekte sich gegenseitig kennen. Vor Beginn einer Spielsimulation werden deshalb auf dem Server durch einen Zufallsgenerator die Reihenfolge der Stapel der Ligretto- und Handkarten bestimmt. In dieser Initialisierungsphase teilt der Server den Spieler-Objekten ebenfalls alle benötigten Referenzen auf die anderen Spieler und Tischteile mit (Methode `init()` im `Player` Interface). Somit weiss jede Spieler-Komponente, wo sie die Kartenstapel auf dem "Tisch" finden kann und welche anderen "Spieler" er bei Erreichen von Ligretto informieren muss.

## 2.5 Verarbeitung des Auftrags

Die Methode `startGame()` – welche im Interface `Player` definiert ist – wird vom Spielleiter bei jedem Spieler aufgerufen. Die Spieler beginnen daraufhin mit dem Umdrehen der Handkarten und dem Ablegen der möglichen Karten auf den Kartestapeln. Dazu fragen sie alle Tischeile in regelmässigen Abständen ab, ob es möglich ist, Karten abzulegen. Auch die eigenen Ligretto- und Handkarten müssen regelmässig abgefragt werden. Diese gesamte Funktionalität ist in der Künstlichen Intelligenz programmiert.

Wollen zwei Spieler ihre Karte quasi-gleichzeitig auf den selben Tischstapel ablegen, wird nur derjenige, welcher zuerst die Referenz auf das Objekt erhält, dieses sperren (der Zugriff auf das Objekt ist synchronisiert), der zweite Spieler kann seine Karte nicht ablegen.

## 2.6 Ende der Simulation

Hat ein Spieler Ligretto, teilt er allen anderen Spielern dies mit (Methode `ligretto()`) und benachrichtigt gleichzeitig auch den `GameLeader` (Webserver). Hat ein Spieler die Ligretto-Nachricht erhalten, sperrt er sofort seinen zugehörigen Tischeile, so dass dort keine Karten mehr abgelegt werden können. Durch die unterschiedliche Laufzeit der Nachricht zu den einzelnen Spielern ist es jedoch möglich, dass ein oder sogar mehrere Spieler auch nach der Ligretto-Nachricht ihre Karten noch ablegen. Dieser Effekt ist aber gewollt, da er zur realitätsnahen Simulation beiträgt. Falls dies nicht erwünscht wäre, könnte das Problem durch die Einführung von absoluten Zeitstempeln in den Nachrichten vermieden werden.

## 2.7 Zusammenführen des Schlussresultates

Nach Eintreffen der Nachricht `ligretto()` bei einem Spieler, zählt dieser seine verbleibenden Ligretto-Karten (Minuspunkte) und teilt diese dem Server (`GameLeader`) mit (Methode `deliverLigrettoScore()` im Interface `GameLeader`). Sobald der `GameLeader` von allen Spielern die Ligretto-Punkte erhalten hat, kann er mit dem Auszählen der Tischkarten beginnen.

## 2.8 Resultat aus dem System entnehmen

Der Spielleiter hat von alle Spielern die Punkte der Ligretto-Karten erhalten. Danach beginnt er mit dem auszählen der auf den Tischstapeln abgelegten Karten (Pluspunkte) mit Hilfe der Methode `examine()` aus dem `TablePart` Interface. Aus den Resultaten erstellt der Spielleiter, eine statische XHTML-Seite mit der Rangliste des Spiels. Denkbar wäre auch, dass gewisse statistische Informationen wie Spieldauer, Anzahl benutzter Tischstapel etc. ebenfalls aufgeführt werden. Da sich der Spielleiter auf dem Webserver befindet, kann er die generierte Datei direkt ins Dateisystem ablegen, so dass diese für den HTTP-Daemon zugänglich ist. Zum verbessern der Auffindbarkeit der Resultate für den Benutzer wird die Seite auch in einer Übersichtsseite vermerkt.

### 3 Datenbeschreibung

Listing 4: CardColor.java

```
package ligretto;

public enum CardColor {
    RED, GREEN, BLUE, YELLOW
}
```

Listing 5: Card.java

```
package ligretto;

/*
 * Memory-Bedarf: 3 Integer + Java Verwaltungsinformationen
 */

public class Card {

    public Card(int number, CardColor color, int owner) {
        this.number = number;
        this.color = color;
        this.owner = owner;
    }

    public int getNumber() { return number; };
    public CardColor getColor() { return color; };
    public int getOwner() { return owner; };

    private int number;
    private CardColor color;
    private int owner;
}
```

Listing 6: ResultPart.java

```
package ligretto;

/*
 * Memory-Bedarf: 2 Integer + Java Verwaltungsinformationen
 */

public class ResultPart {

    public ResultPart(int playerId, int score) {
        this.playerId = playerId;
        this.score = score;
    }

    public int getScore() { return score; };
    public int getPlayerId() { return playerId; };

    private int playerId;
    private int score;
}
```

## 4 Schnittstellenbeschreibung

### 4.1 Schnittstellen für Spielverlauf

Die folgenden Schnittstellen beziehen sich auf die während des Spielverlaufs relevanten Objekte.

Listing 7: GameLeader.java

```
package ligretto;

/*
 * Memory-Bedarf: je nach Implementation,
 * mindestens aber Referenzen auf alle Spieler
 */

public interface GameLeader extends java.rmi.Remote {
    /**
     * Teilt dem Spielleiter die Ligretto-Punktzahl des jeweiligen
     * Spielers mit.
     * Muss von jedem Spieler bei Spielende einmal aufgerufen werden,
     * damit das Spiel abgeschlossen werden kann.
     */
    void deliverLigrettoScore(ResultPart result) throws java.rmi.
        RemoteException;
}
```

Listing 8: Player.java

```
package ligretto;

/*
 * Memory-Bedarf: je nach Implementation,
 * mindestens Referenzen auf alle anderen Spieler und Tischteile
 * sowie auf die eigenen Ligretto- und Handkarten
 */

public interface Player extends java.rmi.Remote {

    /**
     * Überprüft, ob der Spieler Ligretto hat
     */
    boolean isLigretto() throws java.rmi.RemoteException;

    /**
     * Teilt dem Spieler mit, dass irgendein Spieler Ligretto hat.
     * (=> Spieler muss stoppen)
     */
    void ligretto() throws java.rmi.RemoteException;

    /**
     * Beendet das Spiel und wertet es aus.
     * Die Auswertung beinhaltet nur den das Auszählen seiner
     * Ligretto-Garten. Der Rest wird vom Server übernommen.
     * @return Array von ResultParts (bestehend aus Spieler
     *         und dessen Score)
     */
    ResultPart finishGame() throws java.rmi.RemoteException;

    /**
     * Initialisiert den Spieler mit seiner ID und weist ihm seine
     * Handkarten und Ligretto-Karten zu.
     */
}
```

```

    * Gibt dem Spieler Referenzen auf den Spielleiter, auf die
    * verschiedenen Tischteile und auf die anderen Spieler.
    * @param tableParts je tiefer der Index, desto schneller sollte
    *                   der Zugriff auf den Tisch sein.
    */
    void init(GameLeader gameLeader,
              int playerId,
              HandCards handcards,
              LigrettoCards ligrettoCards,
              TablePart tableParts[],
              Player otherPlayers[]) throws java.rmi.
                RemoteException;

    /**
     * Startpfiff
     */
    void startGame() throws java.rmi.RemoteException;
}

```

Listing 9: LigrettoCards.java

```

package ligretto;

/*
 * Memory-Bedarf: je nach Implementation,
 * mindestens aber 13 Karten (39 Integer)
 */

public interface LigrettoCards extends java.rmi.Remote {

    /**
     * Gibt die 4 sichtbaren Karten zurück.
     */
    Card[] getTopCards() throws java.rmi.RemoteException;

    /**
     * Nimmt die Karte vom entsprechenden Stapel
     */
    void popCard(int stackNr) throws java.rmi.RemoteException;

    /**
     * Prüft ob der Zehnerstapel leer ist. (Ligretto?)
     */
    boolean isLigretto() throws java.rmi.RemoteException;

    /**
     * Initialisiert die Karten
     * @param cards 13 Karten
     */
    void init(Card cards[]) throws java.rmi.RemoteException;
}

```

Listing 10: HandCards.java

```

package ligretto;

/*
 * Memory-Bedarf: je nach Implementation,
 * mindestens aber 27 Karten (81 Integer)
 */

public interface HandCards extends java.rmi.Remote {
    /**

```

```

    * Gibt die oberste Karte des Stapels
    */
    Card getTopCard() throws java.rmi.RemoteException;

    /**
     * rotiert den Stapel um 3 Karten.
     * Falls der Stapel durch ist, wird der Stapel automatisch
     * gedreht.
     */
    void rotate() throws java.rmi.RemoteException;

    /**
     * Initialisiert den Kartenstapel
     * @param cards Array mit 27 Karten.
     */
    void init(Card cards[]) throws java.rmi.RemoteException;
}

```

Listing 11: TablePart.java

```

package ligretto;

/*
 * Memory-Bedarf: je nach Implementation
 */

public interface TablePart extends java.rmi.Remote {

    /**
     * Gibt die ID aller Stacks zurück
     */
    int[] getStackIds() throws java.rmi.RemoteException;

    /**
     * Gibt die oberste Karte des angegebenen Stapels zurück
     */
    Card getTopCardOfStack(int stackId) throws java.rmi.
        RemoteException;

    /**
     * Versucht die Karte auf den angegebene Stapel zu legen.
     * @return -1: Stapel gesperrt / 0: Nicht möglich / 1: Abgelegt
     * @param stackId StackId oder -1 für einen neuen Stapel
     */
    int pushCard(Card card, int stackId) throws java.rmi.
        RemoteException;

    /**
     * Sperrt den Tischteil für weitere Aktionen
     */
    void lock() throws java.rmi.RemoteException;

    /**
     * Wertet den Tischteil aus
     */
    ResultPart[] examine() throws java.rmi.RemoteException;
}

```

## 4.2 Schnittstellen für Initialisierung

Die folgenden Schnittstellen werden bei der Initialisierung des Systems (vgl. Abschnitt 2.3 auf Seite 4) verwendet. Das Interface `Communication` wird von jedem Rechner im verteilten System einmal implementiert und instanziiert und identifiziert ihn danach beim Spielleiter. Das Interface `WebServerCommunication` dient zur Anmeldung dieser `Communication` Objekte.

Listing 12: `Communication.java`

```
package ligretto;

/*
 * Memory-Bedarf: je nach Implementation
 */

public interface Communication extends java.rmi.Remote {

    /**
     * Erstellt auf diesem Rechner ein HandCards-Objekt
     */
    HandCards createHandCards() throws java.rmi.RemoteException;

    /**
     * Erstellt auf diesem Rechner ein LigrettoCards-Objekt
     */
    LigrettoCards createLigrettoCards() throws java.rmi.
        RemoteException;

    /**
     * Erstellt auf diesem Rechner ein Player-Objekt
     */
    Player createPlayer() throws java.rmi.RemoteException;

    /**
     * Erstellt auf diesem Rechner ein TablePart-Objekt
     */
    TablePart createTablePart() throws java.rmi.RemoteException;

    /**
     * Gibt Rückmeldung über diesen Rechner
     */
    boolean getState() throws java.rmi.RemoteException;
}
```

Listing 13: WebServerCommunication.java

```
package ligretto;

/*
 * Memory-Bedarf: je nach Implementation,
 * mindestens aber Referenzen auf alle Communication-Interfaces
 * der Rechenclients.
 */

public interface WebServerCommunication extends java.rmi.Remote {

    /**
     * Meldet einen Rechner beim Server an
     */
    void signOn(Communication client) throws java.rmi.RemoteException
        ;

    /**
     * Meldet einen Rechner beim Server ab
     */
    void signOff(Communication client) throws java.rmi.
        RemoteException;
}

```

## 5 Analyse

### 5.1 CPU-Auslastung

Die CPU-Auslastung ist abhängig von der Implementation der KI. Die Verwaltung der Spiel-Objekte erfordert im Vergleich dazu verschwindend wenig CPU-Zeit.

### 5.2 Speicherbedarf

Der Speicherbedarf der Simulation hängt stark von der jeweiligen Implementierung der Interfaces ab. Grundsätzlich werden aber, wo immer möglich nur Referenzen auf die benötigten Objekte gehalten, die eigentlichen Daten befinden sich in einer anderen Komponente, also unter Umständen auf einem anderen Rechner. Die wesentlichen, bestimmbaren Daten sind nachfolgend aufgeführt.

#### Auf dem Webserver

- Referenzen auf die RMI-Objekte für den Spielablauf (Ligretto-, Hand und Tischkarten). (4 Referenzen pro Spieler)
- Referenzen auf Communication RMI-Objekte (1 Referenz pro Rechenclient)
- Spielresultat

#### Auf den Rechenclients

- RMI-Objekte selber (siehe Listings)
- Referenzen auf andere Spieler und Teiltische

Ausserdem muss sowohl auf dem Server als auch auf den Clients der Speicherbedarf der Java Virtual Machine und der `rmiregistry` mit eingerechnet werden.

### 5.3 Netzwerk-Traffic

Grundsätzlich gehen alle Anfragen an Handkarten, Ligrettokarten etc. über das Netzwerk. Somit entsteht der gewünschte Effekt des Traffics auf dem Netzwerk, der Verzögerungen und des zu spät Kommens. Alten Ligrettohasen ist dieser Effekt auch in der Realität bekannt: Tausend (so meint man es zumindest) Hände kommen einem in den Weg und sind schneller als man selbst. Und man muss einen neuen Versuch starten.

Wie bereits im vorhergehenden Abschnitt erwähnt, ist die definitive Berechnung der Objektgrößen nicht trivial, da sie auch sehr stark von RMI abhängt. Um den Traffic wenigstens annäherungsweise zu berechnen, werden folgende Annahmen gemacht:

- Eine Anfrage (Request und Antwort) hat die Grösse von 100 Byte (Header- und RMI-Overhead, mehrfacher Informationsaustausch)
- Auf dem Spielfeld liegen pro Spieler immer 3 Stapel
- Ein Spieler legt pro Spiel insgesamt 20 Karten ab
- Die Erfolgsquote für abgelegte Karte/Alle Tischkarten abfragen liegt am Anfang bei 0.1. Für jeden zusätzlichen Gegenspieler dividiert sich die Erfolgsquote durch 2 (Effekt: Andere Spieler kommen dazwischen)
- Die Erfolgsquote wird jedoch pro Gegenspieler linear erhöht (Effekt: Je mehr Gegenspieler, desto mehr Karten auf dem Tisch)

### 5.3.1 Berechnung

Symbol	Bedeutung
$S$	= Anzahl Spieler
$P_E$	= Erfolgsquote; Wahrscheinlichkeit, eine Karte erfolgreich abzulegen
$K$	= Datenmenge pro abgelegte Karte in Byte
$M$	= Datenmenge über das ganze Spiel in Byte

$$P_E = \frac{0.1}{2^S} \cdot S \quad (1)$$

$$\begin{aligned} K &= \text{Stapel auf dem Tisch} \cdot \frac{1}{P_E} \cdot \text{Requestgrösse} \\ &= 3 \cdot S \cdot 10 \cdot 2^S \cdot \frac{1}{S} \cdot 100 \text{ Byte} = 2^S \cdot 3000 \text{ Byte} \end{aligned} \quad (2)$$

$$\begin{aligned} M &= S \cdot 20 \text{ Karten} \cdot \text{Datenmenge pro abgelegte Karte} \\ &= S \cdot 20 \cdot 2^S \cdot 3000 \text{ Byte} = 2^S \cdot S \cdot 60 \text{ Kbyte} \end{aligned} \quad (3)$$

Die asymptotische Komplexität der Funktion zur Berechnung des Datenmengen-Austausches ist somit  $O(2^n)$ . Dies wird auch aus der Abbildung 5.3 ersichtlich.

Die Simulationsdauer lässt sich annäherungsweise berechnen:

$$t = \frac{m}{\frac{v}{10 \frac{MBit}{MByte}}} = \frac{m \cdot 10 \frac{MBit}{MByte}}{v}$$

- $t$  = Simulationsdauer in s
- $m$  = Datenmenge in MByte
- $v$  = Netzwerkgeschwindigkeit in MBit/s

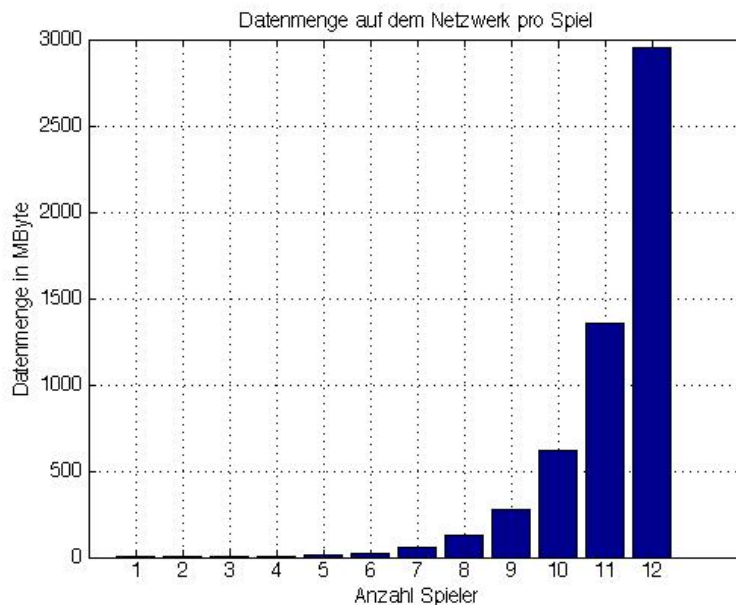


Abbildung 3: Datenmengen in Abhängigkeit der Anzahl Spieler

- Beispiel mit 4 Spielern: Datenmenge: 3.84 MB, Simulationsdauer auf 100MBit/s: ca. 0.4s
- Beispiel mit 8 Spielern: Datenmenge: 123 MB, Simulationsdauer auf 1GBit/s: ca. 1.2s
- Beispiel mit 12 Spielern: Datenmenge: 2949 MB, Simulationsdauer auf 1GBit/s: ca. 40s

Dieser Traffic ist gewollt und trägt zur realitätsnahen Simulation bei. Bei einer konkreten Implementierung müsste aber sicher darauf geachtet werden, dass genügend Daten ausgetauscht werden, um so den gewünschten Effekt der Nicht-Determiniertheit zu erzielen.

## 6 Mengengerüst

### 6.1 Minimalanforderungen

Damit ein Spielverlauf einer Partie Ligretto berechnet werden kann, ist mindestens 1 Rechenclient nötig. Auf diesem Rechner sind dann alle RMI-Objekte. Jedoch ergeben sich dann keine verschiedene Laufzeiten über das Netzwerk und somit ist das Spielresultat nur abhängig von der zufälligen Mischung der Karten und des zufälligen Aufrufs der RMI-Objekte (Thread-Abarbeitung). Um ein Spiel auch durchführen zu können, wenn keine Rechenclients vorhanden sind, könnte diese Aufgabe der Webserver übernehmen.

### 6.2 Maximale Anzahl Rechenclients

Da es für jeden Ligretto-Spieler 4 Teilaufgaben gibt (Spieler, Handkarten, Ligrettokarten, Tischkarten) werden maximal 4 x Anzahl Ligrettospieler Rechenclients benutzt. Bsp: Für eine Spielsimulation mit 6 Spielern macht es keinen Sinn mehr, als 24 Rechenclients bereitzustellen. (vgl. auch Abschnitt [2.4.1](#))

### 6.3 Skalierbarkeit

Die Skalierung ist nur durch die Kommunikation zwischen den Objekten beschränkt, die jedoch für die realitätsnahe Simulation erforderlich ist (ein Spiel mit 20 Spieler würde in der Realität ebenfalls langsamer ablaufen als ein Spiel mit 4 Spieler).

## 7 Anhang

### Abbildungsverzeichnis

1	Ablauf der Simulation . . . . .	3
2	Aufteilung der Software-Komponenten auf einzelne Clients . . . . .	6
3	Datenmengen in Abhängigkeit der Anzahl Spieler . . . . .	16

### Listings

1	Starten der Client-Software . . . . .	4
2	Anmelden beim Server . . . . .	5
3	Erstellen der Objekte auf dem Server . . . . .	7
4	CardColor.java . . . . .	9
5	Card.java . . . . .	9
6	ResultPart.java . . . . .	9
7	GameLeader.java . . . . .	10
8	Player.java . . . . .	10
9	LigrettoCards.java . . . . .	11
10	HandCards.java . . . . .	11
11	TablePart.java . . . . .	12
12	Communication.java . . . . .	13
13	WebServerCommunication.java . . . . .	14

## Index

Handkarten, 4, 5

Künstliche Intelligenz, 4, 5, 8

Ligretto-Karten, 4, 5

Spieler, 5

Spilleiter, 4, 5

Teiltisch, 5

Websserver, 3, 4