Systementwurf

Projektteam

Mitglieder: - Nico Günzel - Daniel Lutschinski - Tim Härle - Jonas Graf

Teamleiter: - Jonas Graf

Zuständigkeiten: - **Client**: Tim Härle und Jonas Graf

- **Server**: Nico Günzel und Daniel Lutschinski

Aufbau des Systems

Prozesse und Threads

Client-Prozess:

Der Client stellt die GUI zum Quiz zur Verfügung. Während der Vorbereitung zeigt der Client die verschiedenen Fragekataloge an, sowie die momentan angemeldeten Nutzer. Ausschließlich der Spielleiter kann über den Client einen Fragekatalog auswählen und das Spiel beginnen. Im laufenden Spiel zeigt der Client die Rangliste, sowie die Fragen und die Antwortmöglichkeiten an. Sobald der Benuter eine Antwort ausgewählt hat, wird dies dem Server gemeldet, der die Antwort auswertet und das Ergebnis dem Client zurück sendet .

Threads des Client-Prozesses :

GUI- Thread besteht aus der Ereignis-Schleife (guiMain). Reagiert selbstständig auf Benutzeraktionen durch Aufruf der entsprechenden Callback-Funktionen. Diese schicken dann eine zum Ergebnis passende Nachricht.

Listener-Thread reagiert auf Nachrichten des Servers, indem er die grafische Oberfläche über die GUI-API aktualsiert und evtl. den Fragewechsel-Thread aktiviert.

Fragewechsel-Thread wird vom Listener-Thread aktiviert, wenn dieser die Auswertung einer Antwort oder eine Timeout-Nachricht erhalten hat. Der Fw-Thread wartet daraufhin eine

kurze Zeit lang(3-5 Sekunden) und schickt dem Server dann eine Aufforderung, dass er die nächste Fragesenden soll. So wird erreicht, dass die Auswertung einen Moment lang auf dem Bildschirm erscheint, bevor die nächste Frage gestellt wird.

Server-Prozess:

Der Server verwaltet die Lister der Teilnehmer, die Fragen und die Rangliste. An einem Server können sich bis zu vier Clients über das Netzwerk anmelden. Dazu wird zunächst auf einem Rechner im Netzwerk über die Konsole der Server geladen. Dieser startet den Loader automatisch als eigenständigen Kindprozess auf dem selben Rechner. Während der Spielvorbereitung können sich dann die Clients von anderen Rechnern anmelden. Sobald das Spiel läuft ist dies nicht mehr möglich. Ein Client, der das dennoch versucht, wird mit einer Fehlermeldung abgewiesen.

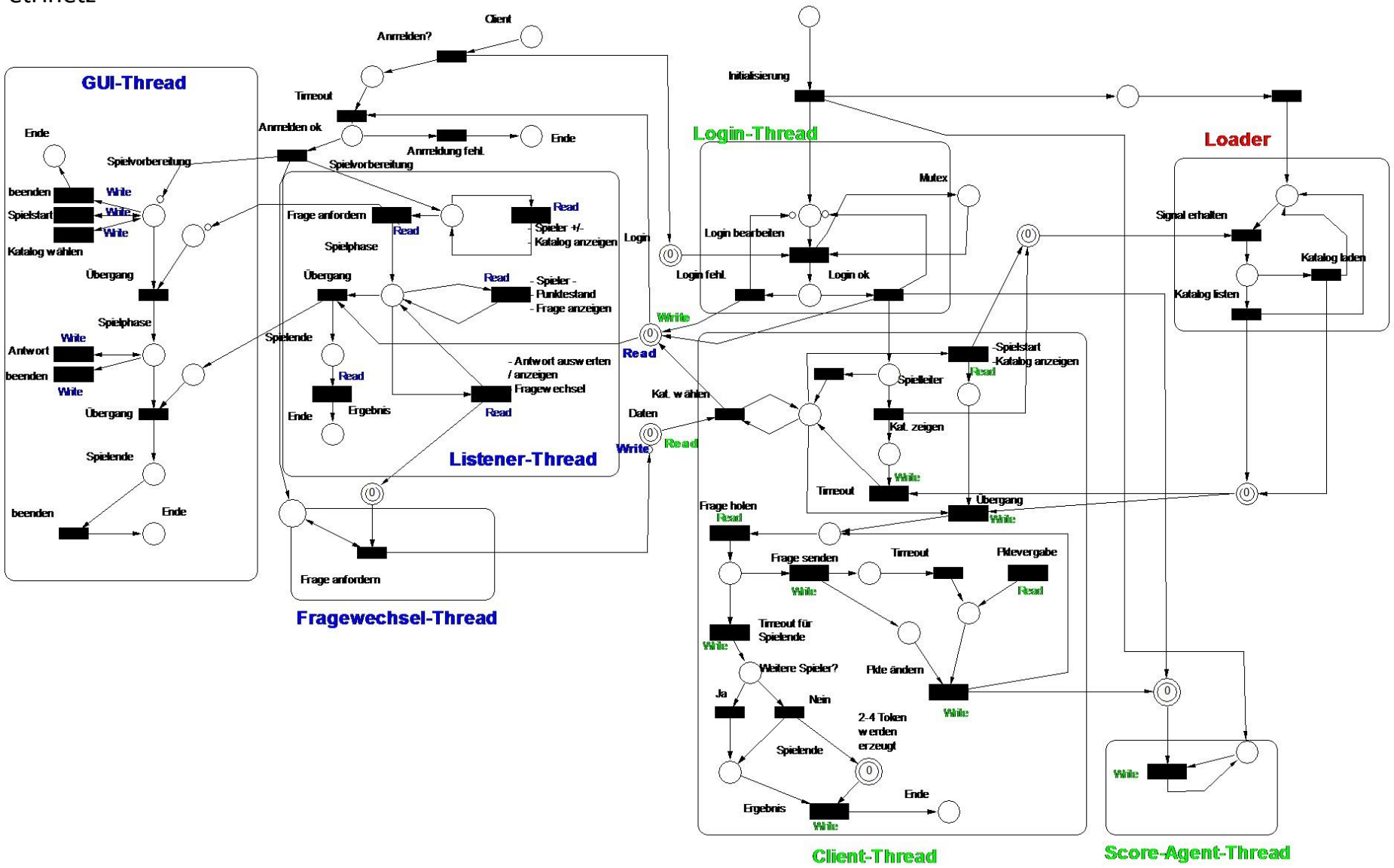
Threads des Server-Prozesses:

Der Server besteht aus einem Login-Thread, mehreren Client-Threads und einem Score-Agent. Bei jeder Anmeldung wird ein neuer Client-Thread erzeugt, der die Kommandos des Clients entgegen nimmt und weiterverarbeitet. Zudem wird der neue Benutzer in einer User-Liste eingetragen. Jeder Client kann auf diese Liste zugreifen. Die Liste enthält Userdaten (Name und Socket-Deskriptor), Punktestand sowie weitere Daten die für die Verwaltung nötig sind. Meldet sich ein Benutzer ab, so wird der zuständige Thread beendet und der Benutzer aus der Liste gelöscht. Verlässt der Spielleiter das Spiel, so beendet sich der Server. Es gibt also für jeden angemeldeten Benutzer genau einen Client-Thread. Für die einzelnen Spielphasen eines Clients wird immer der gleiche Client-Thread verwendet. Der Score-Agent ist ein eigenständiger Thread im Server. Wird der Score-Agent aktiv, so berechnet er eine sortierte Rangliste aus der User-Liste des Servers und schickt diese an alle angemeldeten Clients.

Loader-(Kind)Prozess:

Während der Initialisierung startet der Server den Loader als Kindprozess. Da der Loader über Standardein-und Ausgabe kommuniziert, erzeugt der Server zwei Pipes, in die jeweils einen der beiden Kanäle umleitet. Dadurch kann der Server über eine Pipe die Liste der Fragekataloge sowie den Erfolgstatus des Loaders entgegen nehmen.

Petrinetz


GUI-Thread
Ende
beenden
Spielstart
Katalog wählen
Spielvorbereitung
White
Übergang
Spielphase
Antwort
Spielende
Listener-Thread
Client
Anmelden?
Timeout
Anmelden ok
Anmeldung fehl.
Frage anfordern
Read
- Spieler +/-
- Katalog anzeigen
Login
Übergang
- Spieler -
- Punktestand
- Frage anzeigen
Write
- Antwort auswerten / anzeigen
Fragewechsel
Ergebnis
Fragewechsel-Thread
Initialisierung
Login-Thread
Mutex
Login bearbeiten
Login fehl.
Login ok
Loader
Signal erhalten
Katalog laden
Katalog listen
Spielleiter
-Spielstart
-Katalog anzeigen
Kat. wählen
Daten
Kat. zeigen
Frage holen
Frage senden
Timeout
Pktevergabe
Timeout für Spielende
Weitere Spieler?
Pkte ändern
Ja
Nein
2-4 Token werden erzeugt
Client-Thread
Score-Agent-Thread

Abläufe in den Prozessen/Threads

Synchronisation/Kommunikation

Login: Benutzereingaben beim Anmelden des Clients müssen vom Server im Login-Thread überprüft und ausgewertet werden. Ein Mutex stellt hierbei sicher, dass immer nur von einem User gleichzeitig ein Datenabgleich stattfindet. Der Client wartet hierbei auf eine Antwort vom Server. Der Datenaustausch erfolgt über Sockets.

Vorbereitungsphase: Nach erfolgreichem Login müssen sich GUI-, Listener-, Client- und Fragewechsel-Thread synchronisieren. Dabei erfolgt die Synchronisation zwischen Server und Loader über **Pipes** (Laden des Fragekatalogs), die Kommunikation zwischen Server und Client über Sockets (Senden der Fragen, Anzeigen aller angemeldeten Benutzer).

Änderung des Fragekatalogs: Ändert der Spielleiter den Fragekatalog, müssen alle beteiligten Mitspieler darüber informiert werden. Zunächst erfolgt eine Synchronisation zwischen Server und Loader (Laden des neuen Katalogs), anschließend werden die neuen Daten zwischen Server und Client ausgetauscht.

Spielstart: Nach der Vorbereitung beginnt das eigentliche Spiel. Jedem Benutzer werden dabei in zufälliger Reihenfolge die einzelnen Fragen aus dem vom Spielleiter gewählten Katalog gestellt (Synchronisation Client-Thread und Katalog-Thread). Für jede Frage gibt es ein Zeitlimit. Klickt der Spieler während dieser Zeit eine Antwort an, so wird überprüft, ob die Antwort richtig war (Server Read). Wenn ja, dann bekommt der Benutzer dafür eine Punktzahl, die davon abhängt, wie viel Zeit er zum Beantworten benötigt hat. Anschließend wird die Rangfolge der Spieler neu berechnet und an alle Teilnehmer geschickt (Synchronisation Spielstand-Thread und Client-Thread). Wählt der Spieler eine falsche Antwort oder läuft die Zeit für die Frage ab, so werden keine Punkte vergeben (Kommunikation Client-Thread und Spielstand-Thread).

Spielende: Hat ein Benutzer alle Fragen beantwortet, so befindet er sich in der Endphase. Während dieser wartet er, bis alle anderen Spieler ebenfalls mit allen Fragen fertig sind (Synchronisation Listener- und GUI-Thread). Dabei sollen Änderungen der Rangfolge weiterhin angezeigt werden. Am Ende wird jedem Teilnehmer seine Endplatzierung in einem Dialogfenster angezeigt. Danach werden die einzelnen Prozesse des Systems heruntergefahren.

Damit das System netzwerkfähig ist, erfolgt die Kommunikation zwischen Client und Server über Sockets.

Der Score-Agent

Der Score-Agent ist ein eigenständiger Thread im Server. Wird der Score-Agent aktiv, so berechnet er eine sortierte Rangliste aus der User-Liste des Servers und schickt diese an alle angemeldeten Clients. Folgende Ereignisse führen zur Aktivierung des Score- Agents:

- Ein neuer Teilnehmer hat sich (während der Vorbereitung) erfolgreich am Server angemeldet (Socket Client-Server).
- Ein Teilnehmer hat sich abgemeldet. (Kommunikation. Socket Client-Server)
- Ein Spieler hat eine Frage richtig beantwortet. Dadurch hat sich dessen Punktestand und gegebenenfalls auch die Rangfolge geändert.

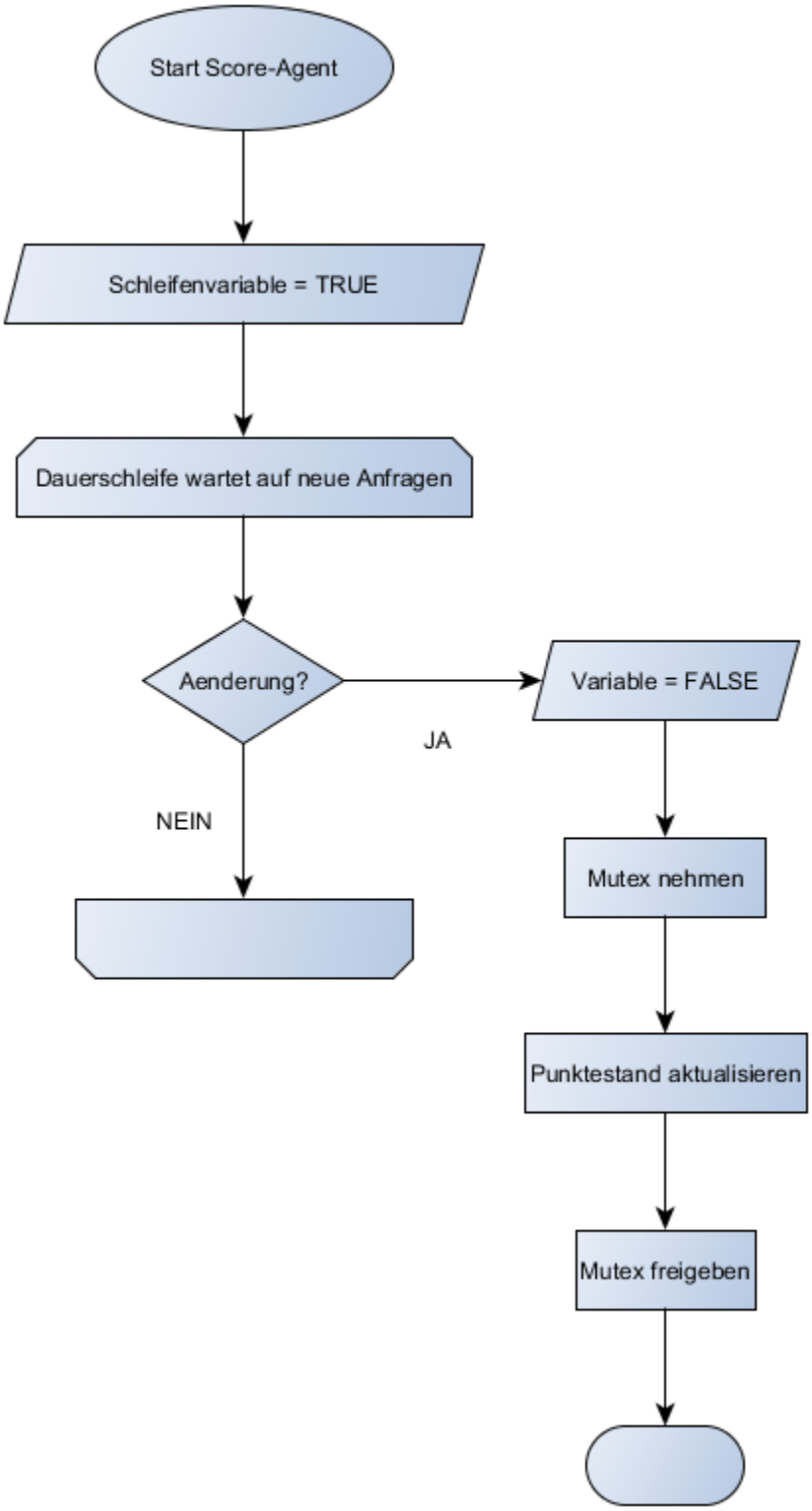
Der Score-Agent wartet dazu auf ein Semaphor. Dessen Freigabe erfolgt durch den Client-Thread, wenn eines der oben genannten Ereignisse auftritt. Der Score-Agent erzeugt daraufhin die sortierte Rangliste und schickt diese an alle teilnehmenden Clients.

Shared-Memory

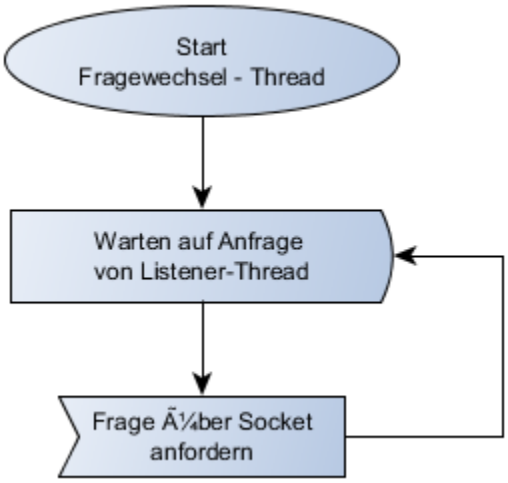
Die Quiz-Fragen erhält der Server vom Loader über einen gemeinsam verwendeten Speicherbereich (Shared Memory). Dieser wird vom Loader angelegt und nach Verwendung vom Server wieder gelöscht.

Ablaufdiagramme

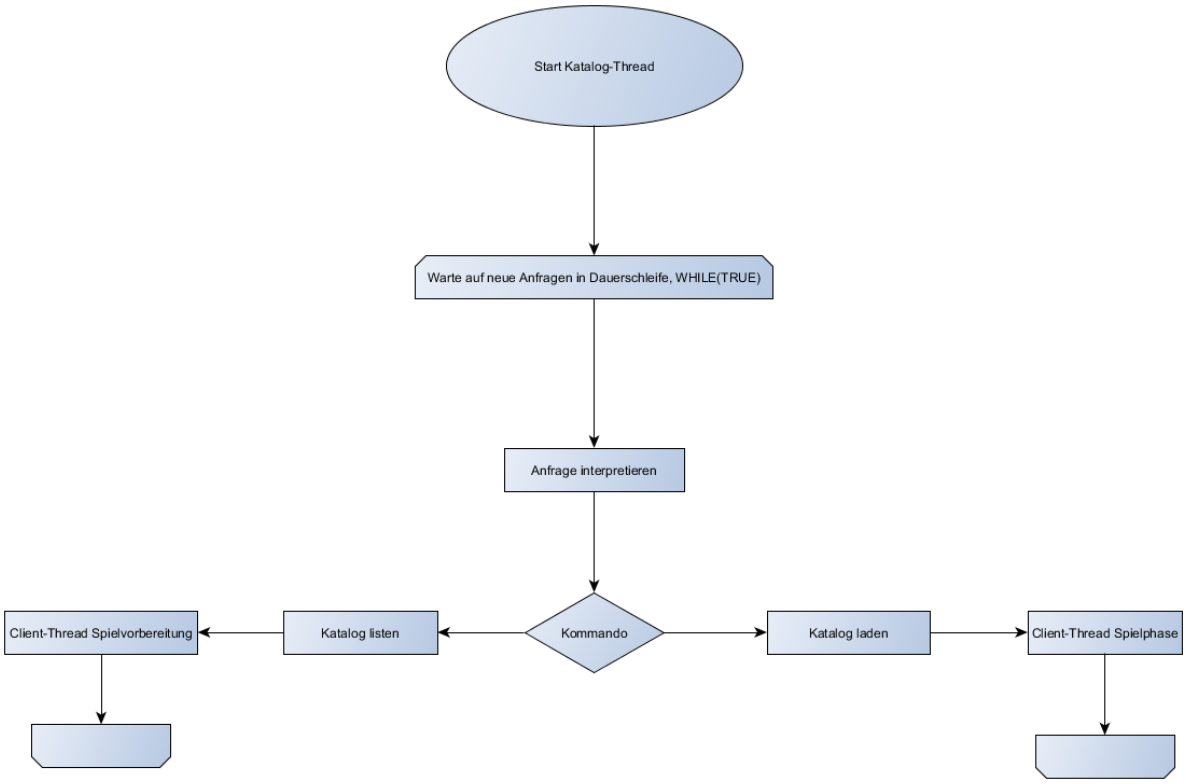
Score-Agent



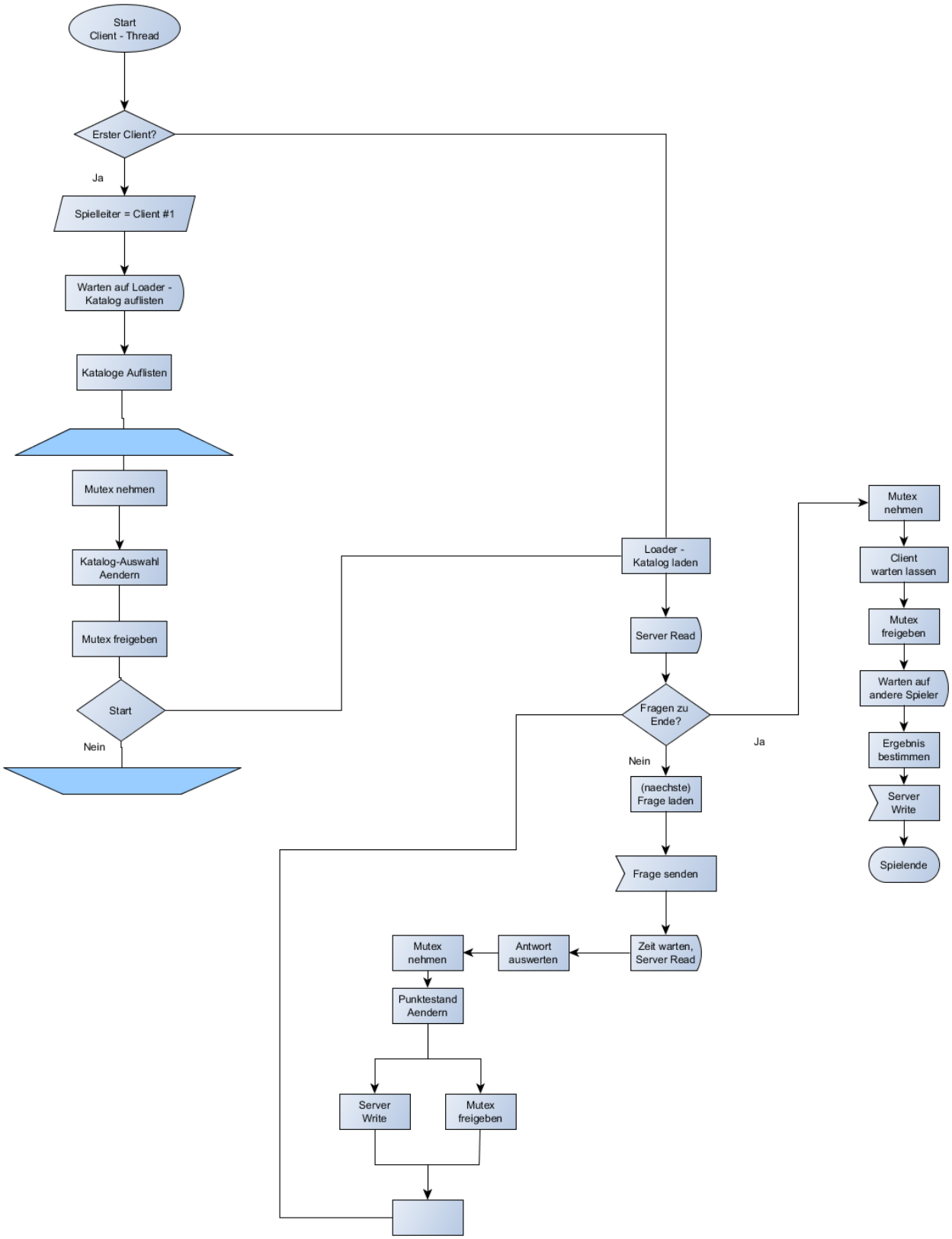
Fragewechsel Thread



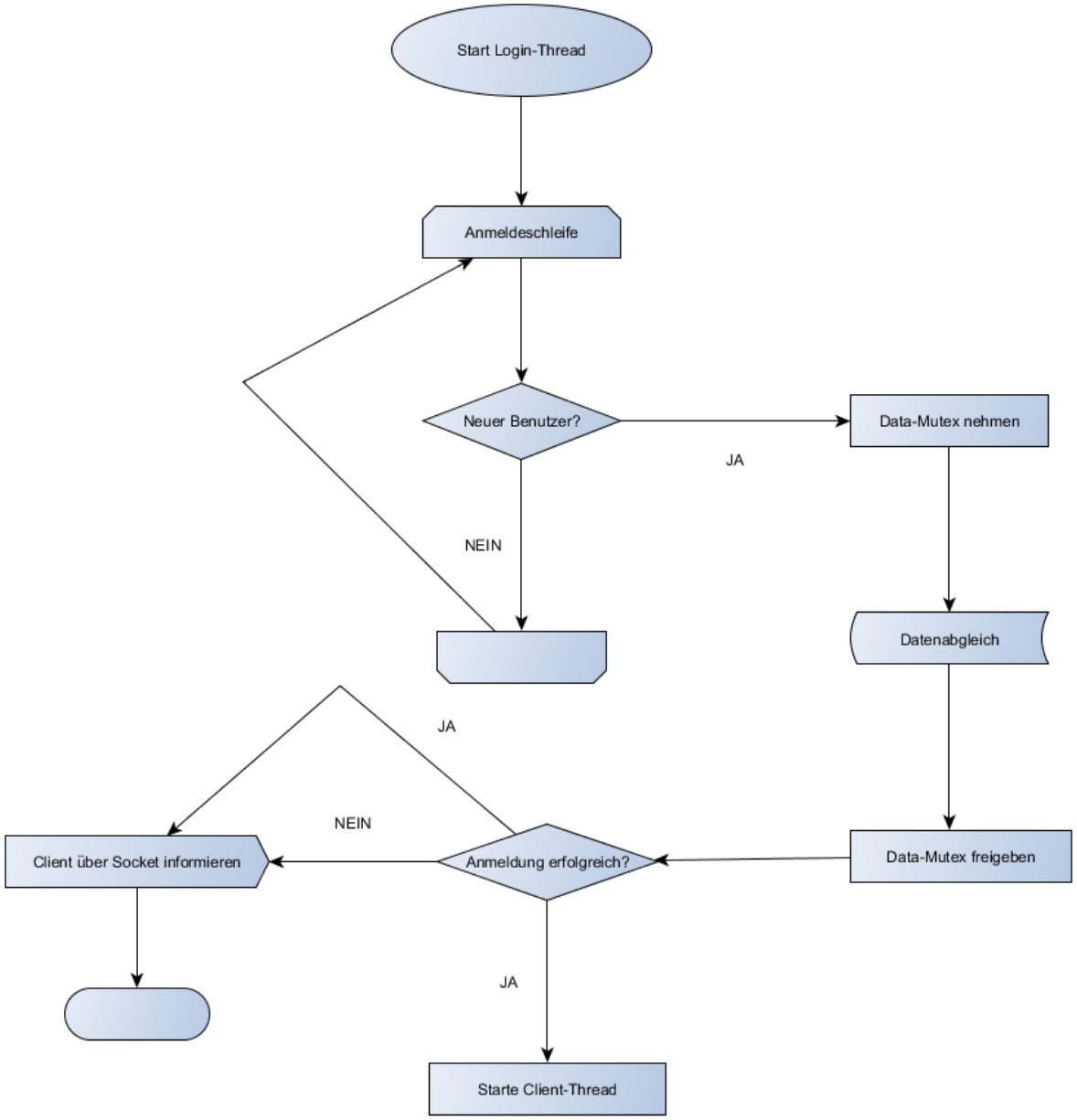
Katalog Thread



Client Thread



Login Thread



Listener Thread

Komplexe Datenstrukturen

Datenpakete zwischen Client und Server

Die Datenpakete müssen in der vom RFC vorgegebenen Form versendet werden. Jede Nachricht beginnt mit dem gleichen Header, welcher aus zwei Feldern besteht (3 Byte). Das erste Feld gibt den Typ der Nachricht an, das zweite die Länge der nachfolgenden Daten in Bytes.

```
0             7             15             23
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Type        |  Length                       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

Type:                   uint8_t, gibt die Art der Nachricht an
Length:                 uint16_t, Länge der nachfolgenden Zusatzdaten in
Bytes
```

Je nach Typ des Datenpakets werden die Nachfolgenden Daten unterschiedlich behandelt.

Datenpakete für die Spielvorbereitung

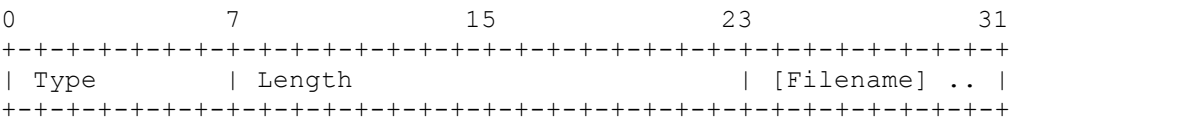
Typ 1 – Login Request (zum Server)

```
0             7             15             23              31
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Type        | Length                        | Name ........ |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

Type:                   1
Length:                 Länge des Namens (Length <= 31)
Name:                                Login-Name des clients, UTF-8, nicht
nullterminiert, maximal 31 Bytes
```

Typ 2 – Login Response Ok (zum Client)

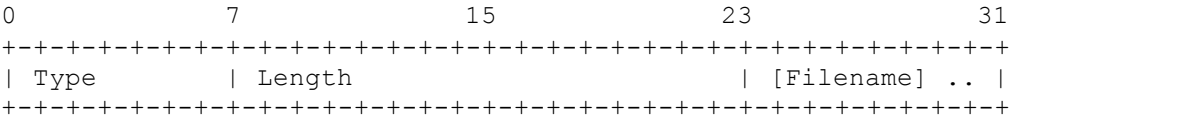
```
0             7             15             23              31
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Type        | Length                        | Client-ID     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

Type:                   2
Length:                 1
Client-ID:              uint8_t, zugewiesene ID des Clients
```

Typ 3 – Catalog Request (zum Server)



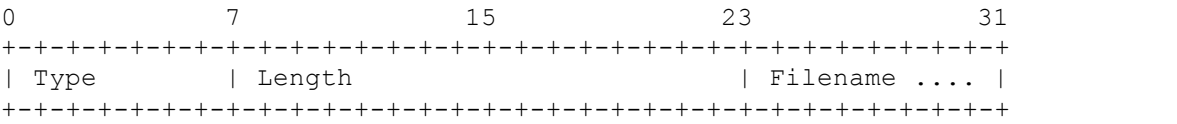
Type:	4
Length:	Länge des Dateinamens, oder 0 für Endemarkierung
Filename:	Dateiname eines Fragekataloges (UTF-8, nicht null-terminiert), oder leer als Kennzeichnung für Ende der Auflistung

Typ 4 Catalog Response (zum Client)



Type:	4
Length:	Länge des Dateinamens, oder 0 für Endemarkierung
Filename:	Dateiname eines Fragekataloges (UTF-8, nicht null-terminiert), oder leer als Kennzeichnung für Ende der Auflistung

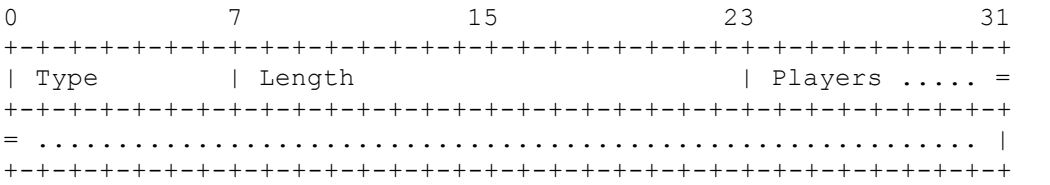
Typ 5 Cataloge Change (Beidseitig)



Type:	5
Length:	Länge des Dateinamens
Filename:	Dateiname des gewählten Fragekataloges (UTF-8, nicht nullterminiert)

Datenpakete während des Spiels

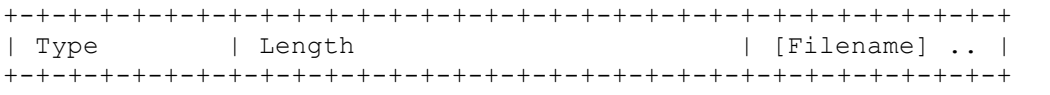
Typ 6 Player List (zum Client)




```
Type:                   6
Length:                 37*Spieleranzahl (maximal 4*37 = 148)
Players:                Liste aller derzeit angemeldeten Benutzer, siehe
unten

Aufbau der Spielerliste "Players":

        32 Bytes        Spielername 1 (UTF-8, nullterminiert)
         4 Bytes        Punktestand Spieler 1, vorzeichenlos
         1 Byte         ID Spieler 1

        32 Bytes        Spielername 2 (UTF-8, nullterminiert)
         4 Bytes        Punktestand Spieler 2, vorzeichenlos
         1 Byte         ID Spieler 2
```

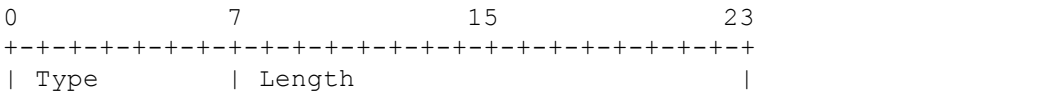
Sortiert nach aktuellem Punktestand!

Typ 7 Start Game (beidseitig)



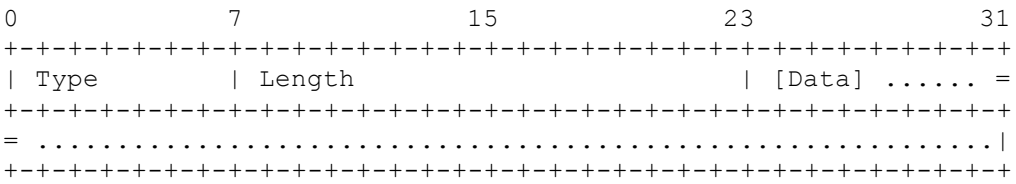

```
Type:                   7
Length:                 Länge des Dateinamens
Filename:               Dateiname des zu spielenden Fragekataloges (UTF-8,
nicht
                        nullterminiert), kann beim Versand Server ==>Client
                        (nicht Client ==> Server!) auch leer gelassen       werden
```

Typ 8 Question Request (zum Server)




```
Type:                   8
Length:                 0
```

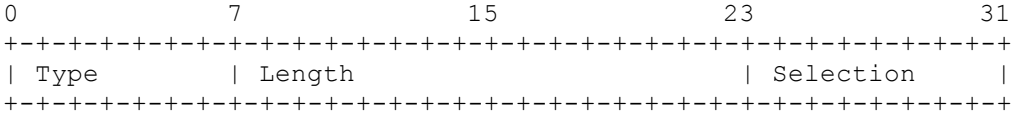
Typ 9 Question (zum Client)




```
Type:                   9
Length:                 769 oder 0 (falls keine Fragen mehr)
Data:                   Eine Struktur, die wie unten angegeben aufgebaut
ist,
                        oder leer falls Ende des aktuellen Kataloges
erreicht
```

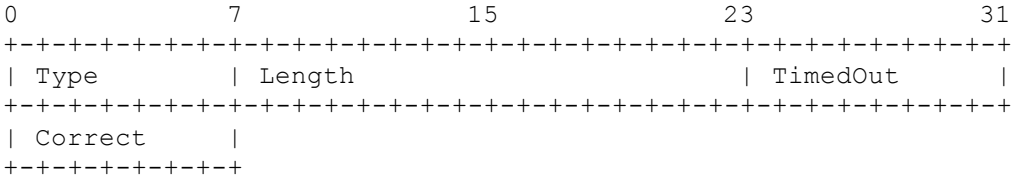
```
Aufbau des Data-Felds:
        256 Bytes       Text der Fragestellung (UTF-8, nullterminiert)
        128 Bytes       Antworttext 1 (UTF-8, nullterminiert)
        128 Bytes       Antworttext 2 (UTF-8, nullterminiert)
        128 Bytes       Antworttext 3 (UTF-8, nullterminiert)
        128 Bytes       Antworttext 4 (UTF-8, nullterminiert)
          1 Byte        Zeitbegrenzung in Sekunden
```

Typ 10 Question Answered (zum Server)




```
Type:                   10
Length:                 1
Selection:              uint8_t, Index der vom Benutzer gewählten Antwort-
                        möglichkeit (0 <= Selection <= 3)
```

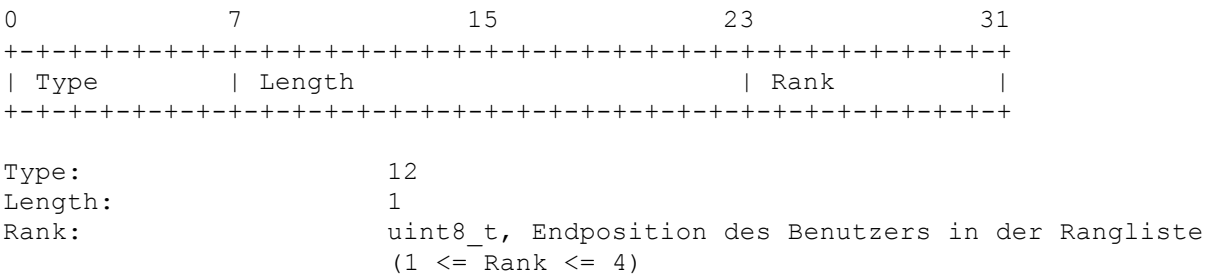
Typ 11 Question Result (zum Client)




```
Type:                   11
Length:                 2
Selection:              uint8_t, wenn Timeout für Frage erreicht wurde
                        ungleich 0, sonst 0
Correct:                uint8_t, Index der richtigen Antwort (0 <= Correct  <= 3)
```

Datenpakete für das Spielende

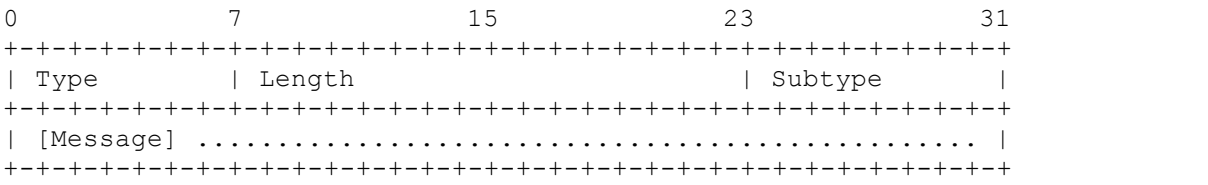
Typ 12 Game Over (zum Client)




```
Type:                   12
Length:                 1
Rank:                   uint8_t, Endposition des Benutzers in der Rangliste
                        (1 <= Rank <= 4)
```

Datenpakete für Fehlermeldungen

Typ 255 Error Warning (beidseitig)




```
Type:                   255
Length:                 1 + Länge(Message)
Subtype:                uint8_t
                        0 -> Warnung
                        1 -> fataler Fehler, Client muss sich beenden
Message:                Beschreibung des Fehlers im Textformat (UTF-8),
nicht
                        nullterminiert
```

Hinweise:

uint8_t = zahl mit 1 Byte vorzeichenlos

uint16_t= zahl mit 2 Byte vorzeichenlos

UTF-8 = Unicode 1 Byte/Zeichen

Zahlen in Big Endian

Shared Memory

Der Shared Memory für die Quizfragen wird vom vorgegebenen Loader Thread erstellt und besitzt daher eine vorgegeben Struktur für Quizfragen die im S.M. in einem Array liegen.

Question:

```
char question[QUESTION_SIZE];  //Text der Frage

char answers [NUM_ANSWERS] [ANSWER_SIZE] ; // NUM Antworten mit SIZE größe

uint8_t timeout ; //Zeit zum Beantworten der Fragen

int Correct; //richtige Antwort
```

Standardwerte:

QUESTION_SIZE **= 256 Byte,** ANSWER_SIZE **= 128 Byte,** NUM_ANSWERS **= 4 Antworten**

Bei QuestionMessage wird Correct weggelassen.

User Daten

Im Server werden die Daten der User gespeichert mehrere Threads müssen darauf zugreifen können. Sie bestehen aus zwei Teilen, dem Punktestand und den Login Daten.

Login-Daten: - Name - Socket Deskriptor - User ID

Punktestand - aktuelle Frage

Diese Daten werden in eine User Liste eingetragen. Implementierung als „struct“ variable in einem Array welches für die nötigen Threads erreichbar ist.

Programmmodule

Die Implementierung der Module wird mithilfe von „Pairprogramming“ realisiert.

Folgende Module werden benötigt:

für den Server:

```
catalog.c/catalog.h
clientthread.c/clientthread.h
login.c/login.h
main.c
score.c/score.h
user.c/user.h
```

für den Client:

```
fragewechsel.c/fragewechsel.h
listener.c/listener.h
main.c
guy_interface.h
```

gemeinsame Module:

```
rfc.c/rfc.h
util.cutil.h
question.h
server_loader_protocol.h
```

Folgende nach Themen sortierte Funktionen werden gebraucht:

Prozesse

```
execl-Familie          //Programm in Prozess laden und
                       //ausführen
fork                   //neuen Prozess aus aktuellem
                       //abspalten
getpid                 //eigene ID herausfinden
getppid                //ID des Elternprozesses herausfinden
wait,waitpid           //Prozess wartet auf Beenden des
                       //Kindprozesses/Prozess mit „id"
exit                   //aktuellen Prozess beenden
```

Threads

```
pthread_create         //Erzeugen eines Threads
pthread_self           //Rückgabe der Thread-ID
pthread_equal          //Thread-IDs auf Gleichheit
                       //untersuchen
pthread_exit           //Thread terminieren
pthread_join           //Auf Terminierung eines Threads
                       //warten
```

WICHTIG: **Alle Programme mit „-pthread" kompilieren und linken**

Signale

```
sigaction              //neuen Signalhandler einrichten
kill                   //Signal an Prozess senden
pthread_kill           //Signal an Thread desselben
                       //Prozesses senden
pthread_sigmask        //Signalmaske des aktuellen Prozesses
ändern/abfragen
```

ACHTUNG: **einige Aufrufe erwarten Signalmenge „sigset_t" als Parameter**

Synchronisierung

Mutexe:

```
pthread_mutex_t mutex    //Anlegen einer globalen Variable
pthread_mutex_init       //Initialisierung
pthread_mutex_lock       //Sperrung
pthread_mutex_unlock     //Freigabe
pthread_mutex_destroy    //Vernichtung
```

Semaphore

```
sem_init            //Initialisierung
sem_wait            //Dekrementieren, ggf. warten
sem_post            //Inkrementieren
sem_getvalue             //Abfrage des aktuellen Wertes
sem_destroy              //Löschung
sem_open            //Semaphor öffnen/erzeugen
sem_unlink               //benannten Semaphor löschen
sem_close           //benannten Semaphor schließen
```

ACHTUNG: Option -lrt muss beim Linken hinzugefügt werden

Ein- und Ausgabe

```
read                     //aus Datei lesen
write                    //in Datei schreiben
fstat                    //Informationen abfragen
pselect                  //Ablauf einer Zeitspanne oder
                         //Deskriptoren warten
clsoe                    //Filedeskriptor schließen
lseek                    //Schreib-/Leseposition
                         //abfragen/ändern
mmap                     //Datei in Adressraum einbinden
munmap                   //Datei aus Adressraum entfernen
open                     //Datei öffnen
unlink                   //Datei löschen
pipe                     //pipe erzeugen
```

Sockets

```
getaddrinfo          //Umwandlung eines Rechnernamens in
                     //eine Socket-Adressstruktur
freeaddrinfo         //Freigabe der gelieferten (von
                     //getaddrinfo) Daten
getnameinfo          //Socket-Adressstruktur in „lesbaren“
                     //Namen umwandeln
gai_strerror         //Fehlermeldung für fehlgeschlagene
                     //Namensauflösung
socket               //Socket erzeugen
setsockopt           //zusätzliche Optionen setzen
bind                 //Socket einen Port/Adresse zuweisen
listen               //Warten auf Verbindungen für Socket
                     //aktivieren
accept               //eine Verbindung abwarten, annehmen
                     //und neuen Socket erzeugen
connect              //zu einen Rechner verbinden
recv                 //Daten empfangen
send                 //Daten über Socket versenden
```

Shared Memory

```
shm_open             //Objekt öffnen/erzeugen
ftruncate            //Größe eines neuen Objekts setzen
mmap                 //Objekt in Adressraum des aktuellen
                     //Prozesses einbinden
munmap               //... aus Adressraum entfernen
close                //Objekt schließen
shm_unlink           //Objekt löschen
```

ACHTUNG: **Option -lrt beim Linken hinzufügen**

Message Queues

```
mp_open             //öffnen/erzeugen
mp_receive          //Nachricht aus einer MQ holen
mp_send             //Nachricht in MQ stellen
mq_close            //schließen
mq_unlink           //zerstören
```

ACHTUNG: **Option -lrt beim Linken hinzufügen**