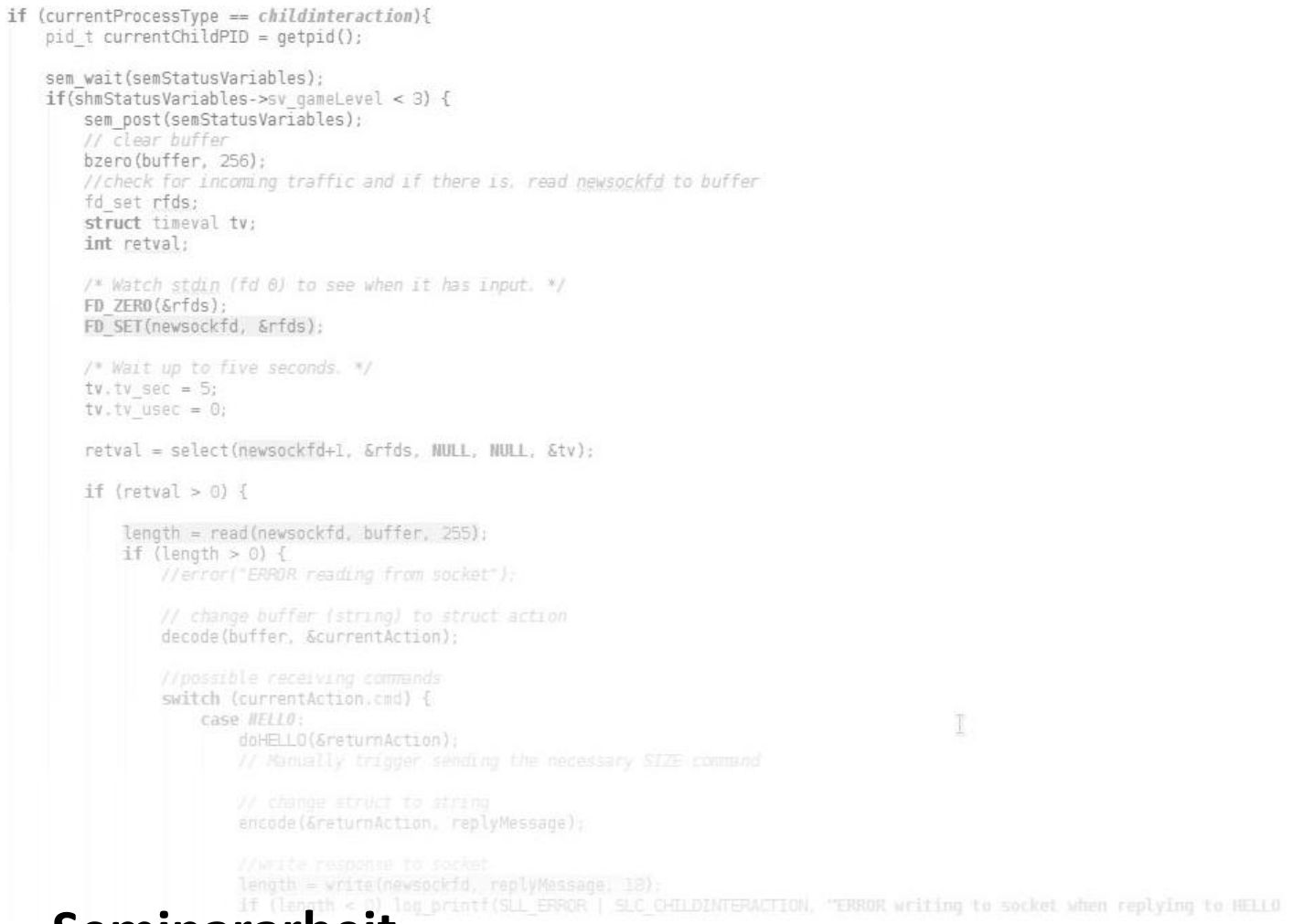

Seminararbeit

Concurrent C Programming

Désirée Sacher, ZHAW

8. Semester

21. Juni 2015

Inhaltsverzeichnis

1. Zusammenfassung

Im Rahmen der Seminararbeit „Concurrent C Programming“ sollte ein Spiel nach Vorgaben des Lehrers umgesetzt werden. Die Herausforderung lag dabei im Organisieren eines parallel zugegriffenen Speicherbereiches, über welchen der Spielstand festgehalten werden sollte.

Der Empfehlung des Lehrers, Forks und Shared Memory zu verwenden, wurde gefolgt, da damit in der Startphase keine unüberwindbaren Probleme identifiziert werden konnten. Für das Projekt wurden zuerst alle Anforderungen zusammengetragen und nach Entwurf von logischen Verantwortungsbereichen ein Entwicklungs- und Ressourcenplan entworfen. Obwohl der Projektfortschritt über längere Zeit knapp hinter dem vorgesehenen Plan lag, konnte durch rechtzeitiges Einplanen einer Reserve vor Abgabe des Projekts, das Ziel doch noch zeitgerecht erreicht werden.

Die klare und logische Trennung der Entwicklungsschritte zu Beginn des Projekts, vor allem am Anfang auf die Zurechtlegung eines passenden Systemdesigns, machte zudem weitere Entscheidungen einfacher, wodurch nachfolgende Herausforderungen schneller überwunden werden konnten.

Die Implementierung ist auf Github[1] zu finden.

[1] Siehe Literaturverzeichniseintrag L5.

2. Einleitung

2.1. Aufgabenstellung

Für dieses Projekt wurden das Ziel und die Anforderungen in einer Readme vorgegeben. Die Anforderungen an das Programm stellen sich wie folgt zusammen.

Spielregeln:

1. Ziel ist die Eroberung aller Felder des Spielfeldes durch einen Spieler (Client)
2. Alle Clients müssen sich an ein vordefiniertes Protokoll halten, wobei jeder Befehl mit „\n“ abgeschlossen wird und kein Befehl länger als 256 Zeichen sein darf (inkl. „\n“)
3. Es darf jeweils nur 1 Kommando gesendet werden und es muss auf die Antwort gewartet werden
4. Nur der schnellste Client „gewinnt“ bei einem Feld Zugriff, alle anderen erhalten eine „nicht erfolgreich“ Meldung (NACK) als Antwort
5. Der Server prüft alle y Sekunden den aktuell eingefrorenen Spielfeldstatus und stellt so einen allfälligen Gewinner fest

Vorgegebene Bedingungen:

1. Es gibt keinen globalen Lock auf alle Spielfelder (Client-seitig)
2. Der Server speichert den Namen des Feldbesitzers
3. Die Kommunikation findet über TCP/IP statt
4. Für jede Verbindung wird ein Prozess (Fork) oder Thread erstellt
5. Die Debug- und Fehler-Meldungen werden auf STDERR ausgegeben
6. Es können während des Spiels neue Spieler hinzukommen oder Spieler das Spiel verlassen
7. Das Spielfeld ist zwischen 4 und 256 Felder gross (der Maximalwert wurde an der Kick-Off Veranstaltung mündlich festgehalten)
8. Bei der Abgabe wird ein Makefile mitgeliefert, welches das Spiel kompiliert

Die technische Herausforderung des Projekts wurde definiert im Koordinieren der gleichzeitigen Zugriffe aus mehreren Prozessen oder Threads auf dieselben Ressourcen (u.a. das Spielfeld).

2.2. Annahmen

1. Folgende Annahmen wurden für das Spiel getroffen:
2. Clients dürfen nicht gleich heissen (Unterscheidung anhand des Namens und der ID[2])
3. Noch freie Felder liefern „-“ als Name zurück (bei der STATUS-Anfrage)
4. Es werden keine andere Befehle zum Testen verwendet, damit die Protokollausnahme (der Server antwortet auf eine STATUS-Anfrage nur mit den Namen ohne einen Befehlstyp mitzusenden) korrekt interpretiert werden kann

[2] Die Übermittlung der Namen ist laut Protokoll nur über den TAKE Befehl vorgesehen, die ID-Reihenfolge hängt entsprechend davon ab.

3. Anleitung

3.1. Erzeugung von Server und Client

Das Makefile generiert beim Ausführen von *make* (ohne weitere optionalen Aufrufparametern) sowohl den Server „server“, als auch den Client „client“. Das Makefile kann zudem über „make clean“ das Verzeichnis, inklusive den Logfiles (welche auf „.log“ enden) aufräumen. Über weitere optionale *make*-Aufrufparameter „server“, „client“, „clean“, „clean-server“, „clean-client“ und „clean-logs“ kann zudem spezifischer auf die Erzeugung oder das Aufräumen Einfluss genommen werden.

Die vollständige Aufruf-Syntax ist wie folgt:

```
$ make [all|server|client|clean|clean-server|clean-client|clean-logs]
```

Abbildung 1: Aufruf-Syntax für *make*

3.2. Anwendung von Server und Client

Über folgende Startparameter kann der Server gestartet werden:

```
$ ./server <portnummer> <n> <y> [loglevel]
```

Abbildung 2: Startaufruf des Servers

Die Port-Nummer <portnummer> kann frei gewählt werden, soll vorzugsweise aber im höheren Bereich sein da viele tiefere Port-Nummer Standard-Services zugeordnet sind. Mit <n> wird die Spielfeldgrösse übergeben, welche nach Vorgaben einem Wert zwischen 4 und 256 entsprechen soll. Die Option <y> bestimmt das Zeitintervall, in welchem der Spielstand geprüft werden soll. Dieser wurde mit einem Wert zwischen 1 und 30 vorgegeben. Der optionale Log Level [loglevel] definiert in welchem Ausmass Log-Informationen ausgegeben werden sollen. Gültige Werte sind RELEASE oder DEBUG. Falls dieser Parameter nicht existiert, wird als Standard RELEASE gewählt, mit einer minimalen Ausgabe an Log-Informationen (wie z.B. der regelmässige Spielstatus).

Der Client muss zum Starten unbedingt die Server IP-Adresse <ip addresse> und den Ziel-Port <portnummer> kennen. Jegliche weitere Konsolenangaben sind optional und können auch nach Aufruf des Clients per Eingabeaufforderung festgelegt werden.

```
$ ./client <ip addresse> <portnummer> [clientname] [spielmodus]
```

Abbildung 3: Startaufruf des Clients

Der Client-Name [Clientname] kann beliebig gewählt werden, sollte aber 243 Zeichen nicht überschreiten, damit die Vorgabe an die Maximallänge eines Befehls eingehalten werden kann. Für den Spielmodus stehen diverse Spielmodi zur Verfügung:

```
Please select the strategy ID:
1: simple from 0/0 to n/n - loop
2: simple quick - loop
3: random
4: only STATUS check
5: inactive client
```

Abbildung 4: Client-Strategien

Die Strategien 1 und 2 haben den gleichen Weg, sind aber unterschiedlich implementiert und haben deshalb leichte Performanceunterschiede. Strategie 3 stellt einen anderen Ansatz dar auf Basis der zufälligen

Spielfeldelementwahl. Die übrigen zwei Strategien wurden entwickelt, um die Server-Applikation besser testen zu können. So testet Strategie 4 explizit das STATUS-Kommando (ohne wirklich zu spielen). Strategie 5 dient dazu den Sonderfall zu behandeln, wenn ein Client nicht kommuniziert (siehe auch Abschnitt 6.2). Die Strategien 4 und 5 helfen auch einen Sieger zu erzwingen (dann wenn nur ein Client Strategie 1, 2 oder 3 spielt), um somit auch den Anwendungsfall des Sieges testen zu können.

4. Architektur

In den nächsten Abschnitten werden die Architektur und die Umsetzungsmodelle für den Server beschrieben. Auf die Realisierung des Client wird nicht speziell eingegangen, da der Fokus des Projektziels auf der Umsetzung des Servers liegt.

Für eine verbesserte Leserlichkeit sind die Fehlerausgaben bei den Code-Beispielen aus der Dokumentation entfernt worden. Zudem werden im Prosatext selber definierte Namen von Variablen und Funktionen mittels „“ umklammert und Funktionen von öffentlichen Bibliotheken mit *Italic*-Schrift formatiert.

4.1. Systemdesign

Zu Beginn wurde entschieden die Applikation über verschiedene Prozesse und einem Shared Memory zu lösen und nicht dem Multithreading-Ansatz zu folgen. Das Systemdesign des Servers wurde in drei Verantwortungsbereiche unterteilt, welche gleich auch als Fork-Prozesse umgesetzt wurden:

Der „Gameplay“-Prozess soll die Kontrolle über den Spielablauf ausüben. Der „Sockethandler“-Prozess (vom „Gameplay“-Prozess „abgeforkt“) steht für neue eingehende Verbindungen zur Verfügung um den Aufbau einer anstehenden Client-Kommunikation zu initialisieren, ohne dabei den „Gameplay“-Prozess zu unterbrechen. Die „Clientinteraction“-Prozesse (n-Prozesse bei n-Clients) werden bei einer neuen Verbindung eines Clients erzeugt (also im vom „Sockethandler“-Prozess „abgeforkt“) und sind je einem Client zugeordnet (1:1-Beziehung) und führen die Kommunikation mit den Clients und Versuchen somit deren Spielzüge auf dem gemeinsamen Spielfeld um zusetzen.

Die folgende Tabelle gibt einen detaillierten Blick auf die Verantwortlichkeiten der drei Prozesstypen des Servers:

Tabelle 1: Unterscheidung Prozesstypen

Gameplay	Sockethandler	Clientinteraction
• Setzt alle relevanten Spielablaufparameter • Definiert den Startzeitpunkt für Spiel und somit für die Clients • Stellt einen allfälligen Gewinner in regelmässigen Abständen fest • Räumt den Server am Ende auf	• Regelt TCP/IP Kommunikation • Wartet auf neue Verbindungen • Erzeugt einen neuen Clientinteraction-Fork nach erfolgreicher Anmeldung eines neuen Clients	• Wird pro Client erstellt • Verarbeitet eingehende Kommandos und antwortet auf Anfragen (spielt somit die Client-Strategie auf Server-Seite)

Die Prozesstypen wurden dabei als *enum* definiert damit eine einfache Prüfung auf den Typ möglich war. Solch eine Prüfung findet statt, sobald die *while*-Schlaufe der *main*-Funktion im Server betreten wird, um die jeweiligen Verantwortlichkeiten unterscheiden zu können. Somit teilen sich alle Prozesse programmtechnisch dieselbe *while*-Schlaufe, wobei aber jeweils unterschiedliche Bereiche der Schlaufe je nach Prozesstyp durchlaufen werden.

Durch die saubere und strikte Trennung der Prozesse und den Verantwortungsbereichen war es einfacher möglich, den Code zu strukturieren. Auch wurden damit Fehler oder fehlende Funktionalitäten schneller offensichtlich und die Frage nach dem korrekten „Ort“ einer allfälligen Ergänzung konnte durch das konsequente Umsetzen des Systemdesign implizit beantwortet werden.

Die grundlegende Interaktion findet entsprechend dem folgenden Sequenzdiagram statt:

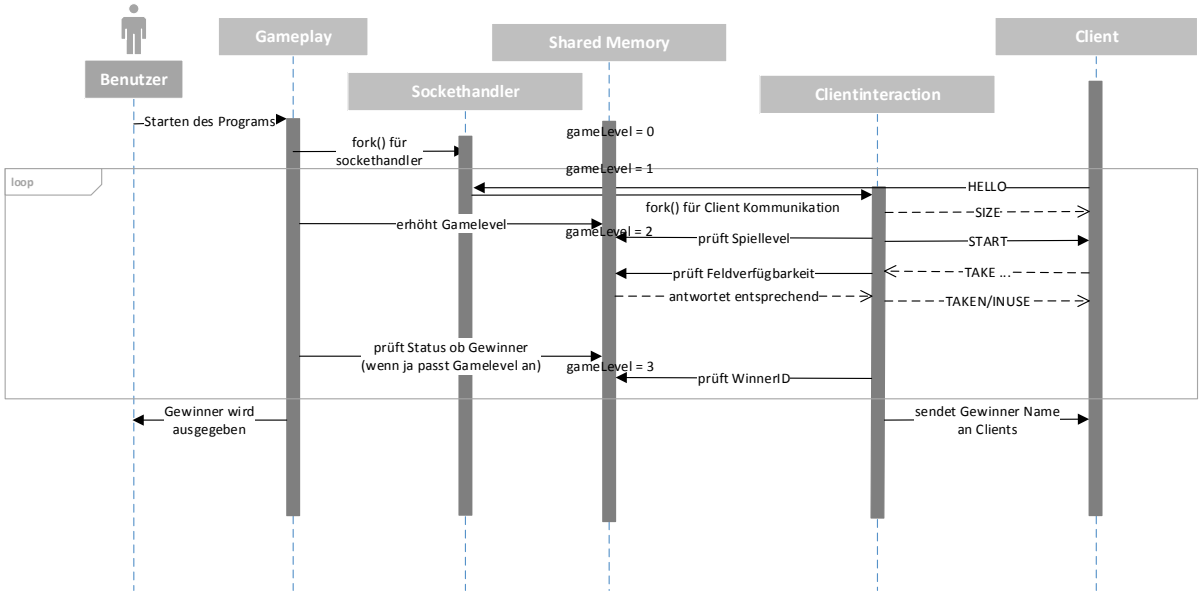

Abbildung 5: Sequenzdiagram Kommunikation zwischen Prozessen

Zentral ist die Verwendung des Game Level, welcher nur durch den „Gameplay“- und dem „Sockethandler“-Prozess gesteuert wird. Die Game Levels entsprechen den Phasen, in welchen sich das Spiel befindet. Bevor die *while*-Schlaufe betreten wird, wird der Game Level auf 1 gesetzt um anzuzeigen, dass Verbindungen angenommen werden. Die Stufe 2 wird erreicht sobald genügend Clients zum Spielstart angekommen sind und das tatsächliche Spiel losgeht. Die Stufe 3 wird erreicht wenn ein Gewinner gefunden wird und die Clients ordentlich terminiert werden. Alternativ wird bei einem Abbruch Stufe 0 gesetzt um die *while*-Schlaufe zu verlassen und jegliche angelegte Ressourcen doch noch aufzuräumen.

4.2. Socket-Kommunikation

Die Socket-Kommunikation wurde als *SOCK_STREAM* eröffnet, um der Anforderung an TCP gerecht zu werden. Der Port wird mit den Startparametern der Applikation übergeben.

Damit eine Verbindung zwischen dem Server und den Clients zustande kommen kann, muss auf beiden jeweils ein Socket geöffnet und an einen Port gebunden werden, welchem dem jeweiligen Programm die Möglichkeit gibt, über diese Art von Stream zu kommunizieren.

Für den Aufbau dieser Kommunikation wurde der Anleitung aus dem Literaturverzeichniseintrag L2 gefolgt.

Die folgenden Zeilen beschreiben den Socket-Aufbau und das Binding des Sockets auf den Port:

```
1   // SOCK_STREAM for TCP, Domain for Internet, protocol chosen automatically
2   sockfd = socket(AF_INET, SOCK_STREAM, 0);
3   if (sockfd < 0) {
4       keepRunning = 0;
5   }
6   // set sizeof(serv_addr) of bytes to zero
7   bzero((char *)&serv_addr, sizeof(serv_addr));
8   // configure serv_addr structure object with server details
9   serv_addr.sin_family = AF_INET;
10  serv_addr.sin_addr.s_addr = INADDR_ANY;
11  serv_addr.sin_port = htons(portno);
12  // bind  socket file descriptor to serv_addr, with length sizeof(serv_addr)
13  and if error print out
14  if (bind(sockfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0) {
15      keepRunning = 0;
16  }
17  listen(sockfd,5);
18  clilen = sizeof(cli_addr);
```

Abbildung 6: Socket-Aufbau

Auf Zeile 2 (Abbildung 6) wird ein socket eröffnet und der dazugehöre File Descriptor in der Variable „sockfd“ gespeichert. Nach der dazugehörigen Fehlerbehandlung wird die *serv_addr* Struktur mit 0 überschrieben, damit danach eine frische Konfiguration gesetzt werden kann. Das Binding des Sockets auf den Port findet in der Zeile 14 (Abbildung 6) statt.

Anschliessend können auf dem socket die ankommenden Pakete mit *accept* (Zeile 1, Abbildung 7) angenommen, mit *read* Daten gelesen (Zeile 8, Abbildung 7) werden. Die akzeptierte Socket-Verbindung mit dem Client wird durch den Socket File Descriptor „newsockfd“ repräsentiert. Die gelesenen Daten sind danach im Char Array „buffer“ zu finden und entsprechen im Normalfall einem Protokollkommando des Spiels, welches als String verpackt ist.

```
1   newsockfd = accept(sockfd, (struct sockaddr *)&cli_addr, &clilen);
2   if (newsockfd < 0) {
3       fprintf(stderr, "[PID:%d] ERROR on accept\n", getpid());
4   }
5   else {
6       childpid = fork();
7   ..
8   length = read(newsockfd, buffer, 255);
9   if (length > 0) {
10      decode(buffer, &currentAction);
```

Abbildung 7: Lesen von socket

4.3. Client/Server-Protokoll

Das bereits genau definierte Client/Server-Protokoll der Aufgabenstellung machte das Handling einfacher.

Es wurde entschieden zur besseren internen Verarbeitung die ankommenden und zu versendenden Befehle in eine Struct umzuwandeln und danach mit dieser „action“ Struct zu arbeiten. Die Struct „action“ umfasst dabei das Kommando „cmd“, sowohl zwei Integer-Parameter „iParam1“ und „iParam2“ und einen String-Parameter „sParam1“.

```
1   enum command {HELLO, SIZE, NACK, START, TAKE, TAKEN, INUSE, STATUS, END,
2   PLAYERNAME, ERROR};
3   ..
4   struct action {
5       enum command cmd;
6       int iParam1;
7       int iParam2;
8       char sParam1[256];
9   };
```

Abbildung 8: „action" Struct-Deklaration

Jegliche Kommandos des Spielprotokolls konnten somit auf die Struct „action“ entsprechend abgebildet werden. Durch diese strikte Aufteilung und Zuordnung wurde nicht nur die Fehlersuche vereinfacht, auch konnte die Verarbeitung besser strukturiert werden. Zudem ist eine Erweiterung einfach möglich (allenfalls auch durch Erweiterung des Struct „action“, falls neue Protokollkommandos mehr als die vorhandenen Parameter aufweisen sollten).

Die Befehlsübersetzung wurde in das File „grabProtocolTranslator.c“ ausgelagert, da einerseits an mehreren Stellen auf die Enkodierung („encode“) und die Dekodierung („decode“) von Protokollkommandos zugegriffen und andererseits dieselbe Funktionalität auch im Client genutzt werden sollte.

Bei der Dekodierung eines Protokollkommando-Strings wird dieser über *sscanf* aufgeteilt und die identifizierten Teil-Strings in vier lokalen Variablen zur Weiterverarbeitung zwischengespeichert (Zeile 5, Abbildung 9). Je nach identifiziertem Kommando, beispielsweise TAKE in Zeile 9 (Zeile 7, Abbildung 9), werden die Elemente der „action“ Struct entsprechend gesetzt (Zeilen 8 – 11, Abbildung 9). Am Ende der Dekodierung steht der Applikation das Protokollkommando also Struct zur weiteren Verarbeitung zur Verfügung.

```
1   void decode(const char* cmd, struct action* a) {
2       char mainCommand[255], proto1st[255], proto2nd[255], proto3rd[255],
3       proto4th[255];
4       ..
5       sscanf(cmd, "%s %s %s %s", proto1st, proto2nd, proto3rd, proto4th);
6       ..
7       else if(strcmp(proto1st,"TAKE") == 0) {
8           a->cmd = TAKE;
9           a->iParam1 = atoi(proto2nd);
10          a->iParam2 = atoi(proto3rd);
11          strcpy(a->sParam1, proto4th);
12      }
13      ..
14  }
```

Abbildung 9: Dekodierung des Char Arrays in eine „Action“ Struct

In umgekehrter Weise wird ein erzeugtes Kommando (z.B. eine Antwort) in Struct-Form per „encode“ wiederum in ein String (Char Array) umgewandelt, resp. wieder für die Socket-Kommunikation vorbereitet. Dabei wird die „action“ Struct entsprechend dem Kommando „a->cmd“ beispielsweise über *strcpy* (für HELLO; Zeile 4, Abbildung 10) oder *sprintf* (für TAKE; Zeile 8, Abbildung 10) entsprechend transformiert und in die „returnMessage“ gespeichert. Diese wird dann wiederum per Socket-Kommunikation an den Empfängerprozess gesendet.

```
1   void encode(struct action* a, char* returnMessage) {
2       switch (a->cmd) {
3           case HELLO:
4               strcpy(returnMessage, "HELLO\n");
5               break;
6           ..
7           case TAKE:
8               sprintf(returnMessage, "TAKE %d %d %s\n", a->iParam1, a->iParam2,
9               a->sParam1);
10              break;
11      ..
12  }
```

Abbildung 10: Enkodierung der „action“ Struct in einen Char Array

Das zentrale Element der Protokollkommandoverarbeitung ist an Funktionen nach folgendem Syntaxschema delegiert: doHELLO, doTAKE, etc. welche jeweils Parameter des empfangenen und zu verarbeitenden Protokollkommandos currentAction (vom Struct-Typ „Action“) übernehmen, das Kommando verarbeiten und eine Antwort als returnAction (ebenfalls vom Struct-Typ „Action“) retournieren.

4.4. Shared Memory-Verwendung

Durch die Duplizierung des Heap und des Stacks beim fork, werden alle vorher definierten Variablen, inklusive deren Inhalte an alle Prozesse vererbt. Alle Variablen die aber einer ständigen Änderung unterlagen, da sie vom Spielverlauf beeinflusst werden, sind im Shared Memory abgelegt.

Zur Verwaltung der zentralen und dynamisch sich ändernden Zustände wurden drei thematisch zusammengehörende identifiziert und im Shared Memory angelegt. Dies sind die Shared Variables, die Spielernamenliste und das Spielfeld selbst. Alle drei Bereiche sind mit eigenen Semaphoren geschützt, wobei für das Spielfeld die Semaphore pro Spielfeldelement erzeugt werden, um jeweils den Zugriff auf Spielfeldelementebene zu kontrollieren. Damit wird der Bedingung Rechnung getragen, für TAKE-Zugriffe keine globalen Spielfeld-Lock zu erzwingen.

4.4.1. Shared Variables

In der „sharedVariables“ Struct, werden der aktuelle Spiellevel „sv_gameLevel“ (1: warten auf Clients; 2: spielend; 3: erfolgreich beendet; 0: abgebrochen), die Anzahl aktueller Spieler „sv_numberOfPlayers“, die Anzahl bereits gespeicherter Spielernamen „sv_numberOfPlayerNames[3]“ und die ID des Gewinners „sv_winnerID“ per Integer-Werte festgehalten.

```
1   struct sharedVariables {
2       int sv_gameLevel;
3       int sv_numberOfPlayers;
4       int sv_numberOfPlayerNames;
5       int sv_winnerID;
6   };
```

Abbildung 11: Struct für Shared Variables

Der „sv_gameLevel“ und die „sv_winnerID“ werden grundsätzlich nur vom „Gameplay“-Prozess geschrieben, von den anderen Prozessen aber nur gelesen. Im Abbruchsfall darf der „Sockethandler“-Prozess die „sv_gameLevel“ Variable auf 0 setzen. „sv_numberOfPlayers“ wird vom „Sockethandler“-Prozess geschrieben und von allen „Clientinteraction“-Prozessen und dem „Gameplay“-Prozess gelesen. Die

[3] Anzahl, welche durch die im TAKE-Befehl übertragenen Namen inkrementiert wird. Die Spieler ID ist entsprechend der Reihenfolge der eingehenden Spielernamen.

„sv_numberOfPlayerNames“ wird von den „Clientinteraction“-Prozessen lesend und schreibend genutzt, der „Gameplay“-Prozess greift für die Statusausgaben lesend darauf zu.

Der Zugriff auf die „sharedVariables“ Struct wird mittels dem Semaphore „semStatusVariables“ geschützt.

4.4.2. Liste der Spielernamen

Die Liste der Spielernamen hat ebenfalls ihren eigenen Shared Memory-Bereich da die Spielernamen von mehreren „Childinteraction“-Prozessen beim jeweils ersten TAKE des stellvertretenen Clients registriert werden.

Zur Einfachheit wird die Grösse des Char Arrays „shmPlayerList", welche die Spielernamenliste repräsentiert, für den maximalen Fall an Clients ausgelegt (256 Clients, mit je einem maximalen Spielernamen von 243 Characters (inkl. ‘\0‘)). Der n-te (n: 0..255) Spielernamen liegt dabei beginnend an der Stelle shmPlayerList[n*256]. Auf diese Weise konnte auf ein dynamisches Vergrössern (resp. bei Bedarf auch Verkleinern) während der Laufzeit verzichtet werden, auch wenn mit diesem Lösungsansatz damit Speicher „verschwendet“ wird.

```
1   #define MAX_PLAYER_NAME_LENGTH 243
2   int shmPlayerListID;
3   int shmPlayerListMemSize;
4   char* shmPlayerList;
5   const char* SHMSN_PLAYER_LIST = "/PLAYERLIST";
6   ..
7   // create player list in shared memory
8   shmPlayerListMemSize = MAX_CLIENTS*MAX_PLAYER_NAME_LENGTH*sizeof(char);
9   // create segment and set permissions
10  if ((shmPlayerListID = shmget(SHMK_PLAYER_LIST, shmPlayerListMemSize,
11  IPC_CREAT | 0666)) < 0) {
12      keepRunning = 0;
13  }
14  // attach segment to data space
15  if ((shmPlayerList = (char*)shmat(shmPlayerListID, NULL, 0)) == (char*)-1) {
16      keepRunning = 0;
17  }
18  memset(shmPlayerList, 0, shmPlayerListMemSize);
```

Abbildung 12: Shared Memory-Allozierung der Spielernamenliste

Auf die Spielernamenliste greifen hauptsächlich die „Clientinteraction“-Prozesse zu, im Extremfall parallel, sowohl lesend wie auch schreibend. Für die Statusausgaben im „Gameplay“ Prozess wird auf den Bereich lesend zugegriffen. Der Zugriff auf die gesamte Spielernamenliste wird über den Semaphore „semPlayerList“ kontrolliert und geregelt.

4.4.3. Spielfeld

Das Spielfeld wird dynamisch anhand der Startparametergrösse (Zeile 21, Abbildung 13) generiert. Es wird als ein zusammenhängender Block von Integern abgebildet, wobei während der Laufzeit des Spiels jedes Spielfeldelement die ID (0..255) desjenigen Clients repräsentiert, der aktuell das Spielfeldelement besetzt. Ein Wert von -1 bedeutet, dass das Spielfeld noch unbesetzt ist.

Zuerst wird pro Spielfeldelement ein Semaphore mit dem dynamischen Namen „/PLAYFIELDELEMENT**I**“ (**I** steht für das i-te Spielfeldelement, wobei eine Transformation von zwei Dimensionen (X/Y) in eine Dimension (I) entsprechend I = Y*FIELDSIZE+X) erzeugt (Zeilen 6 - 16, Abbildung 13). Die erzeugten Semaphore selbst werden wiederum in einem eindimensionalen Array „semPlayfieldElement“ abgelegt (Zeile 12, Abbildung 13). Danach wird anhand der Spielfeldseitengrösse („FIELDSIZE“) ein entsprechend grosser Speicherbereich für das Spielfeld alloziert (Zeilen 19 – 28, Abbildung 13). Die Initialisierung erfolgt in Zeile 29 (Abbildung 13) und besagt, dass jedes Spielfeld noch unbesetzt ist.

```
1   const char* SHMSN_PLAYFIELD_ELEMENT_PREFIX = "/PLAYFIELDELEMENT";
2   #define SHMK_PLAYFIELD 3000
3   sem_t** semPlayfieldElement;
4   ..
5   // semaphores for playfield element
6   int i;
7   char shmmnPlayfieldElementName[256];
8   semPlayfieldElement = malloc(FIELDSIZE*FIELDSIZE*sizeof(sem_t));
9   for (i = 0; i < FIELDSIZE*FIELDSIZE; i++) {
10      sprintf(shmmnPlayfieldElementName, "%s%d",
11      SHMSN_PLAYFIELD_ELEMENT_PREFIX, i);
12      if ((semPlayfieldElement[i] = sem_open(shmmnPlayfieldElementName,
13                                    O_CREAT, 0600, 1)) == SEM_FAILED) {
14          keepRunning = 0;
15      }
16  }
17  ..
18  // create playfield in shared memory
19  shmPlayfieldMemSize = FIELDSIZE*FIELDSIZE*sizeof(int);
20  // create segment and set permissions
21  if ((shmPlayfieldID = shmget(SHMK_PLAYFIELD, shmPlayfieldMemSize, IPC_CREAT |
22                        0666)) < 0) {
23      keepRunning = 0;
24  }
25  // attach segment to data space
26  if ((shmPlayfield = (int*)shmat(shmPlayfieldID, NULL, 0)) == (int*)-1) {
27      keepRunning = 0;
28  }
29  memset(shmPlayfield, -1, shmPlayfieldMemSize);
```

Abbildung 13: Shared Memory des Spielfeldes

Durch die Zugriffskontrolle per individuellen Semaphoren pro Spielfeldelement kann ein Maximum an parallelen Zugriffen durch die Clients, repräsentiert durch die „Childinteraction“-Prozesse, auf das Spielfeld erreicht werden.

Einzig dem „Gameplay“-Prozesse ist es erlaubt das gesamte Spielfeld (durch Lock aller Spielfeldelement-Semaphore) zu locken. Dies ist aber auch notwendig, um einen in sich konsistenten Zustand des Spielfelds zu erhalten, um feststellen zu können ob ein Sieger vorhanden ist.

5. Wichtige Anwendungsfälle

5.1. Spielfeldelementbesetzung

Das Spielfeld wurde wie bereits erklärt als einfaches Integer Array konzipiert, bei welchem der Index des i-ten Spielfeldelements „index“ jeweils über die Seitenlänge des Spielfelds („FIELDSIZE“) und die Koordinaten „x“ und „y“ berechnet wird.

Für ein TAKE wird entsprechend zuerst der passende „index“ berechnet (Zeile 1, Abbildung 14). Der Zugriff auf die Felder wird über *sem_trywait* (Zeile 7, Abbildung 14) kontrolliert, um den Spielfluss der Clients nicht zu blockieren. Je nach Ausgang des Lock-Versuchs mittels *sem_trywait* wird die Antwort TAKEN bei einem erfolgreichen Zugriff zurückgemeldet oder INUSE, falls das Feld gerade gesperrt ist (also ein anderer Client parallel das Spielfeldelement in Besitz nimmt). Der verantwortliche Semaphor wird dabei über den zuvor berechneten Indexwert eruiert (auch Zeile 7, Abbildung 14).

```
1   int index = y*FIELDSIZE+x;
2   ..
3   if (!existPlayer(shmStatusVariables,playerName)) {
4       addPlayer(shmStatusVariables, playerName);
5   }
6   ..
7   if (sem_trywait(semPlayfieldElement[index])== 0) {
8       shmPlayfield[index] = getPlayerID(shmStatusVariables,playerName);
9       sem_post(semPlayfieldElement[index]);
10      returnAction->cmd = TAKEN;
11  }
12  else {
13      returnAction->cmd = INUSE;
14  }
```

Abbildung 14: TAKE-Zugriff auf Spielfeldelement

Der Grund für den Entscheid eines eindimensionales Array, anstatt eines zweidimensionalen, liegt einerseits in der einfacheren Anwendung und andererseits in der transparenteren Speicherallozierung, da der C Compiler ein zweidimensionales Array automatisch in ein eindimensionales Array umwandelt[4]. Damit werden weniger versteckte Vorgänge im Hintergrund ausgeführt, was beim Analysieren von Fehlern die Suche vereinfachen kann.

5.2. Spielernamenverwaltung

Da die Übertragung des Client-Namens jeweils nur bei einem TAKE stattfindet, wird bei diesem Befehl jeweils zuerst geprüft, ob der Name bereits in der Spielernamenliste vorhanden ist. Falls nicht, wird dieser entsprechend hinzugefügt (Zeile 6 – 7, Abbildung 14).

Die entscheidenden Verarbeitungen auf dem Shared Memory der Spielernamenliste für die „addPlayer“ und „existPlayer“ Funktionen sind in den Abbildung 15 und Abbildung 16 dargestellt.

[4] Siehe Literaturverzeichniseintrag L3.

```
1   int i;
2   _Bool result = FALSE;
3   ..
4   sem_wait(semPlayerList);
5   sem_wait(semStatusVariables);
6   for (i = 0; i < shmStatusVariables->sv_numberOfPlayerNames; i++) {
7       if (strcmp(playername, &shmPlayerList[i*MAX_PLAYER_NAME_LENGTH]) == 0) {
8           result = TRUE;
9           break;
10      }
11  }
12  sem_post(semStatusVariables);
13  sem_post(semPlayerList);
14  ..
15  return result;
```

Abbildung 15: Existenzprüfung Spielername im Shared Memory

```
1   sem_wait(semPlayerList);;
2   sem_wait(semStatusVariables);
3   strcpy(&shmPlayerList[shmStatusVariables-
4   >sv_numberOfPlayerNames*MAX_PLAYER_NAME_LENGTH], playername);
5   shmStatusVariables->sv_numberOfPlayerNames++;
6   sem_post(semStatusVariables);
7   sem_post(semPlayerList);
```

Abbildung 16: Hinzufügen eines Spielnamens im Shared Memory

Verglichen werden die Einträge anhand des Namens (über die *strcmp*-Funktion), weshalb in den Annahmen festgehalten wurde, dass nicht mehrere Clients den gleichen Namen verwenden dürfen. Eine Identifikation anhand des File Descriptors der Verbindung findet nicht statt, wäre aber eine mögliche Erweiterung um Eindeutigkeit trotz gleicher Spielernamen zu erzwingen.

Durch die Definition der Spielernamenliste entsprechend Abschnitt 4.4.2 gestaltet sich die Identifikation der Spieler-ID anhand des Spielernamens ähnlich einfach:

```
1   sem_wait(semPlayerList);
2   sem_wait(semStatusVariables);
3   for(i = 0; i< shmStatusVariables->sv_numberOfPlayerNames; i++) {
4       if (strcmp(playername, &shmPlayerList[i*MAX_PLAYER_NAME_LENGTH]) == 0) {
5           id = i;
6           break;
7       }
8   }
9   sem_post(semStatusVariables);
10  sem_post(semPlayerList);
```

Abbildung 17: Finden der Spieler-ID anhand des Spielernamens

5.3. Ressourcenverwaltung und Abbruchbehandlung

Nach den ersten Tests mit eingebautem Shared Memory war bald offensichtlich, dass jeweils beim Erzeugen des Shared Memory-Bereichs wie auch beim Anlegen der Semaphore diese auch wieder aufgeräumt werden müssen.

Damit dies nicht nur beim korrekten Durchlauf des Programms stattfindet, wurde ein Signalhandler eingebaut, um speziell einen Programmabbruch mittels CTRL-C abzufangen. Wenn nun CTRL-C im laufenden Betrieb gedrückt wird, wird das Verlassen der *while*-Schlaufe in der *main*-Funktion erzwungen (unabhängig vom Game Level „sv_gameLevel“, welcher durch den „Gameplay“-Prozess kontrolliert ist), indem „keepRunning“ auf 0 gesetzt wird (Zeile 5, Abbildung 18). Dadurch wird erreicht, dass der Prozess ein korrektes Deallozieren und Freigeben aller angelegten Ressourcen durchführt.

```
1   void  intHandler(int sig) {
2       fprintf(stdout,"\n");
3       fprintf(stdout,"=== Ctrl-C was hit ... cleaning up ===\n");
4       fprintf(stdout,"\n");
5       keepRunning = 0;
6   }
7   ..
8
9   main(…) {
10      ..
11      struct sigaction sigIntHandler;
12      ..
13      sigIntHandler.sa_handler = intHandler;
14      sigemptyset(&sigIntHandler.sa_mask);
15      sigIntHandler.sa_flags = 0;
16      ..
17      sigaction(SIGINT, &sigIntHandler, NULL);
18      ..
19      while ((shmStatusVariables->sv_gameLevel >= 1) && (keepRunning == 1)) {
20          ..
21      }
22      // cleanups und shmStatusVariables->sv_gameLevel = 0
23      ..
24  }
```

Abbildung 18: Signalhandler

Zur Abbruchbehandlung wird zuerst wird ein Struct vom Typ *sigaction* erstellt (Zeile 11, Abbildung 18), danach wird die auszuführende Aktion zugewiesen (Zeile 13, Abbildung 18) und das weitere Verhalten des Signals definiert (Zeile 15, Abbildung 18). Über den Befehl *sigaction* wird unser Signalhandling dem Interrupt Signal (SIGINT; Zeile 17, Abbildung 18) zugeordnet, welches über CTRL-C[5] jeweils ausgelöst wird.

Pro Prozesstyp gibt es eine Cleanup-Funktion, welche jeweils vor dem Ende des Prozesses ausgeführt wird.

Das Ende im „Gameplay“-Prozess wird entweder erreicht in dem die prozesslokale „keepRunning“-Variable durch den Interrupt Handler auf 0 gesetzt wird oder das Spiel regelkonform durch einen festgestellten Sieger durch entsprechendes Setzen des Game Levels auf 3 beendet wird. Durch die Unterscheidung von Hauptprozessen („Gameplay“ und „Sockethandler“) und Unterprozessen („Clientinteraction“) und dem Verwenden einer lokalen Variable um den eigenen Prozess stoppen zu können, kann verhindert werden, dass bei Problemen mit einem Unterprozess gleich auch der Hauptprozess gestoppt wird. Die Hauptprozesse setzen dann am Ende ihrer Laufzeit die geteilte Variable „sv_gameLevel“ auf 0 (bei einem Abbruch mittels CTRL-C) oder auf 3 (falls ein Sieger gefunden wurde) und benachrichtigen entsprechend die anderen Server-Prozesse (alle „Childinteraction“-Prozesse), so dass diese ebenfalls die *while*-Schlaufe verlassen und sich aufräumen, bevor sie terminieren. Der „Gameplay“-Prozess wartet schlussendlich auf die Beendigung aller anderen Server-Prozesse mittels *wait* und räumt danach sich selbst (Semaphore (Zeilen 5 - 16, Abbildung 19) und das Shared Memory (19 - 31, Abbildung 19) auf, bevor er ebenfalls terminiert.

[5] Siehe Literaturverzeichniseintrag L4.

```
1   int i;
2   char shmmnPlayfieldElementName[256];
3   ..
4   // close semaphores
5   sem_close(semStatusVariables);
6   sem_unlink(SHMSN_STATUS_VARIABLES);
7   sem_close(semPlayerList);
8   sem_unlink(SHMSN_PLAYER_LIST);
9   ..
10  for (i = 0; i < FIELDSIZE*FIELDSIZE; i++) {
11      sem_close(semPlayfieldElement[i]);
12      sprintf(shmmnPlayfieldElementName, "%s%d",
13      SHMSN_PLAYFIELD_ELEMENT_PREFIX, i);
14      sem_unlink(shmmnPlayfieldElementName);
15  }
16  free(semPlayfieldElement);
17  ..
18  // cleanup shared memeory
19  shmdt(shmPlayerList);
20  shmPlayerList = NULL;
21  shmctl(shmPlayerListID, IPC_RMID, NULL);
22  ..
23  // cleanup playfield in shared memory
24  shmdt(shmPlayfield);
25  shmPlayfield = NULL;
26  shmctl(shmPlayfieldID, IPC_RMID, NULL);
27  ..
28  // cleanup shared variables in shared memory
29  shmdt(shmStatusVariables);
30  shmStatusVariables = NULL;
31  shmctl(sharedMemIDStruct, IPC_RMID, NULL);
```

Abbildung 19: „cleanUpGameplay“-Funktion der „Gameplay“-Prozess

Die Cleanup-Funktionen der anderen Prozesse räumen jeweils ihre Shared Memory-Zugriffe auf und schliessen ihre verantwortlichen Sockets.

5.4. Logger

Um das Programm einfacher debuggen zu können wurde ein existierender Logger[6] eingebunden, welcher die Ausgaben sowohl in die Konsole als auch in eine Datei ausgeben kann. Dieser ermöglicht das Setzen verschiedener Fehlerstufen und Kategorien, anhand welcher die Ausgabe gesteuert werden kann. Somit kann für jede Fehlerstufe und Kategorie festgelegt werden, ob die Ausgabe auf der Konsole oder in eine automatisch erzeugte Log-Datei erfolgen soll. Zudem werden die Logger-Fehlerstufen SLL_DEBUG, SLL_WARNING und SLL_ERROR im Falle der Konsolenausgabe nach STDERR umgeleitet, die restlichen auf STDOUT.

```
1   char gameplayLogTag[256];
2   sprintf(gameplayLogTag, "GAMEPLAY:%d", getpid());
3   if(strcmp(loglevel,"RELEASE") == 0) {
4       startup_logger(gameplayLogTag, SLO_CONSOLE | SLO_FILE, SLL_ERROR
5       |SLL_INFO, SLL_ALL_LEVELS | SLC_ALL_CATEGORIES);
6   }
7   else {
8       startup_logger(gameplayLogTag, SLO_CONSOLE | SLO_FILE, SLL_DEBUG|
9       SLL_ERROR |SLL_INFO| SLC_GAMEPLAY, SLL_ALL_LEVELS | SLC_ALL_CATEGORIES);
10  }
```

Abbildung 20: Starten des Loggers mit richtigen Parametern

[6] Siehe Literaturverzeichniseintrag L8.

Der Logger wurde so integriert und den Bedürfnissen angepasst, dass pro erstelltem Prozess eine Log-Datei erzeugt wird und in dieser alle Meldungen festgehalten werden. Zur besseren Identifikation der erzeugten Log-Dateien wird zudem noch die Prozess-ID (des Server-Forks) dem Log-Dateinamen angehängt.

Auf dem *STDOUT* erfolgen nur die für das Spiel relevanten Ausgaben, die Fehlermeldungen wiederum werden in *STDERR* ausgegeben. Der Log Level des Servers wird über einen optionalen Startparameter definiert (Default: „RELEASE“). Die Debug-Meldungen werden im „RELEASE“ Log Level-Modus des Servers in die pro Server-Prozess (pro fork) erstellte Log-Datei ausgegeben, im „DEBUG“ Log Level-Modus hingegen erfolgt die Ausgabe ebenfalls auch auf *STDERR*.

```
1
2   if ((g_output_flags & SLO_CONSOLE) && (g_console_mask_flags & mask_flag)) {
3       if ((mask_flag & SLL_DEBUG) || (mask_flag & SLL_ERROR) || (mask_flag &
4       SLL_WARNING))
5           fprintf(stderr, "%s", output_buffer);
6       else
7           fprintf(stdout, "%s", output_buffer);
8   }
```

Abbildung 21: Filterung der Ausgabe nach STDERR oder STDOUT in SimpleLogger.c

Der Logger erkennt anhand Bit-Maskierung durch bitweises AND, welche Kategorien gesetzt wurden und welche Fehlerstufen für eine Log-Ausgabe definiert wurden.

6. Herausforderungen & Lösungen

6.1. Kommunikationssynchronisation bei Spielaufbau

Da der Spielaufbau durch den Client gestartet wird, vom Server durch Angabe der Spielfeldgrösse anschliessend die Bestätigung erfolgt, der Client danach aber zuerst auf das START-Signal des Servers warten muss, bevor er zu spielen beginnt, ist eine Synchronisation zwischen dem Server und allen Clients notwendig.

Um die den Clients zugeordneten „Childinteraction"-Prozesse in den korrekten Zustand zu versetzen, wird nach einem empfangenen HELLO die „doHELLO" Funktion (Zeile 3, Abbildung 22) ausgeführt, die Antwort enkodiert und an den Client gesendet (Zeile 10, Abbildung 22). Gleich danach wird die „doSTART"-Funktion (Zeile 14, Abbildung 22) aufgerufen. Diese startet nicht gleich das „Spielen", sondern versetzt den „Childinteraction"-Prozess vorerst in eine Warteschlaufe, in welcher auf den offiziellen Spielstart gewartet wird (siehe Abbildung 23).

```
1   switch (currentAction.cmd) {
2       case HELLO:
3           doHELLO(&returnAction);
4           // manually trigger sending the necessary SIZE command
5           ..
6           // change struct to string
7           encode(&returnAction, replyMessage);
8           ..
9           // write response to socket
10          length = write(newsockfd, replyMessage, 18);
11          if (length < 0) fprintf(stderr, "[PID:%d] ERROR writing to socket",
12              getpid());
13          ..
14          doSTART(shmStatusVariables, &returnAction);
15          break;
```

Abbildung 22: Protokoll Handling bei empfangenen HELLO

```
1   while(TRUE) {
2       sem_wait(semStatusVariables);
3       if (shmStatusVariables->sv_gameLevel != 2) {
4           ..
5       }
6       else {
7           // start the game
8           returnAction->cmd = START;
9           sem_post(semStatusVariables);
10          break;
11      }
12      sem_post(semStatusVariables);
13  }
```

Abbildung 23: doSTART-Prüfung auf Spielanfang

Da das Startsignal von einem anderen Prozess (dem „Gameplay"-Prozess) ausgelöst wird, geschieht dies durch Abfrage der entsprechenden Shared Memory Variable (Zeile 3, Abbildung 23). Sobald der „sv_gameLevel" angepasst wurde, wird über *break* die *while*-Schlaufe der „doSTART"-Funktion verlassen und der „Childinteraction"-Prozess kehrt in die *while*-Schlaufe der *main*-Funktion zurück um dort die Interpretation und Verarbeitung der empfangenen Protokollkommandos wieder aufzunehmen (und in diesem Fall zu „Spielen").

6.2. Blockierung durch accept und read

Um zu verhindern dass der „Sockethandler“-Prozess blockiert, solange er auf neue Verbindungen mit *accept* wartet, was vor allem für die korrekte Beendigung des Prozesses im Falle eines Sieges oder Abbruchs wichtig ist, wird der Stand des Sockets vorher mit *select*[7] geprüft. Ansonsten würde der Prozess Änderungen am Spielablauf gar nicht mehr wahrnehmen, sollten sich nicht dauernd neue Clients anmelden.

```
1   fd_set rfds;
2   struct timeval tv;
3   int retval;
4   ..
5   // watch stdin (fd 0) to see when it has input
6   FD_ZERO(&rfds);
7   FD_SET(sockfd, &rfds);
8   ..
9   // wait up to five seconds
10  tv.tv_sec = 5;
11  tv.tv_usec = 0;
12  ..
13  retval = select(sockfd+1, &rfds, NULL, NULL, &tv);
14  ..
15  if (retval > 0) {
16      ..
17      newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen);
18      if (newsockfd < 0) {
```

Abbildung 24: Prüfen des Sockets vor dem Auslesen

Entsprechend wird zuerst ein Timeout definiert, während welchem der Zustand geprüft werden soll (Zeile 10, Abbildung 24). Wenn innerhalb dieses Zeitraums Daten auf dem Socket ankommen sind („retval“ > 0; Zeile 13, Abbildung 24) kann ein komplettes Auslesen gestartet werden (Zeile 17, Abbildung 24). Falls nicht dreht die Schlaufe weiter und startet die nächste Iteration, wobei bei dieser wiederum zuerst geprüft werden kann ob das Spiel weiterhin noch laufen soll.

Denselben Fall gibt es auch in den „Childinteraction“-Prozessen im Falle mit *read* auf dem Socket zu berücksichtigen (durch denselben Lösungsansatz), da ansonsten im Falle eines zwar angemeldeten, aber nicht-spielenden Clients der entsprechende „Childinteraction“-Prozess dauern auf ein weiteres Protokollkommando warten würde, ohne regelmässig den Spielablauf prüfen zu können.

6.3. Verwaltung von Shared Memory

Die Arbeit mit Shared Memory war speziell zu Beginn der Implementation kompliziert, als das jeweilige Aufräumen noch nicht realisiert war.

Da der Zugriff zu einem gemeinsam genutzten Speicherbereich parallel in allen Prozessen stattfindet, wurde teilweise beim nächsten Programmstart wieder auf dieselben Speicherbereiche zugegriffen, weshalb lange nicht eindeutig war, wann und wo und vor allem weshalb ein Fehler auftrat, da die nicht aufgeräumten Shared Memory-Bereich Fehler verschleierten oder gar erst erzeugten. Die Analyse wurde dabei nicht selten von Core Dumps begleitet.

Der Zugriff auf die Shared Memory-Bereiche findet über den zu Beginn definierten Schlüsselwert statt, welcher bei einem fehlerhaften Abbruch allerdings nicht korrekt aufgeräumt wurde. Nach temporärem Zuordnen eines neuen Schlüsselwertes konnte zwar ein neuer Versuch gestartet werden bis der effektive Fehler lokalisiert war. Doch war dies nicht sehr elegant. Alternativ hätte ein Reboot zwischen den Versuchen

[7] Siehe Literaturverzeichniseintrag L6.

auch geholfen. Dies wäre aber sehr ineffizient gewesen. Zudem hätte die Identifikation des Shared Memory-Bereiches im System mittels des entsprechenden Werts über das Kommando *ipcs* und der Anwendung von *ipcrm*-Kommandos zum Löschen des Eintrags[8] geholfen, solche Problemfälle effizienter zu meistern. Dabei müsste nur der jeweilige hexadezimale *Key*-Wert der *ipcs*-Ausgabe in Dezimal umgerechnet werden, um eine entsprechende Zuordnung finden zu können.

```
1   // shared memory keys
2   #define SHMK_STATUS_VARIABLES 1000
3   #define SHMK_PLAYER_LIST 2000
4   #define SHMK_PLAYFIELD 3000
```

Abbildung 25: Shared Memory Key-Werte

Die umgekehrte Umrechnung zeigt dass 1000 dem hexadezimalen Wert 3e8, 2000 dem hexadezimalen Wert 7d0 und 3000 dem hexadezimalen Wert bb8 entspricht. Abbildung 26 zeigt die entsprechenden im System angelegten Shared Memory-Bereiche.

```
[des@tyra ~]$ ipcs

------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x00000000 65536      des        600        393216     2          dest
0x00000000 98305      des        600        393216     2          dest
0x00000000 131074     des        600        393216     2          dest
0x00000000 229379     des        600        393216     2          dest
0x00000000 393220     des        600        393216     2          dest
0x00000000 688133     des        600        393216     2          dest
0x00000000 884742     des        700        2663808    2          dest
0x00000000 917511     des        600        393216     2          dest
0x00000000 1245192    des        700        328440     2          dest
0x00000000 1114121    des        600        393216     2          dest
0x00000000 1212426    des        700        8304648    2          dest
0x00000000 1277963    des        700        42880      2          dest
0x00000000 1310732    des        600        393216     2          dest
0x00000000 7438349    des        600        393216     2          dest
0x00000000 3735566    des        600        393216     2          dest
0x00000000 7405583    des        600        72072      2          dest
0x00000000 7536656    des        600        16384      2          dest
0x00000000 7569425    des        600        393216     2          dest
0x000003e8 7864338    des        666        16         2
0x00000000 7634963    des        600        4194304    2          dest
0x000007d0 7897108    des        666        62208      3
0x00000bb8 7929877    des        666        100        2

------ Semaphore Arrays --------
key        semid      owner      perms      nsems
0x00000000 32768      http       600        1

[des@tyra ~]$
```

Abbildung 26: Ausgabe von ipcs zur Anzeige von Shared Memory-Bereichen

Im späteren Verlauf des Projekts wurden dann die Shared Memory-Bereiche korrekt durch die bereits erwähnte „cleanUpGameplay“-Funktion über *shmdt* und *shmctl* aufgeräumt, was das weitere Fehlerauftreten und die allfällige Fehlersuche positiv beeinflusste.

[8] Siehe Literaturverzeichniseintrag L7.

6.4. Verwaltung von Semaphoren

Die Verwaltung der Semaphore verursachte zu Beginn eine ähnliche Problematik wie jener der Shared Memory-Bereiche (siehe Abschnitt 6.3).

Ganz nach dem Motto „in Linux ist alles eine Datei“ konnte aber bald festgestellt werden, dass im Verzeichnis „/dev/shm/“ für jedes Semaphore eine Datei angelegt wurde, welche im schlimmsten Fall von Hand gelöscht werden konnten, solange die Semaphore noch nicht durch die Cleanup-Funktion sauber freigegeben wurden.

```
[des@tyra ~]$ ls -al /dev/shm/
total 180
drwxrwxrwt  2 root root       620 Jun 14 14:27 .
drwxr-xr-x 22 root root      3540 Jun 14 12:02 ..
-rwx------  1 des  users 67108904 Jun 14 12:03 pulse-shm-245838462
-rwx------  1 des  users 67108904 Jun 14 12:03 pulse-shm-4168729427
-rw-r--r--  1 des  users       32 Jun 14 14:27 sem.PLAYERLIST
-rw-r--r--  1 des  users       32 Jun 14 14:27 sem.PLAYFIELDELEMENT0
-rw-r--r--  1 des  users       32 Jun 14 14:27 sem.PLAYFIELDELEMENT1
-rw-r--r--  1 des  users       32 Jun 14 14:27 sem.PLAYFIELDELEMENT10
-rw-r--r--  1 des  users       32 Jun 14 14:27 sem.PLAYFIELDELEMENT11
-rw-r--r--  1 des  users       32 Jun 14 14:27 sem.PLAYFIELDELEMENT12
-rw-r--r--  1 des  users       32 Jun 14 14:27 sem.PLAYFIELDELEMENT13
-rw-r--r--  1 des  users       32 Jun 14 14:27 sem.PLAYFIELDELEMENT14
-rw-r--r--  1 des  users       32 Jun 14 14:27 sem.PLAYFIELDELEMENT15
-rw-r--r--  1 des  users       32 Jun 14 14:27 sem.PLAYFIELDELEMENT16
-rw-r--r--  1 des  users       32 Jun 14 14:27 sem.PLAYFIELDELEMENT17
-rw-r--r--  1 des  users       32 Jun 14 14:27 sem.PLAYFIELDELEMENT18
-rw-r--r--  1 des  users       32 Jun 14 14:27 sem.PLAYFIELDELEMENT19
-rw-r--r--  1 des  users       32 Jun 14 14:27 sem.PLAYFIELDELEMENT2
-rw-r--r--  1 des  users       32 Jun 14 14:27 sem.PLAYFIELDELEMENT20
-rw-r--r--  1 des  users       32 Jun 14 14:27 sem.PLAYFIELDELEMENT21
-rw-r--r--  1 des  users       32 Jun 14 14:27 sem.PLAYFIELDELEMENT22
-rw-r--r--  1 des  users       32 Jun 14 14:27 sem.PLAYFIELDELEMENT23
-rw-r--r--  1 des  users       32 Jun 14 14:27 sem.PLAYFIELDELEMENT24
-rw-r--r--  1 des  users       32 Jun 14 14:27 sem.PLAYFIELDELEMENT3
-rw-r--r--  1 des  users       32 Jun 14 14:27 sem.PLAYFIELDELEMENT4
-rw-r--r--  1 des  users       32 Jun 14 14:27 sem.PLAYFIELDELEMENT5
-rw-r--r--  1 des  users       32 Jun 14 14:27 sem.PLAYFIELDELEMENT6
-rw-r--r--  1 des  users       32 Jun 14 14:27 sem.PLAYFIELDELEMENT7
-rw-r--r--  1 des  users       32 Jun 14 14:27 sem.PLAYFIELDELEMENT8
-rw-r--r--  1 des  users       32 Jun 14 14:27 sem.PLAYFIELDELEMENT9
-rw-r--r--  1 des  users       32 Jun 14 14:27 sem.STATUS
[des@tyra ~]$ more /dev/shm/
```

Abbildung 27: Semaphore im /dev/shm Verzeichnis

Die Semaphore wurden schlussendlich im späteren Verlauf der Entwicklung in der „cleanUpGameplay“-Funktion mit *sem_close* und *sem_unlink* aufgeräumt.

7. Fazit

7.1. Reflexion

Durch die berufliche Tätigkeit als System Engineer im Netzwerk und Sicherheitsbereich war zwar das Verständnis für die Aufgabenstellung gegeben, das Umsetzen mittels eigenem Code musste allerdings sehr systematisch angegangen werden. Mangelnd beruflicher Routine in der Entwicklung von Software-Anwendungen musste ich zuerst auf Papier ein Grundkonzept des Aufbaus aufzeichnen, um daraus danach eine mögliche Lösungsstrategie abzuleiten. Durch viele Diskussionen mit Personen aus meinem privaten Umfeld wurden Ideen besprochen, Vor- und Nachteile abgewogen und damit grundlegenden Fehler vorgebeugt. Mit einem aufbauenden Umsetzungsphasenplan, welcher jeder Projektphase eine konkrete Projektherausforderungen zuordnete (Definition grundlegendes Systemdesign, Implementierung Client/Server-Skelet, Protokollimplementierung, Umsetzung Client-Strategien, Realisierung Einprozess-Server, Server-Spiel-Loop Implementierung, Transformation Mehrprozess-Server, Integration Shared Memory, Absicherung durch Semaphore, Finalisierung) konnte dann die Umsetzung erfolgen.

Wenn dann später Denkfehler im Code entdeckt wurden, waren diese meist durch mangelnde strikte Umsetzung des Grundkonzepts verursacht worden, zum Beispiel bei der Verarbeitung von Aufgaben in Prozesstypen, für welche diese gar keine Verantwortung hatten. Ein weiteres Beispiel ist die übertriebene Verschiebung von Variablen in den Shared Memory-Bereich, was unnötige Semaphore Locks notwendig machte, wie beispielsweise beim anfänglich Einsatz der „FIELDSIZE“ Variable für die Spielfeldseitenlänge: Diese war zuerst im Shared Memory-Bereich angelegt, stellte sich aber nach der Integration der Semaphore als unnötig heraus, da dieser Wert während des Spiels nicht verändert wird.

Eine Lessons Learned ist, dass man von Anfang an mit der Implementierung der Initialisierung von irgendwelchen Objekten (wie zum Beispiel Shared Memory-Bereiche oder Semaphore), für diese gleich auch den Cleanup umsetzen sollte. Diese hätte einige Korrekturiterationen verhindern oder zumindest reduzieren können (siehe Abschnitte 6.3 und 6.4).

Die zu Beginn erstellte Ressourcenplanung war oft zu pessimistisch geschätzt (zumindest für die ersten Projektphasen), wurde am Schluss aber trotzdem fast ausgeschöpft, da das Analysieren der Fehler rund um Shared Memory und Semaphore einiges an zusätzlicher Zeit benötigte. Dank des regelmässigen Vergleichs zwischen Projektplan, Schätzung und Projektfortschritt und der Berücksichtigung der der bisherigen Erfahrungen konnte kontinuierlich auf die verbleibende Planung Einflussgenommen werden um schlussendlich das Projekt dann doch termingerecht fertigzustellen. Gesamthaft wurden inklusive der Dokumentation 80 Stunden aufgewendet.

7.2. Ausblick

Grundsätzlich bin ich mit dem Server-Realisierung sehr zufrieden. Falls ich ihn weiterentwickeln würde, gäbe es meiner Meinung nach noch folgende offene Punkte:

- Die Identifikation der Spielernamen könnte anhand des File Descriptors und nicht anhand des übertragenen Namens geschehen (wobei mit der aktuellen Lösung ein Verbindungsabbau und Wiederaufbau möglich ist). Diese würde die Restriktion der eindeutigen Spielernamen aufheben.
- Im Zusammenhang mit der Client-Strategie, welche sich sehr lange nicht meldet (Strategie 5) bleibt zurzeit die Socket-Verbindung des Servers sehr lange geöffnet. Dies könnte noch optimiert werden, damit der Abbau sich analog der anderen Verbindungen verhält.

- Die Haupt-*while*-Schlaufe ist zurzeit etwas unübersichtlich und könnte in einzelne Funktionen pro „geforktem“ Prozess mit eigener *while*-Schlaufe unterteilt und ausgelagert werden. Dies würde die Leserbarkeit verbessern.

7.3. Schlusswort

Die Arbeit war eine spannende Herausforderung, weil es mir die Möglichkeit gab, besser im Programmieren von C zu werden und aktiv Lösungen zu Problematiken von Memory-Allozierung und -Zugriffen zu erlernen. Da das Kommunikationsverhalten zudem ähnlich einem Command & Control[9] Server zu seinen Botnet Clients[10] ist, war es zudem interessant zu sehen, wie wenig aufwändig ein solches Programm sein kann.

Ich hatte mich für das Seminar angemeldet um mehr Übung im Lesen und Schreiben von C Code zu erhalten, da mich vor allem die Analyse von C-Programmen aus Sicht meiner beruflichen Tätigkeit der IT-Forensik speziell interessiert. Durch die intensive Beschäftigung mit dem Thema sollte es mir nun möglich sein, weitere berufliche und private Projekte in diesem Bereich verfolgen zu können.

[9] Siehe Glossarverzeichniseintrag G1.

[10] Siehe Glossarverzeichniseintrag G2.

8. Anhang

8.1. Literaturverzeichnis

Jegliche Informationen wurden auf Basis der folgenden Dokumente zusammengetragen:

Tabelle 2: Literaturverzeichnis

Nr.	Verweis
L1	Github des Lehrers mit Aufgabenstellung im README.md File: https://github.com/telmich/zhaw_concurrent_c_programming_fs_2015
L2	Basis Client-Server-Socket Tutorial: http://www.cs.rpi.edu/~moorthy/Courses/os98/Pgms/socket.html
L3	Hinweis zu C Handhabung zweidimensionale Arrays: http://stackoverflow.com/questions/1242705/performance-of-2-dimensional-array-vs-1-dimensional-array
L4	Hinweis zu Abfangen von CTRL-C: http://stackoverflow.com/questions/1641182/how-can-i-catch-a-ctrl-c-event-c
L5	Eigenes Github-Verzeichnis: https://github.com/d3sre/concurrentc
L6	Hinweis zu Non-blocking socket: http://stackoverflow.com/questions/3360797/non-blocking-socket-with-poll
L7	Hinweis zum Troubleshooting von Shared Memory-Segmenten: http://beej.us/guide/bgipc/output/html/multipage/shm.html
L8	Verwendeter Logger: https://bitbucket.org/pdpatrickdiezi/simpleloggerlinux

8.2. Abbildungsverzeichnis

8.3. Tabellenverzeichnis

8.4. Glossar

Hier werden verwendete Fachbegriffe erläutert.

Tabelle 3: Glossar

ID	Begriff	Beschreibung
G1	Command & Control Server	Im Sicherheitsumfeld werden so die zentralen Kommandoserver bezeichnet, welche die Botnet Clients steuern
G2	Botnet Clients	Im Sicherheitsumfeld sind dies Computer, welche über Malware infiziert wurden und unter anderem für Spam Mails oder DDoS-Attacken fremdgesteuert werden