# Seminar: Concurrent C Programming

Dozent: Nico Schottelius
FS 2015

ZHAW SoE Zürich

Marco Buess
Version 1.0
21. Juni 2015

# Inhaltsverzeichnis

**Inhaltsverzeichnis** ..... **2**

**Einleitung** ..... **3**

**Anleitung zur Nutzung** ..... **6**
Vorbereitungen ..... 6
Aufruf Server ..... 6
Clientaufruf ..... 8

**Weg, Probleme, Lösungen** ..... **10**

**Fazit** ..... **11**

**Anhang** ..... **12**
Quellenverzeichnis ..... 12
Quelltexte ..... 12
*Makefile* ..... *12*
*Server.c* ..... *12*
*Client.c* ..... *19*

## Einleitung

Die Aufgabe war eine Client Server Applikation zu schreiben. Der Server erstellt ein quadratisches Spielfeld mit der Seitenlänge n, mindestens jedoch vier. Danach wartet er bis sich mindestens n/2 Client an den Server Verbinden und startet das Spiel.
Der Client versucht einzelne Felder für sich zu reservieren. Der Server überprüft in zufälligen Abständen ob ein Client alle Felder reserviert hat und beendet das Spiel, sobald dieses Szenario eingetreten ist. Die detaillierte Aufgabenstellung ist weiter unten beschrieben.

Der Autor dieser Arbeit bringt nicht sonderlich grosse Erfahrung im Bereich der Softwareentwicklung mit. Mit der Programmiersprache C im Besonderen hat der Autor noch keine grösseren Projekte realisiert.

## Aufgabenstellung

### Protokoll Allgemein

Befehle werden mit \n abgeschlossen
Kein Befehl ist länger als 256 Zeichen inklusive dem \n
Jeder Spieler kann nur 1 Kommando senden und muss auf die Antwort warten

#### *Anmeldung*

Erfolgreiche Anmeldung:

```
Client: HELLO\n
Server: SIZE n\n
```

Nicht erfolgreiche Anmeldung:

```
Client: HELLO\n
Server: NACK\n
    -> Trennt die Verbindung
```

#### *Spielstart*

Der Server wartet auf n/2 Verbindungen vor dem Start.

```
Server: START\n
Client: - (erwiedert nichts, weiss das es gestartet hat)
```

#### *Feld erobern erfolgreich*

Wenn kein anderer Client gerade einen TAKE Befehl für das selbe Feld sendet, kann ein Client es nehmen.

```
Client: TAKE X Y NAME\n
Server: TAKEN\n
```

#### *Feld erobern: nicht erfolgreich*

Wenn ein oder mehrere andere Clients gerade einen TAKE Befehl für das selbe Feld sendet, sind alle bis auf der erste nicht erfolgreich.

```
Client: TAKE X Y NAME\n
Server: INUSE\n
```

#### *Besitz anzeigen*

```
Client: STATUS X Y\n
Server: Name-des-Spielers\n
```

#### *Spielende*

Sobald ein Client alle Felder besitzt wird der Gewinner bekanntgegeben. Diese Antwort kann auf jeden Client Befehl kommen, mit Ausnahme der Anmeldung kommen.

```
Server: END Name-des-Spielers\n
```

```
Client: - (beendet sich)
```

## Bedingungen für die Implementation

Es gibt keinen globalen Lock (!)
Der Server speichert den Namen des Feldbesitzers
Kommunikation via TCP/IP
fork + shm (empfohlen)

- oder pthreads
- für jede Verbindung einen prozess/thread
- Hauptthread/prozess kann bind/listen/accept machen
- Rating Prozess/Thread zusätzlich im Server

Fokus liegt auf dem Serverteil

- Client ist hauptsächlich zum Testen und "Spass haben" da
- Server wird durch Skript vom Dozent getestet

Locking, gleichzeitiger Zugriff im Server lösen
Debug-Ausgaben von Client/Server auf stderr

## Tipps

```
gcc -Wall -Wpedantic -Wextra
Optional: valgrind
```

IDE-Möglichkeiten ** clion ** vim + gdb + gcc + make

# Anleitung zur Nutzung

Die vorliegende Software wurde auf einem xubuntu Image für Virtual Box getestet [1]

## Vorbereitungen

Die Kompilierung findet mittels make statt.

*make*

```
xubuntu@xubuntu-VirtualBox:~/Dropbox/ZHAW/C-Seminar/Linux$ make
gcc server.c -o server -std=gnu99
gcc client.c -o client
xubuntu@xubuntu-VirtualBox:~/Dropbox/ZHAW/C-Seminar/Linux$
```

## Aufruf Server

Der Server lässt sich durch den Aufruf ./server starten

Bsp:
*./server*

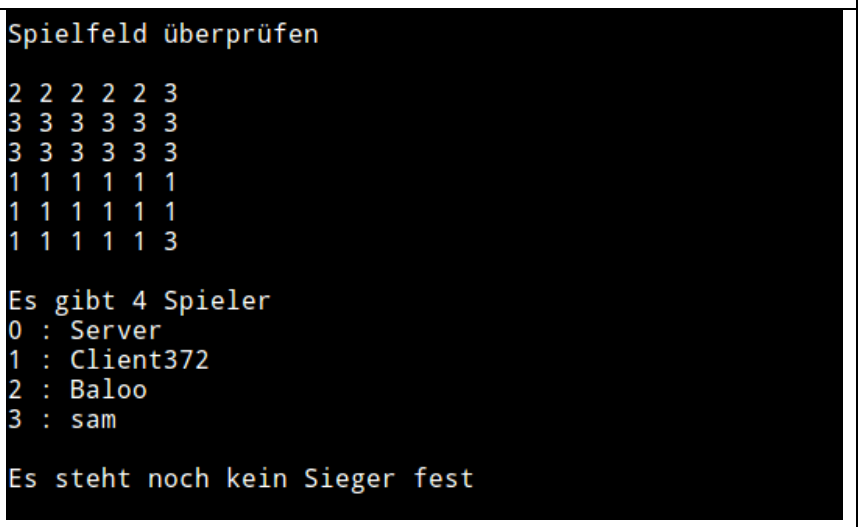
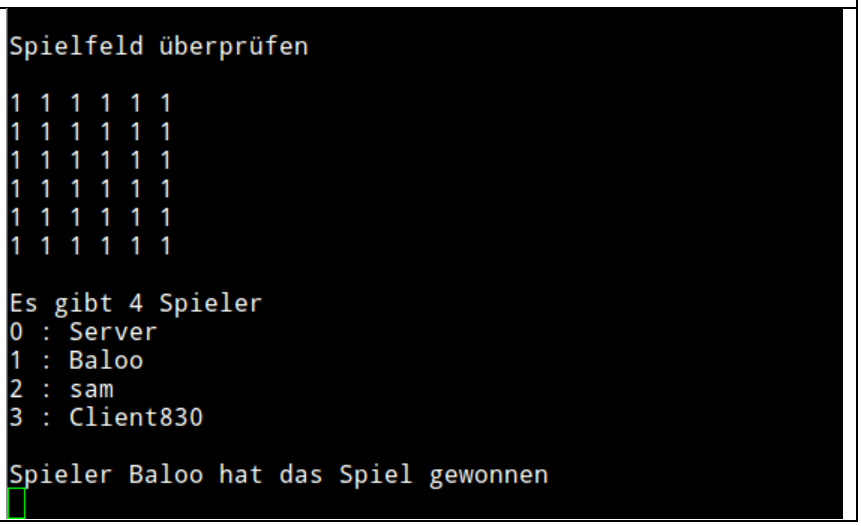
Die Defaultwerte sind folgendermassen festgelegt:
Port: 12345
Spielfeldgrösse: 4

```
xubuntu@xubuntu-VirtualBox:~/Dropbox/ZHAW/C-Seminar/Linux$ ./server

Serverinitialisierung:

Seitenlänge 4
Serverport 12345
```

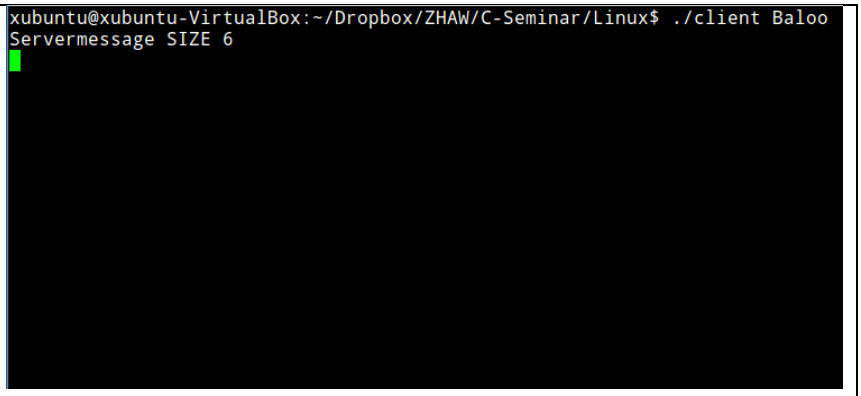
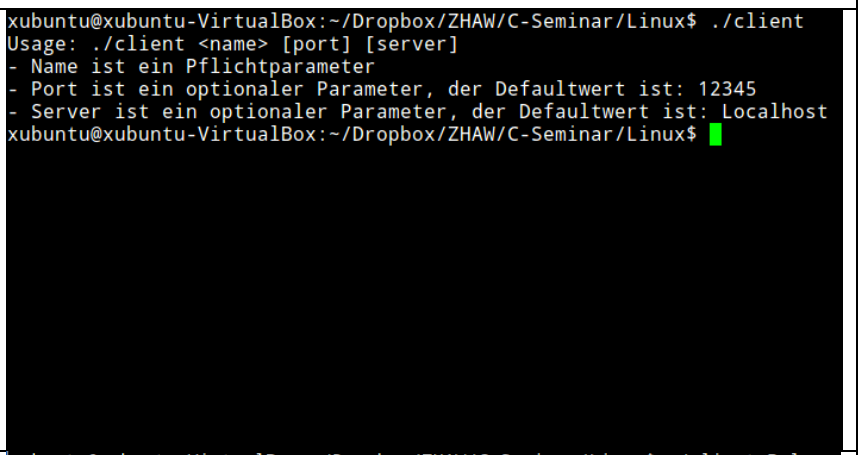
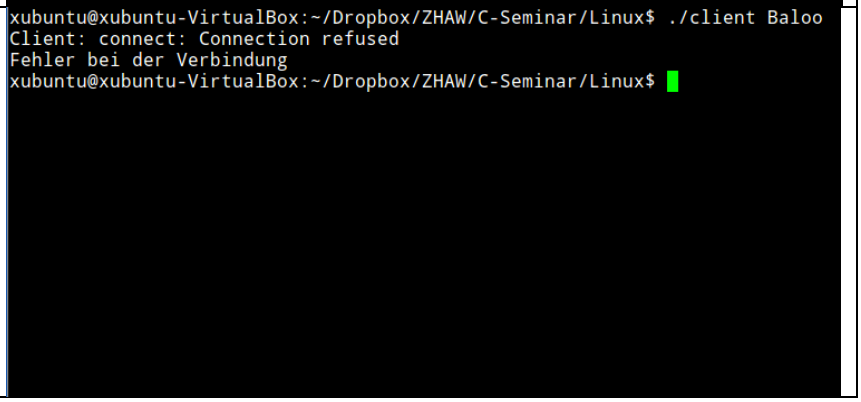
Alternativer Aufruf: Mittels ./server [portnummer] [spielfeldgrösse] lässt sich der Server mit anderen Werten als den Defaultwerten Starten

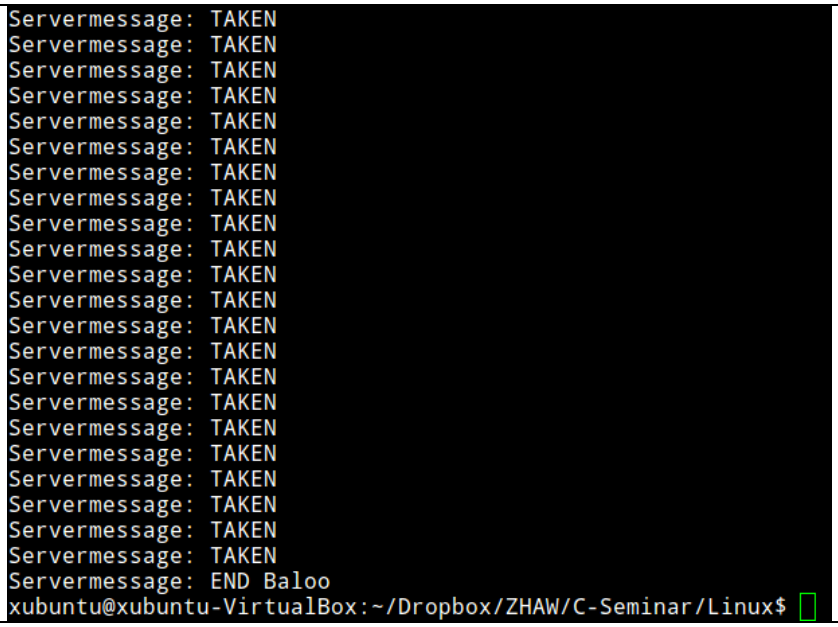
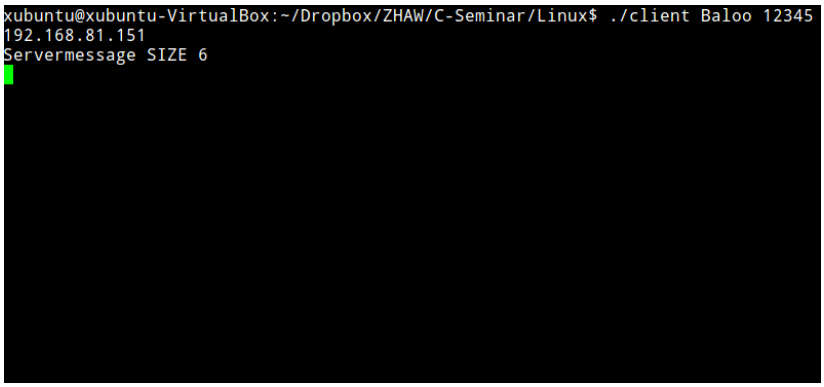
Bsp:
*./server 12345 6*

```
xubuntu@xubuntu-VirtualBox:~/Dropbox/ZHAW/C-Seminar/Linux$ ./server 12345 6

Serverinitialisierung:

Seitenlänge 6
Serverport 12345
```

| | |
|---|---|
| Der Server Wartet nun auf n/2 Verbindungen, wobei n die Spielfeldgrösse ist. | |
| Sobald das Spiel läuft, wird in zufälligen Abständen das Spielfeld überprüft.<br><br>Bei der Überprüfung wird das Spielfeld, und eine Liste der Spieler ausgegeben.<br><br>Die Überprüfung findet jeweils nach einer zufälligen Zeitdauer, welche zwischen 1 und 60 Sekunden dauern kann, statt. |  |
| Sobald ein Spieler das Spiel gewonnen hat, wird dies am Server ausgegeben.<br><br>Das Spiel gewinnt der Spieler, welcher bei einer Überprüfung sämtliche Felder besetzt hat. |  |

## Clientaufruf

| | |
|---|---|
| Der Client wird mittels dem Aufruf ./client <Name> gestartet.<br><br>Bsp:<br>*./client Baloo*<br><br>Die Defaultwerte für Port und Serveradresse lauten:<br>Port: 12345<br>Serveradresse: Localhost |  |
| Der Server übermittelt danach die Spielfeldgrösse für das Spiel. | |
| Im Gegensatz zum Server kann der Cient nicht Parameterlos gestartet werden. Bei einem entsprechenden Versuch, wird eine Meldung mit den Startparametern ausgegeben. |  |
| Wird versucht eine Verbindung aufzubauen, ohne dass der Server gestartet ist, wird ebenfalls eine entsprechende Meldung ausgegeben. |  |

| | |
|---|---|
| Während dem Spiel werden eine Reihe von Meldungen ausgegeben. Bei Spielende wird der Sieger an den Client übermittelt. |  |
| Wie schon beim Server lässt sich auch der Client mittels Angabe von weiteren Parametern starten. Mittels ./client <name> [portnummer] [serveradresse] kann man auch Portnummer und Serveradresse beim Aufruf übergeben.<br><br>Bsp:<br>*./client 12345 192.168.81.151* |  |

## Weg, Probleme, Lösungen

Wie bereits in der Einleitung erwähnt, bringt der Autor praktisch keine Programmiererfahrung mit. Somit war praktisch die gesamte Aufgabe ein riesiges Problem. Viele Recherchen und auch viel „Try and Error“ war notwendig, bis der Autor an diesem Punkt angekommen ist.

Der Autor möchte allerdings doch auf einen spezifischen Punkte eingehen:

```
connectioncount = mmap(NULL, sizeof(int), PROT_READ |
PROT_WRITE,MAP_SHARED | MAP_ANONYMOUS, -1, 0);
*connectioncount = 0;
startsignal = mmap(NULL, sizeof(int), PROT_READ |
PROT_WRITE,MAP_SHARED | MAP_ANONYMOUS, -1, 0);
*startsignal = 0;
endsignal = mmap(NULL, sizeof(int), PROT_READ | PROT_WRITE,MAP_SHARED
| MAP_ANONYMOUS, -1, 0);
*endsignal = 0;
```

Der Beispielcode macht Teile des Speichers für sämtliche fork() Prozesse gemeinsam Nutzbar, da ansonsten mit dem fork() Befehl auch das gesamte Memory kopiert wird, so dass jeder Prozess auf seinem eigenen Speicherbereich Arbeit.
Offenbar gibt es aber gewisse Einschränkungen bei der Benutzung, so hatte „endsignal“ während der Arbeit ständig einen sehr kleinen negativen Wert angenommen, anstelle der zugewiesenen Null. Hat man die Reihenfolge der Aufrufe geändert, wurde entsprechend wieder der letzten Variablen dieser kleine negative Wert zugewiesen.
Das Problem lies sich schlussendlich nur lösen, indem ein Array, welches ebenfalls im gemeinsamen Speicherbereich angelegt wurde, verkleinert wurde.

## Fazit

Die Aufgabe war sehr interessant, allerdings auch sehr Aufwendig. Zumindest wenn man ohne grosse Programmiererfahrung an die Aufgabe ran gegangen ist. Leider wurde gegen Ende die Zeit immer Knapper, so dass der Autor einige Funktionen nicht mehr wie gewünscht umsetzen konnte.
Die Aufgabe bietet einiges an Spielraum um sich zu verwirklichen und dem Autor fallen noch einige Möglichkeiten ein um das Programm zu verbessern. Alleine schon die Strategischen Möglichkeiten die man dem Client programmieren könnte währen sehr interessant.

# Anhang

## Quellenverzeichnis

[1] Xubuntu Virtual Box Image Download. [Online].
http://sourceforge.net/projects/virtualboximage/files/Xubuntu%20Linux/11.10/xubuntu1110.7z/download?use_mirror=garr

## Quelltexte

### Makefile

```
all:
	gcc server.c -o server -std=gnu99
	gcc client.c -o client
```

### Server.c

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <sys/mman.h>

const int MaxPlayer = 1000;
static int *playercount;
static int *startsignal;
static int *connectioncount;
static int *endsignal;
int **board;
char **playername;


void printboard(int boardsize)
{
    printf( "\n");
    for(int i=0;i<boardsize;i++)
    {
        for(int j=0;j<boardsize;j++)
        {
            printf( "%d ", board[i][j] );
        }
        printf( "\n");
    }
    printf( "\n");
}

int checkboard(int boardsize)
{
    int firstfieldvalue = board[0][0];
    for (int i=0;i<boardsize;i++)
    {
        for(int j=0;j<boardsize;j++)
```

```
        {
            if(firstfieldvalue != board[i][j])
            {
                return -1;
            }
        }
    }
    return firstfieldvalue;
}

int setfield(int xvalue, int yvalue, int player, int boardsize)
{
    if(xvalue >= boardsize || yvalue >= boardsize)
    {
        return -1;
    }
    else {
        board[yvalue][xvalue] = player;
        return 1;
    }
}

int getfield(int xvalue, int yvalue, int boardsize)
{
    if(xvalue >= boardsize || yvalue >= boardsize)
    {
        return -1;
    }
    else
    {
        return board[yvalue][xvalue];
    }
}

void boardgenerator(int boardsize)
{
    // initialisiere ein globales board
    board = mmap(NULL, sizeof(int) * boardsize, PROT_READ |
PROT_WRITE,MAP_SHARED | MAP_ANONYMOUS, -1, 0);
    if(board == NULL)
    {
        fprintf(stderr, "out of memory\n");
        exit(-1);
    }
    for(int i = 0; i < boardsize; i++)
    {
        board[i] = mmap(NULL, sizeof(int) * boardsize, PROT_READ |
PROT_WRITE,MAP_SHARED | MAP_ANONYMOUS, -1, 0);
        if(board[i] == NULL)
        {
            fprintf(stderr, "out of memory\n");
            // free(board);
            munmap(board, sizeof(int) * boardsize);
            munmap(*board, sizeof(int) * boardsize);
            exit(-1);
        }
    }
    // Inhalt des boardes wird mit Null initialisiert (Playername:
Server)
    for(int i=0;i<boardsize;i++)
    {
        for(int j=0;j<boardsize;j++)
        {
```

```
            board[i][j] = 0;
        }
    }
}

void printplayerlist()
{
    printf("Es gibt %d Spieler\n", *playercount+1);
    for (int i=0; i<=*playercount ; i++)
        printf("%d : %s \n", i, playername[i]);
    printf( "\n");
}

int setplayername(char buf[256])
{
    *playercount = *playercount+1;;
    if (*playercount > MaxPlayer)
    {
        printf("No more Player allowed \n");
        *playercount = *playercount-1;;
        return -1;
    }
    strcpy(playername[*playercount], buf);
    // printf("Playercount ist: %d\n", *playercount);
    return *playercount;
}

int getplayerID(char buf[256])
{
    for (int i=0;i<=*playercount;i++)
    {
        if (strcmp (playername[i], buf) == 0)
            return i;
    }
    return setplayername(buf);
}

void playerlistgenerator()
{
    // Initalisierung Spielerliste
    playername = mmap(NULL, MaxPlayer, PROT_READ |
PROT_WRITE,MAP_SHARED | MAP_ANONYMOUS, -1, 0);
    if(playername == NULL)
    {
        fprintf(stderr, "out of memory\n");
        exit(-1);
    }
    for(int i = 0; i <= MaxPlayer; i++)
    {
        playername[i] = mmap(NULL, sizeof(char) * 256, PROT_READ |
PROT_WRITE,MAP_SHARED | MAP_ANONYMOUS, -1, 0);
        if(playername[i] == NULL)
        {
            fprintf(stderr, "out of memory\n");
            munmap(playername, MaxPlayer * sizeof(char) * 256);
            exit(-1);
        }
    }

    // Initialisierung Spielerzähler
    playercount = mmap(NULL, sizeof(int), PROT_READ |
PROT_WRITE,MAP_SHARED | MAP_ANONYMOUS, -1, 0);
    *playercount = -1;
```

```
}

void clientmanager(int, int); /* function prototype */

void servermanager(int boardsize)
{
    while (*startsignal == 0)
    {
        if ((boardsize / 2) <= *connectioncount)
            {
                *startsignal = 1;
            }
        sleep(1);
    }

    // Check Winner all few Seconds
    while (1)
    {
        sleep(rand() % 5 + 1); // Set few Seconds 1 <= t <= 30
        printf("Spielfeld überprüfen\n");
        int i = checkboard(boardsize);
        printboard(boardsize);
        printplayerlist();
        if (i > 0)
        {
            printf("Spieler %s hat das Spiel gewonnen\n",
playername[i]);
            *endsignal = i;
            break;
        }
        else
        {
            printf("Es steht noch kein Sieger fest\n\n");
        }
    }
}

void error(const char *msg)
{
    perror(msg);
    exit(1);
}


int main(int argc, char *argv[])
{

    connectioncount = mmap(NULL, sizeof(int), PROT_READ |
PROT_WRITE,MAP_SHARED | MAP_ANONYMOUS, -1, 0);
    *connectioncount = 0;
    startsignal = mmap(NULL, sizeof(int), PROT_READ |
PROT_WRITE,MAP_SHARED | MAP_ANONYMOUS, -1, 0);
    *startsignal = 0;
    endsignal = mmap(NULL, sizeof(int), PROT_READ |
PROT_WRITE,MAP_SHARED | MAP_ANONYMOUS, -1, 0);
    *endsignal = 0;

    // Boardgrösse initialisieren
    int boardsize;
    if (argc < 3)
    {
        // Falls keine Grösse übergeben wird, ist der Defaultwert 4
        boardsize = 4;
```

```
    }
    else
    {
        boardsize = atoi(argv[2]);
    }

    playerlistgenerator();
    boardgenerator(boardsize);
    setplayername("Server");

    // Kommunikation initialisieren
    int sockfd, newsockfd, portno, pid;
    socklen_t clilen;
    struct sockaddr_in serv_addr, cli_addr;

    if (argc < 2) {
        // Falls kein Port übergeben wird, ist der Defaultport 12345
        portno = 12345;
    }
    else
    {
        portno = atoi(argv[1]);
    }

    printf("\nServerinitialisierung:\n\n");
    printf("Seitenlänge %d\n", boardsize);
    printf("Serverport %d \n\n", portno);

    // Serverfunktionenaufruf
    if (fork() == 0)
    {
        servermanager(boardsize);
    }
    // Clientfunktionenaufruf
    else
    {

        sockfd = socket(AF_INET, SOCK_STREAM, 0);
        if (sockfd < 0)
            error("ERROR opening socket");
        bzero((char *) &serv_addr, sizeof(serv_addr));

        serv_addr.sin_family = AF_INET;
        serv_addr.sin_addr.s_addr = INADDR_ANY;
        serv_addr.sin_port = htons(portno);
        if (bind(sockfd, (struct sockaddr *) &serv_addr,
                sizeof(serv_addr)) < 0)
                error("ERROR on binding");
        listen(sockfd,5);
        clilen = sizeof(cli_addr);


        while (1) {
            newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr,
&clilen);
            if (newsockfd < 0)
                error("ERROR on accept");
            pid = fork();
            if (pid < 0)
                error("ERROR on fork");
            if (pid == 0)  {
                close(sockfd);
                clientmanager(newsockfd, boardsize);
```

```
                exit(0);
            }
            else close(newsockfd);
        }

    }

    close(sockfd);
    return 0; /* we never get here */
}

/******** clientmanager() *****************
 There is a separate instance of this function
 for each connection.  It handles all communication
 once a connnection has been established.
 *****************************************/
void clientmanager (int sock, int boardsize)
{
    int n;
    char buffer[256];
    char command[256];

    // Das gehört vermutlich nicht hierher!
    int sieger = -1;

    // Warten auf HELLO
    bzero(buffer,256);
    // Übertragener String wird eingelesen
    n = read(sock,buffer,255);
    if (n < 0) error("ERROR reading from socket");

    // printf("Here is the message: %s\n",buffer);

    if (strcmp(buffer, "HELLO\n") == 0)
    {
        sprintf(buffer,"SIZE %d", boardsize);
        n = write(sock,buffer, sizeof(buffer)+1);
        if (n < 0) error("ERROR writing to socket");

        // printf("Connectioncount %d\n", *connectioncount);
        // printf("Startsignal %d\n", *startsignal);
        // printf("Endsignal %d\n", *endsignal);
        *connectioncount = *connectioncount+1;

        // printboard(boardsize);
        // printplayerlist();
    }
    else
    {
        n = write(sock,"NACK\n",5);
        if (n < 0) error("ERROR writing to socket");
        // TODO: Abbruch der Aktion
    }

    // printf("Startsignal vor Start %d\n", *startsignal);
    // printf("Connectioncount %d\n", *connectioncount);

    while (*startsignal <= 0)
    {
        // Warten auf Startsignal
        sleep(1);
    }
    n = write(sock,"START\n",6);
```

```
    if (n < 0) error("ERROR writing to socket");

    // printf("Startsignal nach Start %d\n", *startsignal);
    // printf("Connectioncount %d\n", *connectioncount);

    while(*endsignal == 0)
    {
        bzero(buffer,256);
        bzero(command,256);
        // Übertragener String wird eingelesen
        n = read(sock,buffer,255);
        if (n < 0) error("ERROR reading from socket");

        // printf("Here is the message: %s\n",buffer);

        char delimiter[] = " \n";
        char *ptr;

        // Feld-Erobern-Komando auseinanderschneiden
        ptr = strtok(buffer, delimiter);
        // printf("Kommando: %s\n", ptr);
        command[0] = ptr[0];
        int e = 0;
        int f = 0;
        ptr = strtok(NULL, delimiter);
        int len = strlen(ptr);
        for(int i=0; i<len; i++){
            e = e * 10 + ( ptr[i] - '0');
        }
        // printf("X-Value: %d\n", e);

        ptr = strtok(NULL, delimiter);
        len = strlen(ptr);
        for(int i=0; i<len; i++){
            f = f * 10 + ( ptr[i] - '0');
        }
        // printf("Y-Value: %d\n", f);


        // TAKE Verarbeitung
        if (strcmp(&command[0], "T") == 0)
        {
            // Nur das Delimeterzeichen "\n" wurde hier verwendet,
damit Clientnamen auch blanks enthalten dürfen
            ptr = strtok(NULL, "\n");
            // printf("Name: %s\n", ptr);

            if ((setfield(e,f,getplayerID(ptr),boardsize)) == 1)
            {
                n = write(sock,"TAKEN",5);
                if (n < 0) error("ERROR writing to socket");
                // printf("TAKEN Sent\n");
            }
            else
            {
                n = write(sock,"INUSE",5);
                if (n < 0) error("ERROR writing to socket");
            }
            // printboard(boardsize);
            // printplayerlist();
        }
```

```
        // STATUS Verarbeitung
        else if (strcmp(&command[0], "S") == 0)
        {
            bzero(command,256);
            strcpy(command, playername[getfield(e,f,boardsize)]);
            n = write(sock,command,strlen(command));
            if (n < 0) error("ERROR writing to socket");
        }
    }
    // Endsignal auslösen
    bzero(buffer,256);
    sprintf(buffer,"END %s", playername[*endsignal]);
    n = write(sock,buffer, sizeof(buffer)+1);
    if (n < 0) error("ERROR writing to socket");
}
```

### Client.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <arpa/inet.h>


char host[] = "localhost";
typedef struct
{
      const char *name;
      const char *port;
} config;

struct sock_s
{
      int fd;
      struct addrinfo *addr;
};

typedef struct sock_s sock_t;

struct addrinfo init_hints(int sock_type, int flags)
{
      struct addrinfo hints;

      memset(&hints, 0, sizeof(hints));
      hints.ai_family = AF_UNSPEC;
      hints.ai_socktype = sock_type;
      if (flags)
            hints.ai_flags = flags;
      return hints;
}

struct addrinfo *resolve_dns(struct addrinfo *hints, char* host,
const char *port)
{
      struct addrinfo *servinfo;

      int err = getaddrinfo(host, port, hints, &servinfo);
```

```
	if (err)
	{
		fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(err));
		exit(1);
	}
	return servinfo;
}

static sock_t connect_socket_to_address(struct addrinfo *servinfo)
{
	struct addrinfo *p;
	int sockfd;
	for (p = servinfo; p != NULL; p = p->ai_next)
	{
		sockfd = socket(p->ai_family, p->ai_socktype, p-
>ai_protocol);
		if (sockfd == -1)
		{
			perror("Client: socket");
			continue;
		}

		int err = connect(sockfd, p->ai_addr, p->ai_addrlen);
		if (err)
		{
			close(sockfd);
			perror("Client: connect");
			continue;
		}

		break;
	}
	return (sock_t) { sockfd, p };
}


int sendall(int s, char *buffer, int len)
{
	int total = 0;
	int bytesrest = len;
	int n;

	while (total < len)
	{
		n = send(s, buffer+total, bytesrest, 0);
		if (n == -1)
			break;
		total += n;
		bytesrest -= n;
	}
	return n == -1 ? -1 : 0;
}



void printUsage(char const argv[])
{
	printf("Usage: %s <name> [port] [server]\n", argv);
	printf("- Name ist ein Pflichtparameter\n");
	printf("- Port ist ein optionaler Parameter, der Defaultwert
ist: 12345\n");
```

```
	printf("- Server ist ein optionaler Parameter, der Defaultwert
ist: Localhost\n");
	exit(0);
}

config process_options(int argc, char const *argv[], config client)
{
	if (argc == 2)
	{
		client.name = argv[1];
		if (strcmp(client.name, ""))
			client.port = "12345";
		else
			printUsage(argv[0]);
	}
	else if (argc == 3)
	{
		client.name = argv[1];
		client.port = argv[2];
		if (!(atoi(client.port) && strcmp(client.name, "")))
			printUsage(argv[0]);
	}
	else if (argc == 4)
	{
		client.name = argv[1];
		client.port = argv[2];
		if (!(atoi(client.port) && strcmp(client.name, "")))
			printUsage(argv[0]);
		strcpy(host,argv[3]);
	}
	else
		printUsage(argv[0]);

	return client;
}

int main(int argc, char const *argv[])
{
	config client;
	client = process_options(argc, argv, client);

	struct addrinfo hints = init_hints(SOCK_STREAM, 0);

	struct addrinfo *servinfo = resolve_dns(&hints, host,
client.port);
	sock_t sock = connect_socket_to_address(servinfo);

	if (sock.addr == NULL)
	{
		fprintf(stderr, "Fehler bei der Verbindung\n");
		exit(2);
	}


	freeaddrinfo(servinfo);

	if (sendall(sock.fd, "HELLO\n", 7) == -1)
		perror("sendall");

	char buffer[256];
	int size = 0;

	int nbytes = recv(sock.fd, buffer, 256 - 1, 0);
```

```
	if (nbytes < 0)
	{
		perror("recv");
		exit(1);
	}
	buffer[nbytes] = '\0';

	printf("Servermessage %s\n", buffer);

	if (strncmp(buffer, "SIZE", 4) == 0)
	{
		size = atoi(buffer+5);
		memset(buffer, 0, sizeof(buffer));

		while(1)
		{
			nbytes = recv(sock.fd, buffer, 255, 0);
			if (nbytes < 0)
			{
				perror("recv");
				exit(1);
			}
			buffer[nbytes] = '\0';
			if (strcmp(buffer, "START\n") == 0)
				break;
			if (strcmp(buffer, "NACK\n") == 0)
			{
				close(sock.fd);
				exit(0);
			}
		}

		while(1)
		{
			int x,y;
			for (y = 0; y < size; ++y)
			{
				for (x = 0; x < size; ++x)
				{
					memset(buffer, 0, sizeof(buffer));

					char take[255] = "TAKE ";
					char field[32];

					sprintf(field, "%d", x);
					strcat(field, " ");
					strcat(take, field);
					memset(field, 0, sizeof(field));

					sprintf(field, "%d", y);
					strcat(field, " ");
					strcat(take, field);
					memset(field, 0, sizeof(field));

					strcat(take, client.name);
					strcat(take, "\n");

					if (sendall(sock.fd, take,
sizeof(take)) == -1)
						perror("sendall");

					do
					{
```

```
                                    nbytes = recv(sock.fd, buffer,
255, 0);
                                    if (nbytes < 0)
                                    {
                                          perror("recv");
                                          exit(1);
                                    }

                              } while(nbytes == 0);
                              buffer[nbytes] = '\0';
                              printf("Servermessage: %s\n", buffer);
                              if (strncmp(buffer, "END", 3) == 0)
                              {
                                    close(sock.fd);
                                    exit(0);
                              }
                        }
                  }
            }
      }

      return 0;
}
```