

Tesina di Sistemi Operativi
Sistema di Prenotazione Posti

D'Amico Danilo
Sistemi Operativi (6 CFU)
Anno Accademico 2019/2020

Specifica

Realizzazione di un servizio di prenotazione posti per una sala cinematografica, supportato da un server.

Ciascun posto è caratterizzato da un numero di fila, un numero di poltrona, e può essere libero o occupato. Il server accetta e processa sequenzialmente o in concorrenza (a scelta) le richieste di prenotazione di posti dei client (residenti, in generale, su macchine diverse).

Un client deve fornire ad un utente le seguenti funzioni:

1. Visualizzare la mappa dei posti in modo da individuare quelli ancora disponibili.
2. Inviare al server l'elenco dei posti che si intende prenotare (ciascun posto da prenotare viene ancora identificato tramite numero di fila e numero di poltrona).
3. Attendere dal server la conferma di effettuata prenotazione ed un codice univoco di prenotazione.
4. Disdire una prenotazione per cui si possiede un codice.

Si precisa che la specifica richiede la realizzazione sia dell'applicazione client che di quella server.

Per progetti misti Unix/Windows è a scelta quale delle due applicazioni sviluppare per uno dei due sistemi.

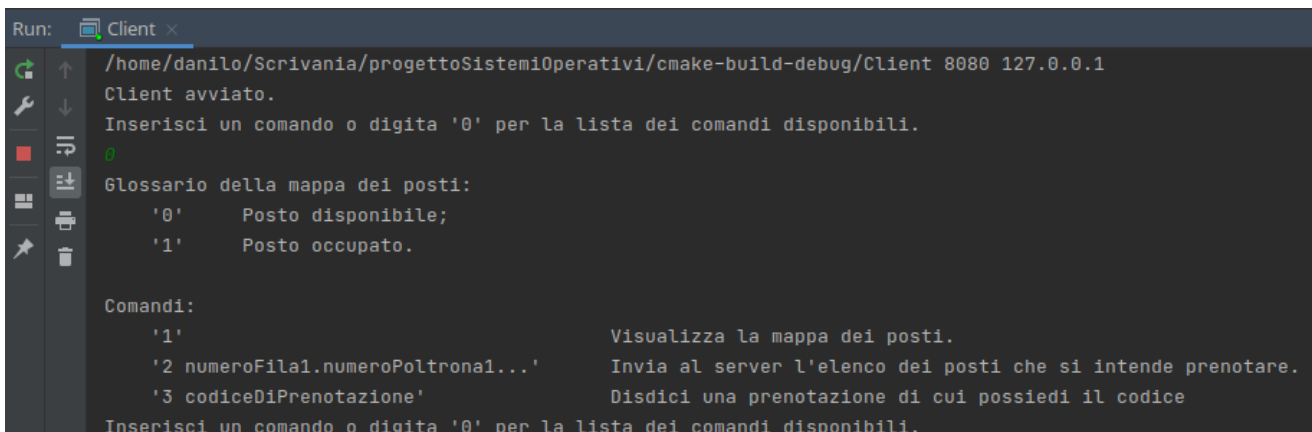
Introduzione

Il progetto è stato svolto in ambiente Linux su Manjaro KDE Plasma 20.2, utilizzando come IDE per lo sviluppo ed il testing CLion 20.2.4 con una licenza Educational fornita dall'email universitaria.

Il server accetta sequenzialmente le richieste dei client che intendono sfruttare i suoi servizi tramite protocollo TCP.

Manuale d'uso e d'installazione

Il progetto contiene due file principali: *clientMain.c* e *serverMain.c*, da eseguire rispettivamente per avviare il client ed il server.



```
Run: Client x
/home/danilo/Scrivania/progettoSistemiOperativi/cmake-build-debug/Client 8080 127.0.0.1
Client avviato.
Inserisci un comando o digita '0' per la lista dei comandi disponibili.
Glossario della mappa dei posti:
'0'      Posto disponibile;
'1'      Posto occupato.

Comandi:
'1'      Visualizza la mappa dei posti.
'2 numeroFila1.numeroPoltrona1...' Invia al server l'elenco dei posti che si intende prenotare.
'3 codiceDiPrenotazione'      Disdici una prenotazione di cui possiedi il codice
Inserisci un comando o digita '0' per la lista dei comandi disponibili.
```

Il client contiene un comando che ne spiega l'utilizzo

Segue l'elenco degli argomenti da immettere contestualmente all'esecuzione dei due file.

Per il file *clientMain.c*:

- Porta su cui è in ascolto il server;
- Indirizzo IP in cui trovare il server.

Per il file *serverMain.c*:

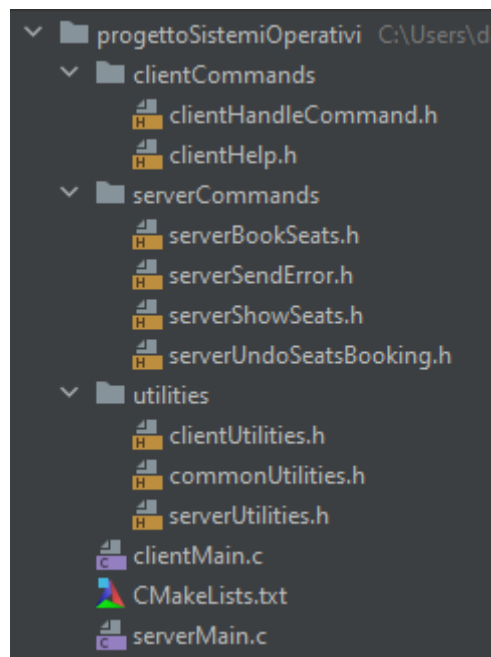
- Porta su cui mettersi in ascolto;
- Numero di file che avrà la sala cinematografica da modellare;
- Numero di posti per ogni fila che avrà la sala cinematografica da modellare.

Una volta che il client è stato avviato, l'utente può inserire il comando *0* per visualizzare a schermo un elenco dei comandi disponibili accompagnati da una breve descrizione.

Implementazione

Struttura del programma

Il progetto si articola in due file sorgente `.c`, contenenti le funzioni *main* di client e server, e tre cartelle di file di header.



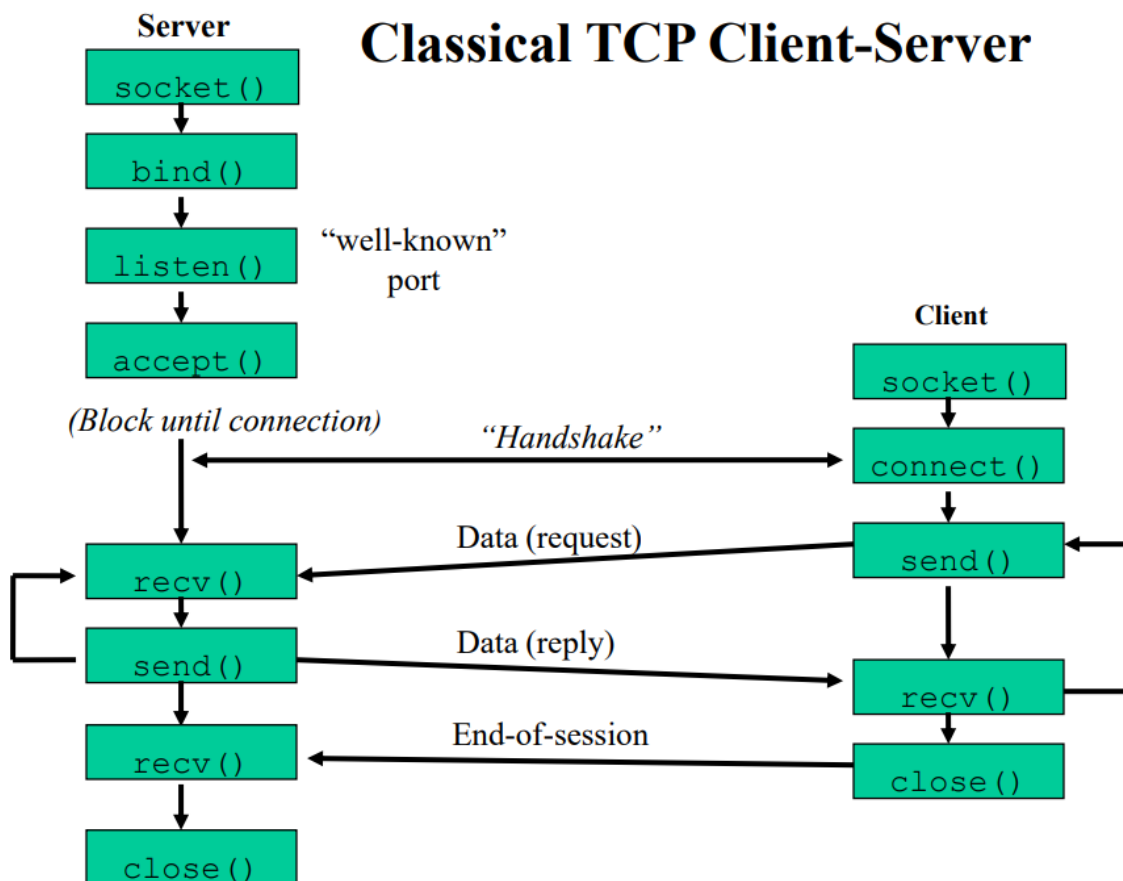
Struttura del programma

All'interno delle cartelle *clientCommands* e *serverCommands* si trovano, rispettivamente, le funzioni invocate da client e server per eseguire i comandi che vengono loro richiesti.

Tutte le altre funzioni sono inserite all'interno della cartella *utilities*, divise in tre file a seconda che siano sfruttate da client, server, o entrambi.

Implementazione protocollo TCP

La comunicazione tra client e server avviene tramite protocollo TCP. Al suo avvio, il server invoca la funzione *serverInitializeListeningSocket()*, la quale si occupa di eseguire gli step rappresentati nella slide in figura fino a *listen()*.



Una volta avviata infatti, *serverInitializeSocket()* invocherà la funzione `socket` con il parametro `SOCK_STREAM` per creare un socket d'ascolto, impostando il riutilizzo dell'indirizzo su cui farà il bind tramite *setsockopt()*. Per effettuare il bind, inizializza prima una struttura *sockaddr_in* impartendole di accettare dati su una qualsiasi delle sue interfacce di rete ed assegnandole un numero di porta in *network byte order*.

Prima di terminare, invoca la funzione *bind()* per assegnare la struttura che abbiamo inizializzato al socket e *listen()* per mettere in ascolto il server sulla porta ed impostare a *BACKLOG* (1024) il numero massimo di richieste in attesa.

```
void serverInitializeListeningSocket(int port) {
    int on = 1;
    struct sockaddr_in serverAddress;

    // crea un socket TCP d'ascolto
    if ((listeningSocket = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("errore nella creazione del socket d'ascolto");
        exit(EXIT_FAILURE);
    }

    // consente il riutilizzo dell'indirizzo con cui si farà il bind
    if (setsockopt(listeningSocket, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on)) == -1) {
        perror("Errore nell'impostazione delle opzioni del socket");
        exit(EXIT_FAILURE);
    }

    // inizializzo sockaddr_in
    bzero((void *) &serverAddress, sizeof(serverAddress));
    serverAddress.sin_family = AF_INET;
    serverAddress.sin_addr.s_addr = htonl(
        INADDR_ANY); // il server accetta dati su una qualunque delle sue interfacce di rete
    serverAddress.sin_port = htons(port); // numero di porta del server per prime comunicazioni

    // assegna l'indirizzo al socket
    if (bind(listeningSocket, (struct sockaddr *) &serverAddress, sizeof(serverAddress)) < 0) {
        perror("Errore in bind");
        closeSocket(listeningSocket);
        exit(EXIT_FAILURE);
    }

    // mettiamo il server in ascolto sulla porta
    if (listen(listeningSocket, BACKLOG) == -1) {
        perror("Errore in listen");
        closeSocket(listeningSocket);
        exit(EXIT_FAILURE);
    }
}
```

Funzione serverInitializeSocket()

Terminata l'esecuzione di *serverInitializeSocket()* il controllo torna a *serverMain.c*, che si prepara ad effettuare l'*accept()* sulle connessioni in arrivo.

```

// accetto connessione in arrivo e la gestisco sul connectionSocket
retry_accept:
if ((connectionSocket = accept(listeningSocket, NULL, NULL)) == -1) {
    if (errno == EINTR)
        goto retry_accept;
    else {
        perror("Errore nella accept");
        continue;
    }
}
}

```

Il client, invece, quando riceve dall'utente un comando che implica la richiesta di dati dal server invoca *clientInitializeSocket()*, che si occuperà della creazione del socket e dell'esecuzione della *connect()*. Durante l'inizializzazione della struttura *sockaddr_in* invoca *inet_pton()* per convertire l'IP inserito dall'utente in forma binaria, in modo che possa essere poi usato dalla *connect()*.

```

void clientInitializeSocket(int port, char *addressString) {
    struct sockaddr_in serverAddress;

    // crea il socket
    if ((socketDescriptor = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("Creazione del socket fallita.");
        exit(EXIT_FAILURE);
    }

    // inizializza serverAddress
    bzero((void *) &serverAddress, sizeof(serverAddress));
    serverAddress.sin_family = AF_INET; // assegna il tipo di indirizzo
    serverAddress.sin_port = htons(port); // assegna la porta del server

    // imposta indirizzo IP del server
    if (inet_pton(AF_INET, addressString, &serverAddress.sin_addr) <= 0) {
        perror("Conversione dell'indirizzo del server fallita.");
        exit(EXIT_FAILURE);
    }

    // connessione al server
    retry_connect:
    if (connect(socketDescriptor, (struct sockaddr *) &serverAddress, sizeof(serverAddress)) == -1) {
        if (errno == EINTR)
            goto retry_connect;
        else {
            perror("connect fallita.");
            exit(EXIT_FAILURE);
        }
    }
}
}

```


Per la lettura e scrittura di dati dalle socket, client e server si affidano alle funzioni *readSocket()* e *writeSocket()* presenti in *commonUtilities.h*.

Dato che in una connessione via *stream* non è garantito l'arrivo contemporaneo di tutti i dati come lo è invece in una connessione a pacchetti come UDP, la lettura in *readSocket()* è effettuata byte per byte fino ad arrivare al carattere di terminazione '\0', che indica la fine del *payload*.

```
int readSocket(int socketDescriptor, char *buffer) {
    int numberOfBytesRead;
    char byteRead = ' ';

    for (int i = 0; i < MAXLINE - 1; i++) {
        if ((numberOfBytesRead = recv(socketDescriptor, &byteRead, 1, 0)) == 1) {
            buffer[i] = byteRead;
            if (byteRead == '\0')
                break;
        } else {
            if (numberOfBytesRead == 0 && i == 0)
                return 0;
            if (numberOfBytesRead == 0 && i != 0) {
                *buffer = '\0';
                break;
            }
            if (errno != EINTR)
                return -1;
        }
    }
    return 1;
}
```

Allo stesso modo, *writeSocket()* scrive byte per byte il contenuto della stringa consegnatagli dal chiamante. Quando il server desidera trasmettere più messaggi come risposta al client, invoca più volte *writeSocket()* senza inserire il carattere di terminazione, premunendosi di inviarlo quando è sicuro di aver terminato le stringhe da inviare.

```

int writeSocket(int socketDescriptor, char *source, int sourceSize) {
    int numberOfBytesWritten;
    char *currentOffset;

    currentOffset = source;

    while (sourceSize) {
        if ((numberOfBytesWritten = send(socketDescriptor, source, sourceSize, 0)) <= 0) {
            if (errno == EINTR)
                numberOfBytesWritten = 0;
            else
                return -1;
        }
        sourceSize -= numberOfBytesWritten;
        currentOffset += numberOfBytesWritten;
    }
    return 1;
}

```

Per la chiusura di un socket, viene invocata *closeSocket()* da *commonUtilities.h*, che invoca a sua volta *close()* facendone un controllo sui possibili errori.

```

void closeSocket(int socketDescriptor) {
    retry_close_socket:
    if (close(socketDescriptor) < 0) {
        if (errno == EINTR) {
            goto retry_close_socket;
        } else {
            perror("Errore nella chiusura di un socket");
            exit(EXIT_FAILURE);
        }
    }
}

```

Scelta della comunicazione tramite stringhe

Client e server comunicano tra di loro solo inviandosi stringhe. Questa scelta è stata effettuata come conseguenza della necessità, nel caso si inviino *struct* contenenti i dati, di aggiungere dati di overhead per l'individuazione esatta dell'inizio e della fine di ogni *struct*.

Chiusura del sistema

Sia client che server catturano i segnali *SIGINT* e *SIGTERM* per compiere operazioni di rilascio di risorse prima della loro chiusura. La logica necessaria per impostare i segnali è rappresentata in client da *clientSetSignals()* in *clientUtilities.h* e nel server da *serverSetSignals()* in *serverUtilities.h*.

```
void clientSetSignals() {
    struct sigaction act;
    act.sa_handler = clientCloseGracefully;

    if (sigfillset(&act.sa_mask) == -1) {
        perror("errore nel riempimento di .sa_mask.");
        exit(EXIT_FAILURE);
    }

    if (sigaction(SIGINT, &act, 0) == -1) {
        perror("errore nella gestione di SIGINT");
        exit(EXIT_FAILURE);
    }

    if (sigaction(SIGTERM, &act, 0) == -1) {
        perror("errore nella gestione di SIGTERM");
        exit(EXIT_FAILURE);
    }
}
```

```

void serverSetSignals() {
    struct sigaction act;
    act.sa_handler = serverCloseGracefully;

    if (sigfillset(&act.sa_mask) == -1) {
        perror("errore nel riempimento di .sa_mask.");
        exit(EXIT_FAILURE);
    }

    if (sigaction(SIGINT, &act, 0) == -1) {
        perror("errore nella gestione di SIGINT");
        exit(EXIT_FAILURE);
    }

    if (sigaction(SIGTERM, &act, 0) == -1) {
        perror("errore nella gestione di SIGTERM");
        exit(EXIT_FAILURE);
    }
}

```

In entrambi i casi viene creata una struttura *sigaction* che, una volta assegnata ai comandi interessati tramite l'omonima funzione, si occuperà di chiamare un *handler* del comando assicurandosi che tutti i segnali ricevuti nel frattempo siano bloccati.

Per il client questa funzione è *clientCloseGracefully()*. Questa funzione si occupa semplicemente di chiudere il socket su cui il client è connesso nel caso questo esista.

```

void clientCloseGracefully(int signo) {
    if (socketDescriptor != -1)
        close(socketDescriptor);
    exit(EXIT_SUCCESS);
}

```

L'equivalente funzione del server invece, *serverCloseGracefully*, si occupa anche di rilasciare le risorse allocate durante l'esecuzione dal programma.

```

void serverCloseGracefully(int signo) {
    char file[MAXLINE] = {0};

    closeSocket(listeningSocket);

    for (int row = 0; row < rowsNumber; row++) {
        free(movieTheatreGrid[row]);
    }

    free(movieTheatreGrid);

    strcpy(file, getenv("HOME"));
    strcat(file, directoryName);
    strcat(file, codesName);

    remove(file);
    exit(EXIT_SUCCESS);
}

```

Comando ShowSeats

Il comando ShowSeats viene eseguito quando l'utente indica *1* come comando. Il client invia questa richiesta al server, che la risolve mediante la funzione *serverShowSeats()*. L'elenco dei posti è modellato come una lista di liste di char, ognuna contenente il valore '0' se il posto è libero o '1' se invece è occupato.

```
void serverShowSeats() {
    printf("serverShowSeats richiesto.\n");
    int charSize = sizeof(char);
    char newline = '\n';
    char terminator = '\0';

    // invia la griglia dei posti al client
    for (int row = 0; row < rowsNumber; row++) {
        for (int seat = 0; seat < seatsNumber; seat++) {

            writeSocket(connectionSocket, &movieTheatreGrid[row][seat], charSize);

            if (seat == seatsNumber - 1)
                writeSocket(connectionSocket, &newline, charSize);
        }
    }
    writeSocket(connectionSocket, &terminator, charSize);
}
```

serverShowSeats() invia allora il contenuto di ogni cella della tabella, aggiungendo un carattere *newline* alla fine di ogni fila. Una volta terminato l'invio dei posti, aggiunge un carattere di terminazione per comunicare al client in lettura la fine del *payload* dei dati.

Comando BookSeats

Nell'esecuzione del comando di prenotazione posti, il server si aspetta di ricevere il comando 2 seguito da una serie di numeroFila.numeroPoltrona. Nel caso in cui non dovesse trovare questa sintassi nel comando invierà al client un messaggio d'errore tramite *serverSendError()*.

```
while (currentToken != NULL) {
    char *tokenContext;

    // conversione del numero di fila
    if ((currentRowChar = strtok_r(currentToken, ".", &tokenContext)) == NULL) {
        perror("Errore nella lettura del numero di fila dal comando client");
        serverSendError();
        return;
    }

    if ((currentRow = convertStringToNaturalInt(currentRowChar)) == -1) {
        perror("Errore nella conversione del numero di fila dal comando");
        serverSendError();
        return;
    }

    // conversione del numero di posto
    if ((currentSeatChar = strtok_r(NULL, ".", &tokenContext)) == NULL) {
        perror("Errore nella lettura del numero di posto dal comando client");
        serverSendError();
        return;
    }

    if ((currentSeat = convertStringToNaturalInt(currentSeatChar)) == -1) {
        perror("Errore nella conversione del numero di posto dal comando");
        serverSendError();
        return;
    }
}
```

Per estrarre i dati d'interesse dalla stringa, la funzione invoca *strtok_r()* con il contesto di esecuzione *stringContext* per spezzare la stringa in corrispondenza dei punti e *strtok_r()* con il contesto di esecuzione *tokenContext* per spezzare i *token* così ottenuti in corrispondenza del punto.

Ottenuti così due stringhe contenenti il numero di fila e posto desiderati, vengono convertite in interi da *convertStringToNaturalInt()*, che ha la responsabilità di chiamare *strtol()* controllandone gli errori ed assicurandosi che il risultato sia un numero maggiore o uguale a zero. Nel caso uno di questi controlli fallisca la funzione terminerà la propria esecuzione inviando un messaggio d'errore al client.

```
// scrittura
if (currentRow < rowsNumber && currentSeat < seatsNumber) {

    if (movieTheatreGrid[currentRow][currentSeat] == freeSeat) {

        uniqueCode = generateUniqueBookingCode();

        snprintf(uniqueCodeLog, MAXLINE, "%llu %d %d\n", uniqueCode, currentRow, currentSeat);

        if (write(fileDescriptor, uniqueCodeLog, strlen(uniqueCodeLog)) != strlen(uniqueCodeLog)) {
            perror("Errore nella scrittura di un codice di prenotazione");
            serverSendError();
            exit(EXIT_FAILURE);
        }

        movieTheatreGrid[currentRow][currentSeat] = '1';

        snprintf(userMessage, MAXLINE,
            "Il posto fila %d, poltrona %d è stato prenotato. L'identificativo della prenotazione è: %llu\n",
            currentRow, currentSeat, uniqueCode);
        fflush(stdout);
        if (!writeSocket(connectionSocket, userMessage, (int) strlen(userMessage))) { // non inserisco '\0'
            perror("Errore nell'invio della risposta ad un client.");
            return;
        }
        bzero((void *) &userMessage, sizeof(userMessage));

    } else {

        snprintf(userMessage, MAXLINE,
            "Il posto fila %d, poltrona %d è già stato prenotato. Non è stato possibile effettuare la prenotazione\n",
            currentRow, currentSeat);
        fflush(stdout);
        if (!writeSocket(connectionSocket, userMessage, (int) strlen(userMessage))) { // non inserisco '\0'
            perror("Errore nell'invio della risposta ad un client.");
            return;
        }
        bzero((void *) &userMessage, sizeof(userMessage));

    }

} else {

    snprintf(userMessage, MAXLINE, "Il posto indicato non è valido.\n");
    fflush(stdout);
}
```



```

    } else {
        snprintf(userMessage, MAXLINE,
            "Il posto file %d, poltrona %d è già stato prenotato. Non è stato possibile effettuare la prenotazione\n",
            currentRow, currentSeat);
        fflush(stdout);
        if (!writeSocket(connectionSocket, userMessage, (int) strlen(userMessage))) { // non inserisco '\0'
            perror("Errore nell'invio della risposta ad un client.");
            return;
        }
        bzero((void *) &userMessage, sizeof(userMessage));
    }
} else {
    snprintf(userMessage, MAXLINE, "Il posto indicato non è valido.\n");
    fflush(stdout);
    if (!writeSocket(connectionSocket, userMessage, (int) strlen(userMessage))) { // non inserisco '\0'
        perror("Errore nell'invio della risposta ad un client.");
        return;
    }
    bzero((void *) &userMessage, sizeof(userMessage));
}
currentToken = strtok_r(NULL, " ", &stringContext);
}

// scrivo il terminatore '\0' per indicare la fine del messaggio del server
if (!writeSocket(connectionSocket, &terminator, sizeof(char))) { // non inserisco '\0'
    perror("Errore nell'invio della risposta ad un client.");
    return;
}

close(fileDescriptor);
}

```

Una volta ottenuti gli interi cercati, viene controllato che siano compatibili con le dimensioni della sala cinematografica in memoria e, in caso affermativo, si controllerà se il posto desiderato è disponibile. Nel caso questo avvenga il posto viene prenotato, viene scritto in un file il codice di prenotazione generato sul momento e viene inviata risposta affermativa al client. Nel caso il posto fallisca uno di questi controlli, invece, viene inviato un messaggio d'errore.

Alla fine della funzione viene scritto nella socket il carattere di terminazione '\0'.

Comando UndoSeatsBooking

Il comando UndoSeatsBooking viene invocato con il numero 3 dall'utente, seguito dall'elenco dei codici di prenotazione che desidera disdire.

```
void serverUndoSeatsBooking(char *bookingCodes, char *fileName) {
    printf("serverUndoSeatsBooking richiesto.\n");
    FILE *fileStream;
    char buffer[MAXLINE] = {0}, rowChar[MAXLINE] = {0}, seatChar[MAXLINE] = {0};
    int row, seat;
    char userMessage[MAXLINE];
    char *currentCode, *codeContext, terminator = '\0';

    if (checkArgumentsExistence(bookingCodes, connectionSocket) == -1)
        return;

    char *filePath = malloc(strlen(getenv("HOME")) + strlen(directoryName) + strlen(fileName) + 1);
    if (filePath == NULL) {
        perror("Allocazione della memoria per il path del file dei codici fallita");
        serverSendError();
        return;
    }

    strcpy(filePath, getenv("HOME"));
    strcat(filePath, directoryName);
    strcat(filePath, fileName);

    retry_file_fopen:
    if ((fileStream = fopen(filePath, "r+")) == NULL) {
        if (errno == EINTR)
            goto retry_file_fopen;
        else {
            serverSendError();
            perror("Apertura del file dei codici fallita.");
            return;
        }
    }

    free(filePath);
    currentCode = strtok_r(bookingCodes, " ", &codeContext);
```

La sua gestione nel server è affidata alla funzione *serverUndoSeatsBooking()*. Appena avviata, questa funzione apre il file contenente i codici per prepararsi a cercarvi i codici ricevuti.

```

while (currentCode != NULL) {
    if (fseek(fileStream, 0, SEEK_SET) == -1) {
        perror("Errore nel riposizionamento dell'indice nel file dei codici");
        return;
    }
    fileNotFinished = 1;

    while (fileNotFinished) { // scorro l'intero file in cerca del codice richiesto

        bzero((void *) &buffer, sizeof(buffer));
        bzero((void *) &rowChar, sizeof(rowChar));
        bzero((void *) &seatChar, sizeof(seatChar));

        if (readNextWord(fileStream, buffer))
            break;

        if (readNextWord(fileStream, rowChar))
            break;

        if (readNextWord(fileStream, seatChar)) {
            // questa è l'ultima riga valida del file
            fileNotFinished = 0;
        }

        if (strstr(buffer, currentCode) != NULL) {

            if ((row = convertStringToNaturalInt(rowChar)) == -1) {
                perror("Errore nella lettura del numero di fila dal file dei codici");
                return;
            }

            if ((seat = convertStringToNaturalInt(seatChar)) == -1) {
                perror("Errore nella lettura del numero di posto dal file dei codici");
                return;
            }

            movieTheatreGrid[row][seat] = '0';

            deletePreviousLine(fileStream);
        }
    }
}

```

```

deletePreviousLine(fileStream);

bzero((void *) &userMessage, sizeof(userMessage));
snprintf(userMessage, MAXLINE, "La prenotazione con il codice %s è stata annullata\n", currentCode);
if (!writeSocket(connectionSocket, userMessage,
    (int) strlen(userMessage))) { // non inserisco il terminatore '\0'
    perror("Errore nell'invio della risposta ad un client.");
    return;
}
break;
}
}

currentCode = strtok_r(NULL, " ", &codeContext);
}

// invio il carattere di terminazione
if (!writeSocket(connectionSocket, &terminator, sizeof(char))) { // non inserisco il terminatore '\0'
    perror("Errore nell'invio della risposta ad un client.");
    return;
}

fclose(fileStream);
}

```

Il server chiama allora *strtok_r* sugli argomenti ricevuti per dividerli in token e si appresta a cercare se questi combaciano con la prima parola (convertita in intero) di ogni riga del file. Nel caso questo sia vero pone a '0' il valore del posto collegato, cancella la riga in questione dal file e comunica il completamento dell'operazione al client.

```

void deletePreviousLine(FILE *fileStream) {
    char characterRead;

    if (fseek(fileStream, -2, SEEK_CUR) == -1) {
        perror("Errore nel riposizionamento dell'indice del file");
        exit(EXIT_FAILURE);
    }

    while (fread(&characterRead, 1, 1, fileStream) == 0) {
        perror("Errore nella lettura da file");
    }

    while (characterRead != '\n') {
        if (fseek(fileStream, -1, SEEK_CUR) == -1) {
            perror("Errore nel riposizionamento dell'indice del file");
            exit(EXIT_FAILURE);
        }

        if (characterRead != ' ') {
            if (fputc('*', fileStream) == EOF) {
                perror("Errore nella scrittura del file");
                exit(EXIT_FAILURE);
            }
        } else {
            if (fseek(fileStream, +1, SEEK_CUR) == -1) {
                perror("Errore nel riposizionamento dell'indice del file");
                exit(EXIT_FAILURE);
            }
        }

        if ((ftell(fileStream) == 1))
            break;

        if (fseek(fileStream, -2, SEEK_CUR) == -1) {
            perror("Errore nel riposizionamento dell'indice del file");
            exit(EXIT_FAILURE);
        }

        while (fread(&characterRead, 1, 1, fileStream) == 0) {
            perror("Errore nella lettura da file");
        }
    }
}

```

La cancellazione della riga che si è appena letto è effettuata dalla funzione *deletePreviousLine()* in *serverUtilities.h*. Questa funzione scorre al contrario la riga di un carattere per leggerlo e verificare che non sia una *newline*. In tal caso, se è parte di una parola lo sostituisce con un asterisco e sposta l'indice del file indietro per leggere il carattere precedente.