Relazione progetto Reti dei Calcolatori

Traccia Università

Simone Acampora: 0124002485
Lorenzo Arcopinto: 0124002626
Daniele Gaudino: 0124002544

Docente: Emanuel di Nardo
Anno Accademico 2023/2024

Indice

1 Descrizione del progetto

In questa sezione, vedremo la traccia che ci è stata assegnata e le note di sviluppo della consegna

1.1 Traccia - Università

- **Gruppo 1 studente**
  - **Segreteria:**
    - Inserisce gli esami sul server dell'università (salvare in un file o conservare in memoria il dato).
    - Inoltra la richiesta di prenotazione degli studenti al server universitario.
    - Fornisce allo studente le date degli esami disponibili per l'esame scelto dallo studente.
  - **Studente:**
    - Chiede alla segreteria se ci siano esami disponibili per un corso.
    - Invia una richiesta di prenotazione di un esame alla segreteria.
  - **Server universitario:**
    - Riceve l'aggiunta di nuovi esami.
    - Riceve la prenotazione di un esame.
- **Gruppo 2 studenti**
  - Il server universitario, ad ogni richiesta di prenotazione, invia alla segreteria il numero di prenotazione progressivo assegnato allo studente.
  - La segreteria a sua volta inoltra il numero di prenotazione allo studente.
- **Gruppo 3 studenti**
  - Se la segreteria non risponde alla richiesta dello studente, questo deve ritentare la connessione per 3 volte.
  - Se le richieste continuano a fallire, lo studente aspetta un tempo random e ritenta.
  - Simulare un timeout della segreteria in modo da testare l'attesa random.

1.2 Note di sviluppo

La prova d'esame richiede la progettazione e lo sviluppo della traccia proposta. Il progetto deve essere sviluppato secondo le seguenti linee:

- utilizzare un linguaggio di programmazione a scelta (C, Java, Python, etc...)
- utilizzare una piattaforma Unix-like;
- utilizzare le socket;
- inserire sufficienti commenti;

2 Descrizione e schema dell'architettura

In questa sezione andremo a descrivere gli strumenti utilizzati e mostreremo lo schema dell'architettura, specificando i ruoli di ogni componente.

2.1 Strumenti di sviluppo

La traccia proposta è stata sviluppata utilizzando il linguaggio di programmazione C e le socket su piattaforma Unix-like, proprio come richiesto.

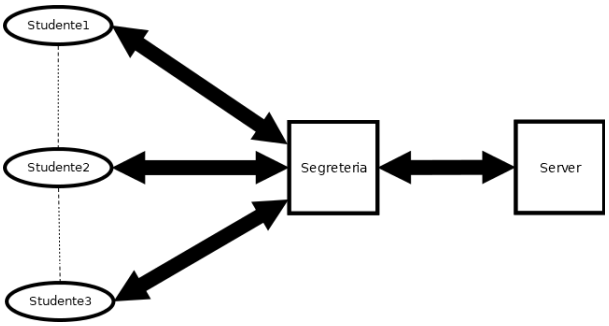
2.2 Descrizione dell'architettura

Il sistema utilizza un'architettura client-server in cui le tre componenti principali comunicano tra di loro utilizzando socket TCP/IP per la trasmissione dei dati. Le tre componenti principali sono:

- **Server Universitario**: Rappresenta il server principale a cui si connette la segreteria. Gestisce le richieste degli studenti che arrivano dalla segreteria.
- **Segreteria**: Funge da intermediario tra lo Studente e il Server universitario. Agisce come client nei confronti del Server universitario quando aggiunge gli esami e inoltra le richiesta degli studenti, e come server nei confronti degli Studenti, fornendogli la lista degli Esami.
- **Studente**: Interagisce con la segreteria per prenotare gli esami e ricevere informazioni.

2.3 Schema dell'architettura

In figura abbiamo una rappesentazione dell'architettura appena descritta.



3 Dettagli implementativi del codice sviluppato

In questa sezione discuteremo dell'implementazione dei client/server con parte del codice sviluppato.

3.1 Libreria Wrapper.h

La libreria **Wrapper.h** è stata scritta per riunire tutte le funzioni riguardanti le socket e il loro funzionamento. Di seguito una lista di tutte le funzioni nel file e il loro utilizzo:

- **Socket(int, int, int)**: Crea un socket con la specifica famiglia di indirizzi, tipo di socket, e protocollo. Se si verifica un errore, la funzione termina il programma.
- **Bind(int, const struct sockaddr *, socklen t)**: Associa un socket a un indirizzo specifico (indirizzo IP e porta). Se fallisce, termina il programma.
- **Ascolta(int, int)**:Imposta il socket in modalità di ascolto, pronto per accettare connessioni in entrata. Il numero di connesioni è deciso dall'utente. Se fallisce, termina il programma.
- **Connetti(int, const struct sockaddr *, socklen t)**: Tenta di stabilire una connessione su un socket a un indirizzo di server specificato. Se fallisce, termina il programma.
- **Accetta(int, struct sockaddr *, socklen t *)**:Accetta una connessione in entrata su un socket in ascolto e restituisce un nuovo socket per la connessione accettata. Se fallisce, termina il programma.

Vediamo anche l'implementazione codice sul file **Wrapper.c**

```c
int Socket(int family, int type, int protocol)
{
    int n;
    if ((n = socket(family, type, protocol)) < 0)
    {
        perror("socket error");
        exit(1);
    }
    return n;
}

void Bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen)
{
    if (bind(sockfd, addr, addrlen) < 0)
    {
        perror("bind error");
        exit(1);
    }
}

void Ascolta(int sockfd, int queueLen)
{
    if (listen(sockfd, queueLen) < 0)
    {
        perror("listen error");
        exit(1);
    }
}

int Accetta(int sockfd, struct sockaddr *clientaddr, socklen_t *addr_dim)
{
    int n;
    if ((n = accept(sockfd, clientaddr, addr_dim)) < 0)
    {
        perror("accept error");
        exit(1);
    }
    return n;
}

void Connetti(int sockfd, const struct sockaddr *servaddr, socklen_t addr_dim)
{
    if (connect(sockfd, (const struct sockaddr *)servaddr, addr_dim) < 0)
    {
        fprintf(stderr, "connect error\n");
        exit(1);
    }
}
```

3.2 Dettagli Server universitario

Il Server universitario svolge le richieste che gli arrivano dalla segreteria. Presenta tre Struct per la richiesta che arriva, gli esami e la prenotazione di un esame.

```c
// Definizione delle struct
struct Esame
{
    char nome[100];
    char data[100];
};

struct Richiesta
{
    int TipoRichiesta;
    struct Esame esame;
};

struct Prenotazione
{
    struct Esame esame;
    int NumPrenotazione;
    char Matricola[11];
};
```

Avviato il programma abbiamo che inizialmente viene creata la socket di ascolto per le connessioni, verrà configurato l'indirizzo per ascoltare connessioni da ogni indirizzo sulla porta **6940**, effettua il bind e si metterà in ascolto di connessioni. Terminato ciò, si entra in un ciclo while infinito in cui il server universitario accetta connessioni in arrivo: in particolare, viene eseguita una **fork** del processo corrente per creare un processo figlio. Se la fork ha successo, il processo figlio continua l'esecuzione e si occuperà di gestire le richieste della segreteria, mentre il processo padre tornerà ad accettare altre connessioni. Se la fork fallisce, viene stampato un messaggio di errore e il programma viene chiuso.

```c
int main()
{
    int universita_connessione_socket;
    int universita_ascolto_socket;
    struct sockaddr_in indirizzo_universita;
    struct Richiesta richiesta_ricevuta;

    // Creazione del socket di ascolto
    universita_ascolto_socket = Socket(AF_INET, SOCK_STREAM, 0);

    //Configurazione dell'indirizzo del server universitario
    indirizzo_universita.sin_family = AF_INET;
    indirizzo_universita.sin_addr.s_addr = htonl(INADDR_ANY);
    indirizzo_universita.sin_port = htons(6940);

    // Associazione del socket all'indirizzo e porta
    Bind(universita_ascolto_socket, (struct sockaddr *)&indirizzo_universita, sizeof(indirizzo_universita));

    // Inizio dell'ascolto
    Ascolta(universita_ascolto_socket, 10);
    printf("Server in ascolto sulla porta 6940...\n");

    while (1)
    {
        // Accettazione di una connessione in ingresso
        universita_connessione_socket = Accetta(universita_ascolto_socket, (struct sockaddr *)NULL, NULL);

        // Creazione di un processo figlio per gestire la richiesta
        pid_t pid = fork();
        if (pid < 0)
        {
            perror("Errore nella fork del server universitario");
            exit(EXIT_FAILURE);
        }

        if (pid == 0)
        { // Processo figlio
```

Dentro al processo figlio abbiamo la gestione delle richieste tramite **TipoRichiesta** che in base al suo valore porterà la segreteria a chiamare la funzione che gestisce il TipoRichiesta specifico in particolare

```c
// Lettura della richiesta dal client
ssize_t bytes_read = read(universita_connessione_socket, &richiesta_ricevuta, sizeof(struct Richiesta));
if (bytes_read != sizeof(struct Richiesta))
{
    perror("Errore nella lettura della richiesta");
    close(universita_connessione_socket);
    exit(EXIT_FAILURE);
}

// Gestione della richiesta in base al tipo
if (richiesta_ricevuta.TipoRichiesta == 1)
{
    //Aggiunta di un esame nuovo
    aggiungi_esame_file(richiesta_ricevuta.esame);
}
else if (richiesta_ricevuta.TipoRichiesta == 2)
{
    //Gestione della prenotazione di uno studente
    gestisci_prenotazione(universita_connessione_socket, richiesta_ricevuta);
}
else if (richiesta_ricevuta.TipoRichiesta == 3)
{
    //Visualizzazione degli esami disponibili sul file "esami.txt"
    gestisci_esami_disponibili(universita_connessione_socket, richiesta_ricevuta);
}
else
{
    fprintf(stderr, "Tipo di richiesta non valido\n");
}

// Chiusura del socket di connessione
close(universita_connessione_socket);
exit(EXIT_SUCCESS);
```

aggiungi_esame_file è una funzione che gestisce l'aggiunta dell'esame mandato dalla segreteria sul file **"esami.txt"** mettendo nome dell'esame e data dell'esame.

```c
// Funzione per aggiungere un esame al file
void aggiungi_esame_file(struct Esame esame)
{
    FILE *Lista_esami = fopen("esami.txt", "a");
    if (Lista_esami == NULL)
    {
        perror("Errore apertura file esami.txt");
        exit(EXIT_FAILURE);
    }

    // Scrittura dell'esame nel file usando fprintf
    if (fprintf(Lista_esami, "%s,%s\n", esame.nome, esame.data) < 0)
    {
        perror("Errore scrittura file esami.txt");
        fclose(Lista_esami);
        exit(EXIT_FAILURE);
    }

    fclose(Lista_esami);
    printf("Esame aggiunto con successo\n");
}
```

Gestisci_prenotazione è una funzione che gestisce la richiesta di prenotazione per un esame che arriva dalla segreteria, leggerà la matricola dello studente, e scriverà sul file **"prenotazioni.txt"** numero della prenotazione, nome dell'esame, data dell'esame e matricola associata a quella prenotazione.

```c
int gestisci_prenotazione(int universita_connessione_socket, struct Richiesta richiesta_ricevuta)
{
    struct Prenotazione prenotazione;
    int esito_prenotazione = 1;

    prenotazione.esame = richiesta_ricevuta.esame;

    // Ottieni un numero di prenotazione unico per l'esame specifico dal file `prenotazioni.txt`
    prenotazione.NumPrenotazione = incrementaContatoreDaPrenotazioni(richiesta_ricevuta.esame);


    // Ricezione della matricola dallo studente tramite la segreteria
    ssize_t MatricolaStudente = read(universita_connessione_socket, prenotazione.Matricola, sizeof(prenotazione.Matricola) - 1);
    if (MatricolaStudente <= 0)
    {
        if (MatricolaStudente == 0)
        {
            fprintf(stderr, "Connessione chiusa dalla segreteria durante la ricezione della matricola.\n");
        }
        else
        {
            perror("Errore ricezione matricola");
        }
        return -1;
    }

    prenotazione.Matricola[MatricolaStudente] = '\0';
```

```c
    ssize_t numero_prenotazione = write(universita_connessione_socket, &prenotazione.NumPrenotazione, sizeof(prenotazione.NumPrenotazione));
    if (numero_prenotazione != sizeof(prenotazione.NumPrenotazione))
    {
        perror("Errore invio numero prenotazione");
        return -1;
    }

    printf("Prenotazione ricevuta per esame: %s, data: %s, matricola: %s, numero:%d\n", prenotazione.esame.nome, prenotazione.esame.data, prenotazione.Matricola, prenotazione.NumPrenotazione);

    // Scrivi la prenotazione nel file "prenotazioni.txt"
    FILE *file_prenotazioni = fopen("prenotazioni.txt", "a");
    if (file_prenotazioni == NULL)
    {
        perror("Errore apertura file prenotazioni.txt");
        esito_prenotazione = 0;
    }
    else
    {
        // Assicurati che i dati siano scritti correttamente
        fprintf(file_prenotazioni, "%d,%s,%s,%s\n",
                prenotazione.NumPrenotazione,
                prenotazione.esame.nome,
                prenotazione.esame.data,
                prenotazione.Matricola);
        fclose(file_prenotazioni);
    }

    // Invia l'esito della prenotazione
    ssize_t bytes_written = write(universita_connessione_socket, &esito_prenotazione, sizeof(esito_prenotazione));
    if (bytes_written != sizeof(esito_prenotazione))
    {
        perror("Errore invio esito prenotazione");
        return -1;
    }

    return 0;
}
```

incrementaContatoreDaPrenotazioni è una funzione che serve a restituire il numero progressivo per la prenotazione dell'esame, controllando che nel file **"prenotazioni.txt"** non ci sia già una persona prenotata per quella data di esame specifica. Questa funzione viene usata in gestisci prenotazione.

```c
int incrementaContatoreDaPrenotazioni(struct Esame esame) {
    FILE *file = fopen("prenotazioni.txt", "r");
    if (file == NULL) {
        // Se il file non esiste, inizializza il contatore a 1
        return 1;
    }

    char buffer[256];
    int maxContatore = 0;

    // Leggi ogni riga del file e cerca le prenotazioni per l'esame specifico
    while (fgets(buffer, sizeof(buffer), file) != NULL) {
        struct Esame esameFile;
        int contatore;
        char matricola[11];

        // Supponendo che il file `prenotazioni.txt` abbia il formato: numero, nome, data, matricola
        sscanf(buffer, "%d,%99[^,],%99[^,],%10s", &contatore, esameFile.nome, esameFile.data, matricola);

        // Controlla se l'esame corrente nel file corrisponde a quello passato alla funzione
        if (strcmp(esameFile.nome, esame.nome) == 0 && strcmp(esameFile.data, esame.data) == 0) {
            if (contatore > maxContatore) {
                maxContatore = contatore;
            }
        }
    }

    fclose(file);
    return maxContatore + 1; // Incrementa di uno per il nuovo numero di prenotazione
}
```

gestisci_esami_disponibili è una funzione che serve a restituire il numero di esami con il nome ricercato dallo studente alla segreteria in modo che possa mandare la lista completa allo studente.

```c
void gestisci_esami_disponibili(int socket, struct Richiesta richiesta_ricevuta)
{
    FILE *Lista_esami = fopen("esami.txt", "r");
    struct Esame esami_disponibili[100]; // Supponiamo un massimo di 100 esami
    int numero_esami = 0;
    char nome[100], data[100];

    if (Lista_esami == NULL)
    {
        perror("Errore apertura file esami.txt");
        exit(EXIT_FAILURE);
    }

    // Leggiamo gli esami dal file e li filtriamo per nome
    while (fscanf(Lista_esami, "%99[^,],%99[^\n]\n", nome, data) == 2)
    {
        if (strcmp(nome, richiesta_ricevuta.esame.nome) == 0)
        {
            strcpy(esami_disponibili[numero_esami].nome, nome);
            strcpy(esami_disponibili[numero_esami].data, data);
            numero_esami++;
        }
    }
    fclose(Lista_esami);

    // Inviamo il numero di esami disponibili alla segreteria
    if (write(socket, &numero_esami, sizeof(numero_esami)) != sizeof(numero_esami))
    {
        perror("Errore invio numero esami alla segreteria");
        exit(EXIT_FAILURE);
    }

    // Inviamo gli esami disponibili alla segreteria
    if (write(socket, esami_disponibili, sizeof(struct Esame) * numero_esami) != sizeof(struct Esame) * numero_esami)
    {
        perror("Errore invio esami disponibili alla segreteria");
        exit(EXIT_FAILURE);
    }
}
```

3.3 Segreteria

La segreteria svolge le richieste che gli arrivano da studente e può anche aggiungere nuovi esami sul file **"esami.txt"**. Presenta due Struct per la richiesta che arriva e gli esami, uguali a quelle del Server.

Nell'aggiunta degli esami la segreteria manda i dati immessi dalla segreteria e il TipoRichiesta che riguarda l'aggiunta di un nuovo esame tramite **inserisci_nuovo_esame** e **mandaEsameNuovoServer**

```c
void inserisci_nuovo_esame()
{
    struct Esame nuovo_esame;

    printf("Inserisci il nome dell'esame: ");
    fgets(nuovo_esame.nome, sizeof(nuovo_esame.nome), stdin);
    nuovo_esame.nome[strcspn(nuovo_esame.nome, "\n")] = '\0';

    printf("Inserisci la data dell'esame (YYYY-MM-DD): ");
    fgets(nuovo_esame.data, sizeof(nuovo_esame.data), stdin);
    nuovo_esame.data[strcspn(nuovo_esame.data, "\n")] = '\0';

    mandaEsameNuovoServer(nuovo_esame);
}

void mandaEsameNuovoServer(struct Esame esame)
{
    int socket_segreteria;
    struct sockaddr_in indirizzo_universita;
    struct Richiesta aggiuntaEsame;

    aggiuntaEsame.TipoRichiesta = 1;
    aggiuntaEsame.esame = esame;

    socket_segreteria = connessione_universita(&indirizzo_universita);

    if (write(socket_segreteria, &aggiuntaEsame, sizeof(aggiuntaEsame)) != sizeof(aggiuntaEsame))
    {
        perror("Errore manda esame server");
        exit(EXIT_FAILURE);
    }

    printf("Esame aggiunto con successo!\n");

    close(socket_segreteria);
}
```

esami_disponibili invece è utilizzata per richiedere al server il numero di esami con il nome dato dallo studente

```c
void esami_disponibili(struct Esame esame, int segreteria_connessione_socket)
{
    int socket_esami;
    struct sockaddr_in indirizzo_universita;
    struct Richiesta ricezione_esami;
    struct Esame esami_disponibili[100]; // Assumiamo che ci siano al massimo 100 esami
    int numero_esami = 0;

    ricezione_esami.TipoRichiesta = 3; // TipoRichiesta 3 per ottenere gli esami disponibili
    ricezione_esami.esame = esame;

    socket_esami = connessione_universita(&indirizzo_universita);
    invio_esame_server(socket_esami, ricezione_esami);

    // Riceviamo il numero di esami dal server universitario
    if (read(socket_esami, &numero_esami, sizeof(numero_esami)) != sizeof(numero_esami))
    {
        perror("Errore ricezione numero esami dal server universitario");
        close(socket_esami);
        exit(EXIT_FAILURE);
    }

    // Inviamo il numero di esami allo studente
    if (write(segreteria_connessione_socket, &numero_esami, sizeof(numero_esami)) != sizeof(numero_esami))
    {
        perror("Errore invio numero esami allo studente");
        close(socket_esami);
        exit(EXIT_FAILURE);
    }

    // Riceviamo gli esami disponibili dal server universitario
    if (read(socket_esami, esami_disponibili, sizeof(struct Esame) * numero_esami) != sizeof(struct Esame) * numero_esami)
    {
        perror("Errore ricezione lista esami dal server universitario");
        close(socket_esami);
        exit(EXIT_FAILURE);
    }

    // Inviamo la lista di esami disponibili allo studente
    if (write(segreteria_connessione_socket, esami_disponibili, sizeof(struct Esame) * numero_esami) != sizeof(struct Esame) * numero_esami)
    {
        perror("Errore invio lista esami allo studente");
        close(socket_esami);
        exit(EXIT_FAILURE);
    }

    printf("Lista esami inviata allo studente\n");
}
```

Riguardo invece le richieste degli studenti, una volta messa in ascolto, Segreteria così come Server entra in un ciclo while infinito in cui accetta le connessioni in arrivo e gestisce le loro richieste; anche qui eseguiamo una fork del processo corrente per creare un processo figlio. Se la fork ha successo, il figlio continuerà l'esecuzione. Se la fork fallisce invece viene stampato un messaggio di errore e il programma viene chiuso.**TipoRichiesta == 1 è per la visualizzazione degli esami, TipoRichiesta == 2 per la prenotazione degli esami**.

```
if (richiesta_ricevuta.TipoRichiesta == 1)
{
    // Richiesta di visualizzare esami disponibili
    esami_disponibili(richiesta_ricevuta.esame, segreteria_connessione_socket);
}
else if (richiesta_ricevuta.TipoRichiesta == 2)
{
    // Gestione della prenotazione esame
    int socket_prenotazione_esame;
    struct sockaddr_in indirizzo_universita;
    struct Richiesta prenotazione_esame;
    prenotazione_esame.TipoRichiesta = 2; // TipoRichiesta 2 per prenotare un esame

    socket_prenotazione_esame = connessione_universita(&indirizzo_universita);

    printf("Connessione al server universitario riuscita\n");

    //Blocco funzioni per la prenotazione dell'esame
    esami_disponibili(richiesta_ricevuta.esame, segreteria_connessione_socket);
    riceviEsame(segreteria_connessione_socket, &richiesta_ricevuta.esame);

    MandaPrenotazioneEsame(socket_prenotazione_esame, richiesta_ricevuta.esame);
    RiceviMatricola(segreteria_connessione_socket, socket_prenotazione_esame);

    EsitoPrenotazione(segreteria_connessione_socket, socket_prenotazione_esame);
    MandaNumeroPrenotazione(segreteria_connessione_socket, socket_prenotazione_esame);

    printf("Prenotazione esame completata\n");

    close(socket_prenotazione_esame);
}
else
{
    fprintf(stderr, "Tipo di richiesta non valido\n");
}
```

3.4 Studente

Quando un client Studente viene eseguito, dopo aver deciso la sua richiesta alla segreteria, crea il socket e si connette mandando la richiesta per la visualizzazione degli esami o la prenotazione di una data di esame. Presenta due Struct per la richiesta e per gli esami. Le costanti definite sono:

- **MAX_RETRY**: Numero di tentativi possibili per connettersi.
- **SIMULA_TIMEOUT**: Costante usata per simulare un timeout.
- **MAX_WAIT_TIME**: Costante per la variabile tempoAttesa.
- **TIMEOUT_SIMULATO_SEC**: Costante usata per definire un tempo per la funzione sleep() in cui il sisterma simula un timeout.

E' importante evidenziare nella connessione alla segreteria come se lo studente non riesce a connettersi per 3 volte esso simulerà un timeout prima di riprovare a riconnettersi.

```c
int ConnessioneSegreteria(int *socket_studente, struct sockaddr_in *indirizzo_server_segreteria) {
    int retry_count = 0;

    // Configurazione dell'indirizzo del server della segreteria
    indirizzo_server_segreteria->sin_family = AF_INET;
    indirizzo_server_segreteria->sin_port = htons(2000);

    if (inet_pton(AF_INET, "127.0.0.1", &indirizzo_server_segreteria->sin_addr) <= 0) {
        perror("Errore inet_pton");
        exit(EXIT_FAILURE);
    }

    // Ciclo per tentare la connessione con massimo 3 tentativi
    while (retry_count < MAX_RETRY) {
        *socket_studente = Socket(AF_INET, SOCK_STREAM, 0);

        // Simula un timeout solo per i primi tentativi
        if (SIMULA_TIMEOUT && retry_count < MAX_RETRY) {
            simulaTimeout(); // Simula il timeout impostando errno
        }

        // Se non c'è errore o se errno non è impostato al valore di timeout, tenta la connessione
        if (errno != EAGAIN && errno != EWOULDBLOCK) {
            // Tentativo di connessione
            Connetti(*socket_studente, (struct sockaddr *)indirizzo_server_segreteria, sizeof(*indirizzo_server_segreteria));

            // Se Connetti non solleva errori, la connessione è riuscita
            printf("Connessione alla segreteria riuscita.\n");
            return *socket_studente; // Esci con successo
        }

        // Errore di connessione (simulato o reale), incrementa il contatore e ritenta
        printf("\nTentativo di riconnessione in corso (%d/%d)...\n", retry_count + 1, MAX_RETRY);
        retry_count++;
        sleep(2); // Pausa breve tra i tentativi
        close(*socket_studente); // Chiudi il socket prima di riprovare
    }
```

```
    // Se si superano i tentativi massimi, attende un tempo casuale e resetta il contatore
    attendiTempoCasuale();
    retry_count = 0;  // Resetta il contatore dei tentativi dopo l'attesa casuale

    // Ripeti il ciclo di tentativi dopo l'attesa casuale
    while (retry_count < MAX_RETRY) {
        *socket_studente = Socket(AF_INET, SOCK_STREAM, 0);

        // Tenta la connessione reale
        Connetti(*socket_studente, (struct sockaddr *)indirizzo_server_segreteria, sizeof(*indirizzo_server_segreteria));

        if (errno != EAGAIN && errno != EWOULDBLOCK) {
            printf("Connessione alla segreteria riuscita dopo attesa.\n");
            return *socket_studente;
        }

        printf("Tentativo di riconnessione in corso (%d/%d)...\n", retry_count + 1, MAX_RETRY);
        retry_count++;
        sleep(2);
        close(*socket_studente);
    }

    // Se il ciclo termina senza successo
    fprintf(stderr, "Impossibile connettersi alla segreteria dopo ripetuti tentativi.\n");
    exit(EXIT_FAILURE);
}
```

attendiTempoCasuale e **simulaTimeout** sono funzioni richiamate nella ConnessioneSegreteria per generare i tempi di attesa tra una connessione e l'altra e la simulazione del timeout finiti i tentativi disponibili per connettersi

```
void attendiTempoCasuale() {
    // Imposta il seme per la generazione casuale basato sul tempo corrente
    srand(time(NULL));
    int tempoAttesa = rand() % MAX_WAIT_TIME + 1; // Genera un numero tra 1 e MAX_WAIT_TIME
    printf("\nAttesa di %d secondi prima di ritentare la connessione...\n", tempoAttesa);
    sleep(tempoAttesa); // Pausa per il tempo generato
}

void simulaTimeout() {
    if (SIMULA_TIMEOUT) {
        printf("Simulazione di timeout: attesa di %d secondi...\n", TIMEOUT_SIMULATO_SEC);
        sleep(TIMEOUT_SIMULATO_SEC); // Simula un'attesa che rappresenta il timeout
        errno = EAGAIN; // Simula un errore di timeout impostando errno
    }
}
```

Queste funzioni qui sotto gestiscono **la lettura e la scrittura di dati necessari per la visualizzazione e la prenotazione degli esami esami**

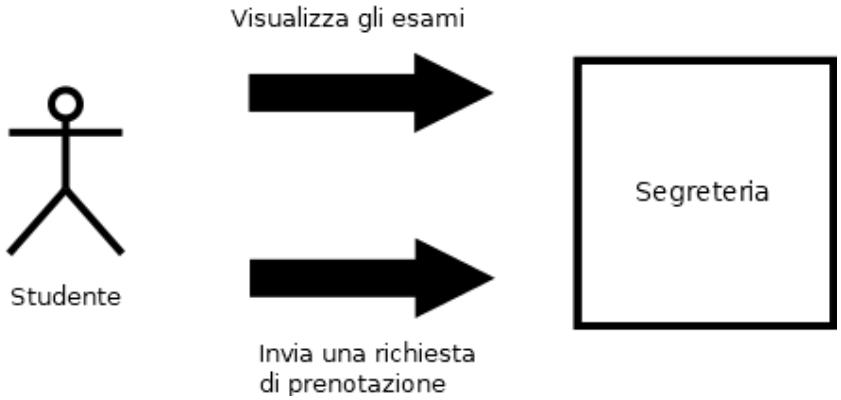
```c
void riceviListaEsami(int socket_studente, int numero_esami, struct Esame *esameCercato)
{
    if (read(socket_studente, esameCercato, sizeof(struct Esame) * numero_esami) != sizeof(struct Esame) * numero_esami)
    {
        perror("Errore ricezione lista esami");
        exit(1);
    }
}

int conto_esami(int socket_studente, int *numero_esami)
{
    if (read(socket_studente, numero_esami, sizeof(*numero_esami)) != sizeof(*numero_esami))
    {
        perror("Errore ricezione numero esami");
        exit(1);
    }
    return *numero_esami;
}

int numeroPrenotazione(int socket_studente, int *numero_prenotazione)
{
    if (read(socket_studente, numero_prenotazione, sizeof(*numero_prenotazione)) != sizeof(*numero_prenotazione))
    {
        perror("Errore ricezione numero prenotazione");
        exit(1);
    }
    return *numero_prenotazione;
}

void mandaMatricola(int socket_studente, char *matricola)
{
    size_t length = strlen(matricola);
    if (write(socket_studente, matricola, length) != length)
    {
        perror("Errore invio matricola");
        exit(1);
    }
}

void mandaEsamePrenotazione(int socket_studente, struct Esame *esameDaPrenotare)
{
    // Invia l'intera struttura Esame
    if (write(socket_studente, esameDaPrenotare, sizeof(struct Esame)) != sizeof(struct Esame))
    {
        perror("Errore invio esame prenotazione");
        exit(1);
    }
}
```

4 Diagrammi

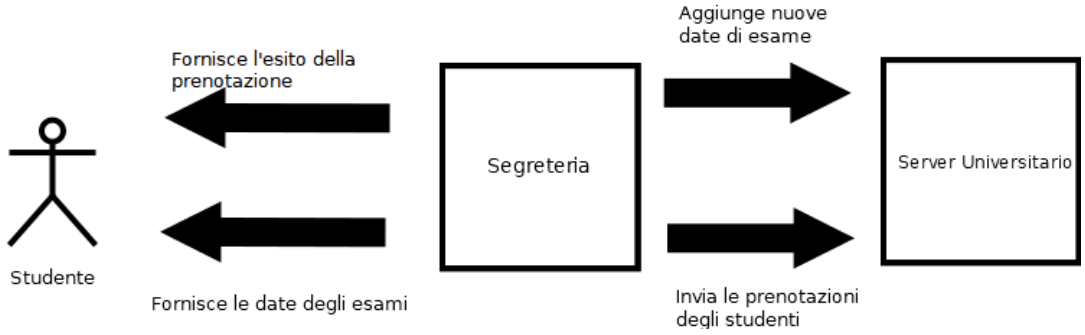
4.1 Diagramma dei casi d'uso

I diagrammi dei casi d'uso servono per rappresentare le interazioni che un utente svolge rispetto al sistema, di seguito mostriamo le varie interazioni che il sistema svolge.

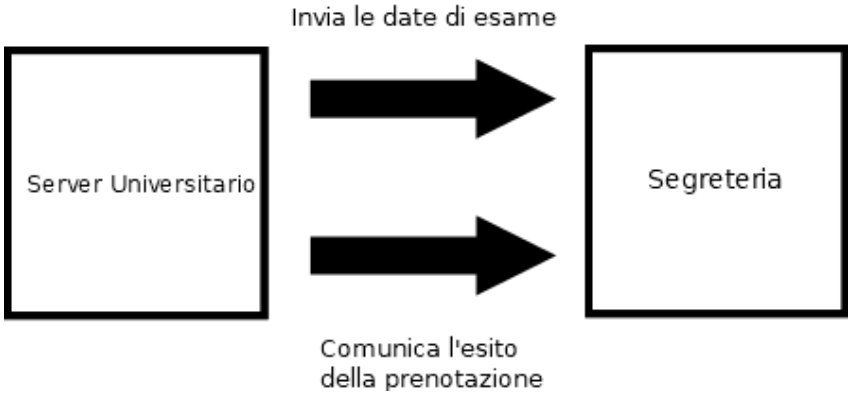
Qui sono rappresentate le **operazioni dello studente**, la ricerca di un esame tramite nome e la prenotazione di esso.



Qui vediamo le **operazioni della segreteria**, che interagisce sia con lo studente fornendogli le date degli esami e mandandogli l'esito della prenotazione con il numero della prenotazione, sia con lo studente universitario quando aggiunge un nuovo esame e quando manda le prenotazioni degli studenti.

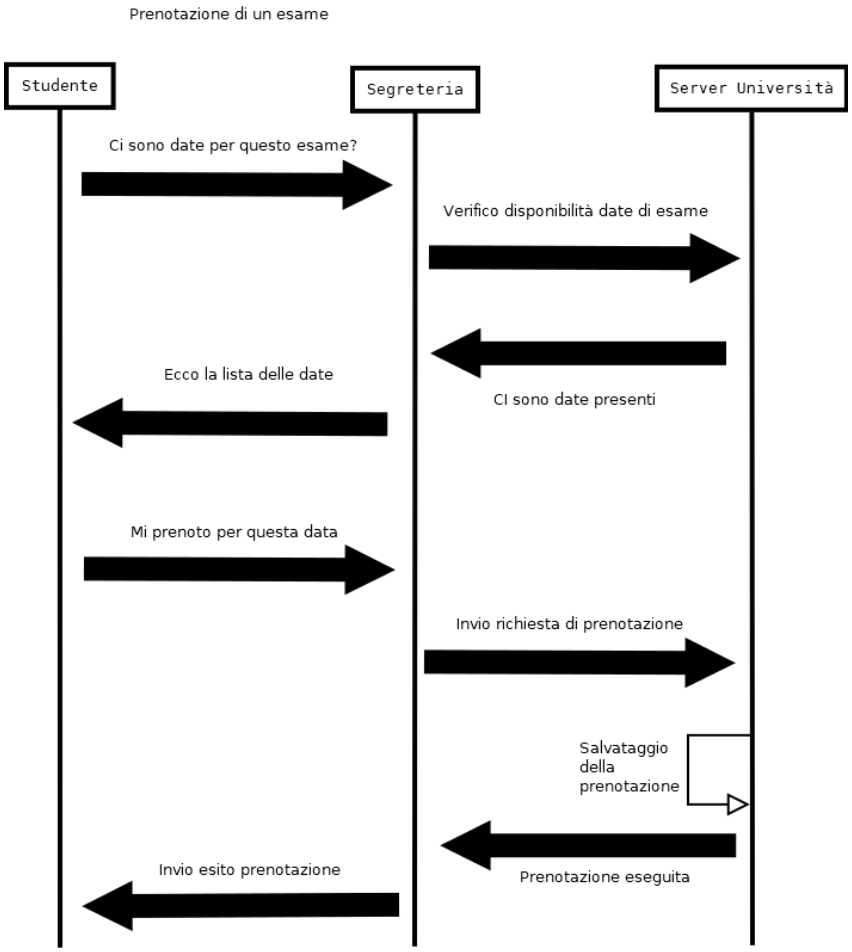


Anche se il server universitario non è un vero e proprio utente del sistema, trovavamo necessario mostrare le **interazioni con la segreteria**.



4.2 Diagrammi delle sequenze

I diagrammi delle sequenze servono a mostrare il flusso di azioni che vengono svolte per un particolare compito tra . Di seguito abbiamo il diagramma delle sequenze della prenotazione di un esame da parte di Studente.



5 Istruzioni per l'utilizzo

5.1 Compilazione del progetto

Per poter compilare il progetto bisogna posizionare il terminale della cartella in cui è contenuto il progetto, nominata **Progetto** in questo caso:

- **Server Universitario:**
  - gcc -o server server_universitario.c wrapper.c
- **Segreteria:**
  - gcc -o segreteria segreteria.c wrapper.c
- **Studente:**
  - gcc -o studente studente.c wrapper.c

Nella cartella del progetto sono anche già presenti dei file compilati ma consigliamo di compilare i file nuovamente per evitare problemi

5.2 Compilazione del progetto

Per la compilazione del progetto digitare sul terminale:

- ./server
- ./segreteria
- ,/studente

Ricordate di seguire quest'ordine per un corretto funzionamento.