Reti di Calcolatori – 2023/2024

Prof. Emanuel Di Nardo

Gestionale Università

Codice gruppo: n4etdb7aa8

0124002384 Nicola Ponticelli

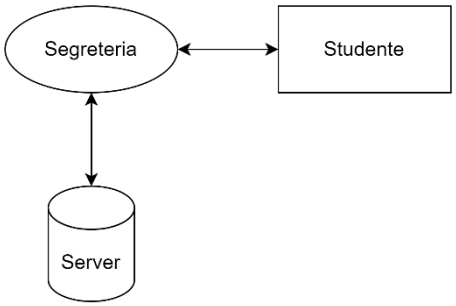
0124002436 Francesco Peluso

Sommario

Descrizione progetto

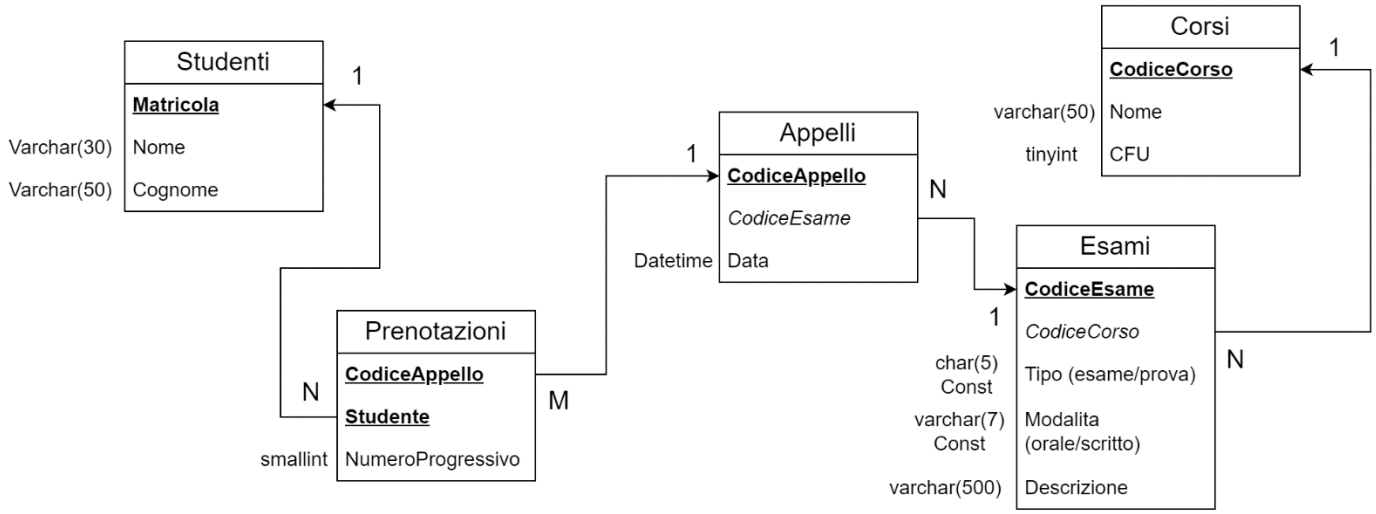
Si vuole scrivere un applicazione client/server concorrente per gestire gli esami universitari. Le tre entità sono: segreteria, studente e server universitario. La **segreteria** inserisce gli appelli sul server dell'università, inoltra la richiesta di prenotazione degli studenti al server universitario e infine fornisce allo studente le date degli esami disponibili per l'esame scelto dallo studente. Lo **studente** chiede alla segreteria se ci sono appelli disponibili per un corso ed infine invia una richiesta di prenotazione di un esame alla segreteria dalla quale riceve il numero progressivo della prenotazione. Infine il **server universitario** riceve l'aggiunta di nuovi esami e riceve la prenotazione di un esame.

Schema architetturale

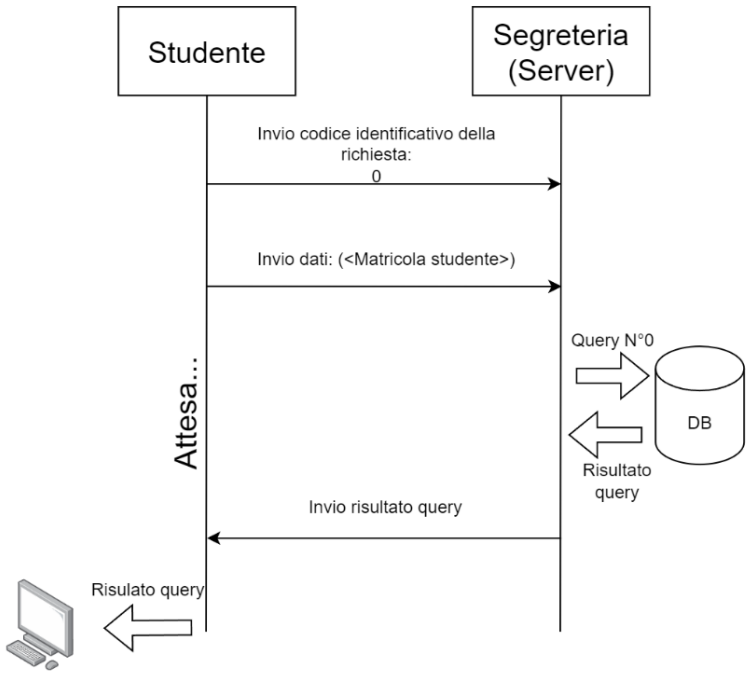


Lo schema architetturale si basa su tre entità: il **server dell'università**, la **segreteria** e lo **studente**. Lo studente non si interfaccia direttamente con il server dell'università, ma lo fa attraverso la segreteria. Quindi quest'ultima altro non è che un **tramite** che inoltra le richieste dello studente al server universitario e poi inoltra i dati restituiti dal server allo studente. La segreteria è coinvolta attivamente solo nell'inserimento di un nuovo appello ed esame.

Data la presenza di dati strutturati si è scelto di utilizzare il database relazionale SQL, in particolare SQLite che possiede un'ottima libreria compatibile con il linguaggio C. Si è scelto di utilizzare il linguaggio C++ principalmente per due motivi: è orientato ad oggetti e le lezioni di laboratorio del corso sono in C.



Operazione 0: autenticazione studente

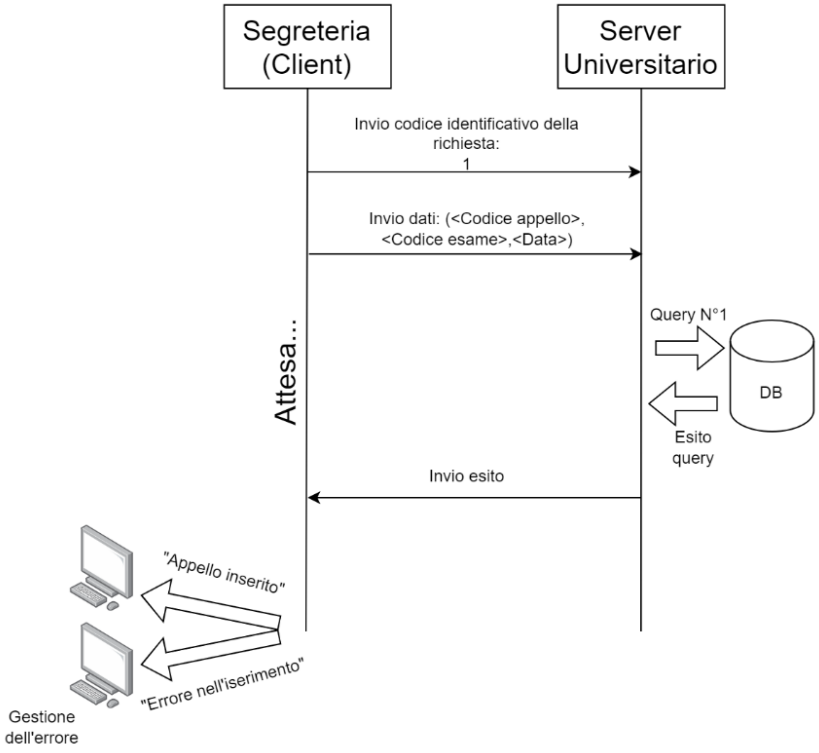


L'operazione inizia con lo studente che inserisce la propria matricola. Alla segreteria sono inviati il codice identificativo dell'operazione insieme alla matricola dello studente. Poi la segreteria inoltra i dati ricevuti al server dell'università che verifica se lo studente esiste nel database. Infine il server restituisce il risultato (se lo studente è autenticato oppure no) alla segreteria che, a sua volta, lo inoltra allo studente.

Il server universitario controlla la presenza della matricola nel database con la seguente query:

```
SELECT * FROM student
WHERE Matricola = '<id studente>'
```

Operazione 1: inserimento nuovo appello dalla segreteria

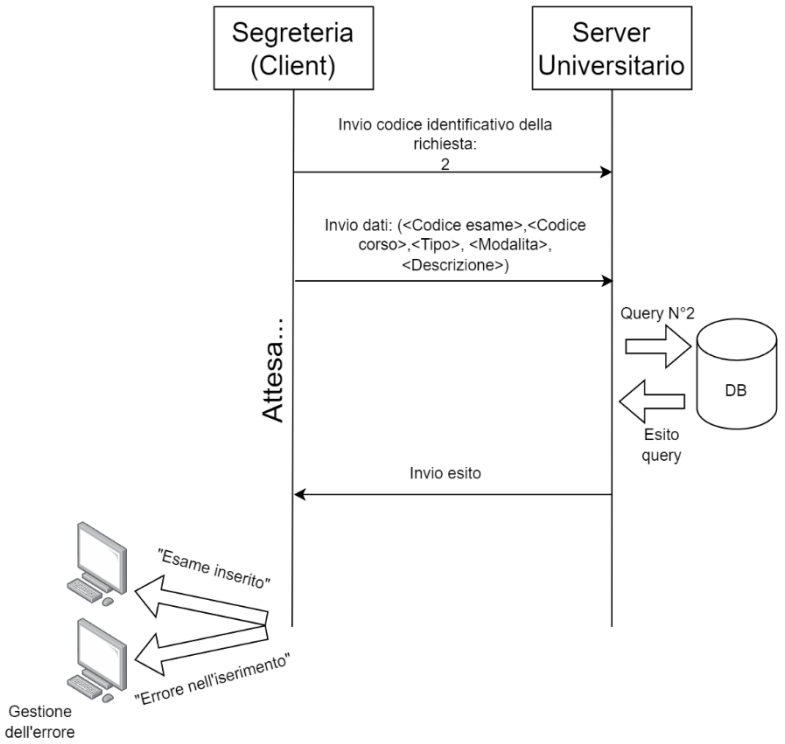


L'operazione inizia con la segreteria che sceglie l'operazione. Al server universitario sono inviati il codice identificativo dell'operazione insieme ai dati da inserire dell'appello. Poi il server dell'università inserisce i dati nel database e infine restituisce alla segreteria l'esito dell'operazione (se è andata a buon fine oppure no).

Il server universitario inserisce i valori ricevuti all'interno del database con la seguente query:

```
INSERT INTO appelli (CodiceAppello,
CodiceEsame, Data)
VALUES ('<id>', '<id>',
'<datetime>')
```

Operazione 2: inserimento nuovo esame dalla segreteria

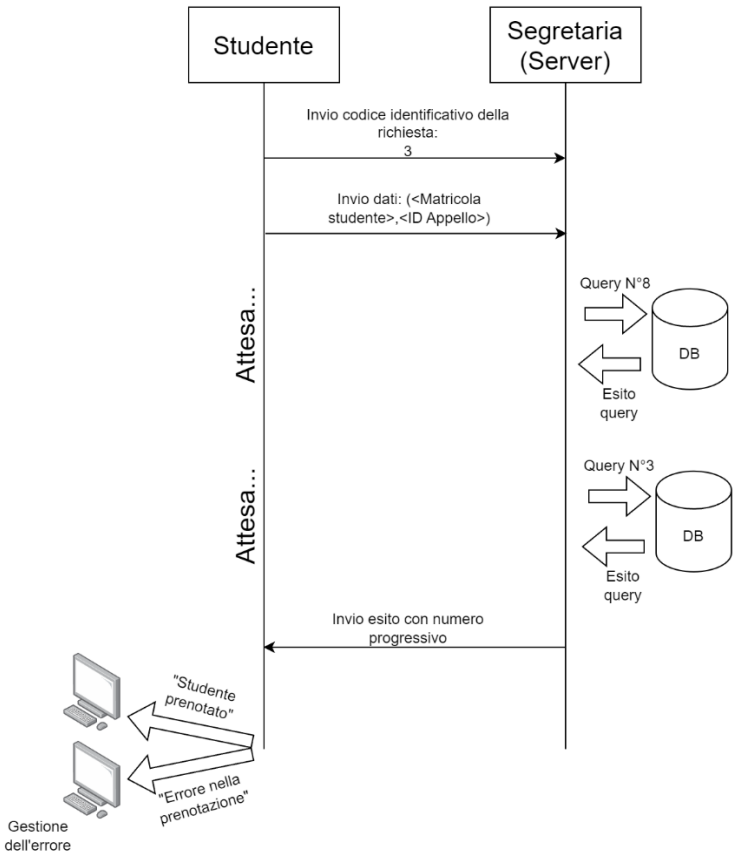


L'operazione inizia con la segreteria che sceglie l'operazione. Al server universitario sono inviati il codice identificativo dell'operazione insieme ai dati da inserire dell'esame. Il server universitario riceve i dati e li inserisce nel database. Infine invia alla segreteria l'esito dell'inserimento (se è andato a buon fine oppure no).

Il server universitario inserisce i valori ricevuti all'interno del database con la seguente query:

```
INSERT INTO esami (CodiceEsame, CodiceCorso, Tipo, Modalita, Descrizione)
VALUES ('<id>', '<id>', '<tipo>', '<modalita>', '<descrizione>')
```

Operazione 3: inserimento nuova prenotazione dello studente



L'operazione inizia con lo studente che sceglie l'operazione. Alla segreteria sono inviati il codice identificativo dell'operazione insieme ai dati: la matricola dello studente e il codice dell'appello al quale prenotarsi. Dopodiché questi dati sono inoltrati al server dell'università che prima genera il numero progressivo del prenotato e poi salva l'appello nel database. Infine il server universitario invia alla segreteria il numero progressivo dello studente che si è appena prenotato la quale, a sua volta, lo inoltra allo studente.

Per generare il numero progressivo, il server universitario deve prima ottenere il numero di prenotati a quell'appello con la seguente query:

```
SELECT count(*) AS MaxNumero
FROM prenotazioni
WHERE CodiceAppello = '<id appello>'
```

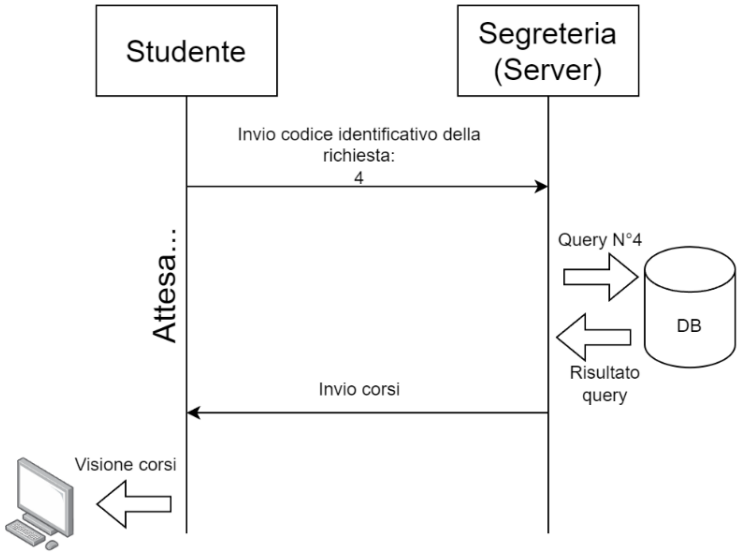
Infine i dati sono inseriti e salvati nel database con la seguente query:

```
INSERT INTO prenotazioni (CodiceAppello, Studente, NumeroProgressivo)
VALUES ('<id>', '<id>', '<smallint>')
```

Richiesta visione appelli disponibili di un determinato esame

La visione degli appelli disponibili di un esame comprende tre sotto operazioni che vengono eseguite in modo indipendente: la visione dei corsi, la visione degli esami e infine la visione degli appelli. Quindi lo studente ottiene prima i corsi disponibili e poi ne seleziona uno, poi ottiene gli esami disponibili del corso selezionato e seleziona un esame e infine ottiene la lista degli appelli disponibili per l'esame selezionato in precedenza.

Operazione 4: Visione corsi

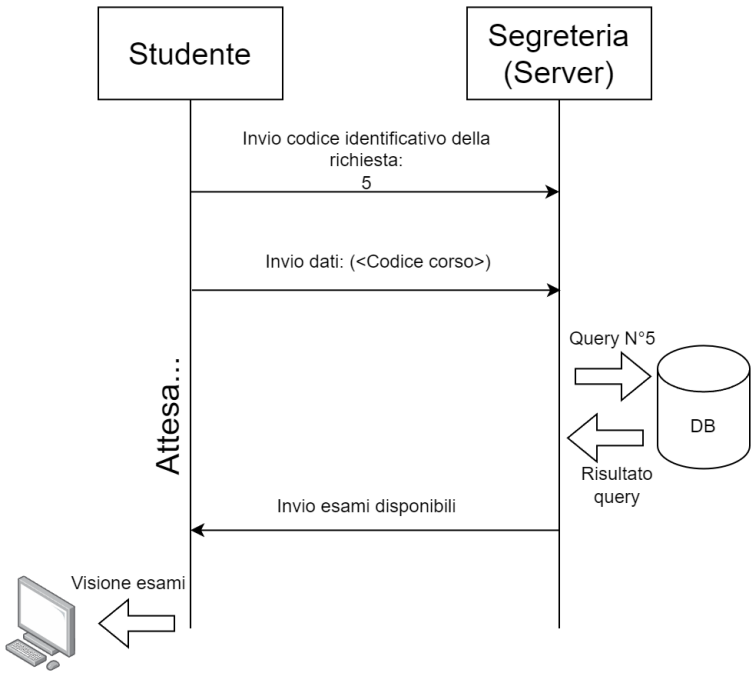


L'operazione inizia con lo studente che sceglie l'operazione. Alla segreteria è inviato il codice identificativo dell'operazione che viene inoltrato al server universitario. Infine il server dell'università preleva i dati dal database, li invia alla segreteria e, a sua volta, la segreteria li inoltra allo studente.

Per prelevare i corsi dal database, il server universitario esegue la seguente query:

```
SELECT CodiceCorso AS Codice, Nome, CFU
FROM corsi
```

Operazione 5: Visione esami di un determinato corso

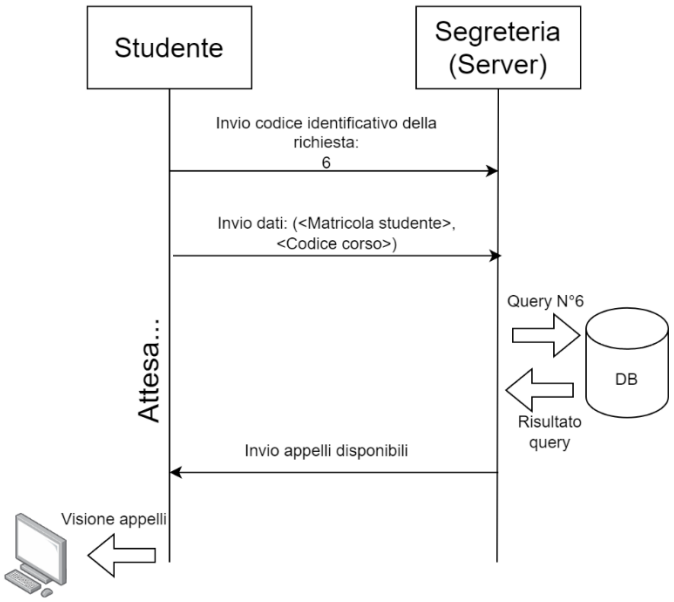


L'operazione inizia con lo studente che sceglie l'operazione. Alla segreteria sono inviati il codice identificativo dell'operazione insieme al codice del corso selezionato. Poi la segreteria inoltra tutto al server universitario che preleva i dati dal database. Infine il server dell'università invia alla segreteria i dati che, a sua volta, li invia allo studente.

Per prelevare gli esami dal database si utilizza la seguente query:

```
SELECT CodiceEsame AS Codice, Tipo,
Modalita, Descrizione
FROM esami
WHERE CodiceCorso = '<id corso>'
```

Operazione 6: Visione appelli disponibili di un determinato esame

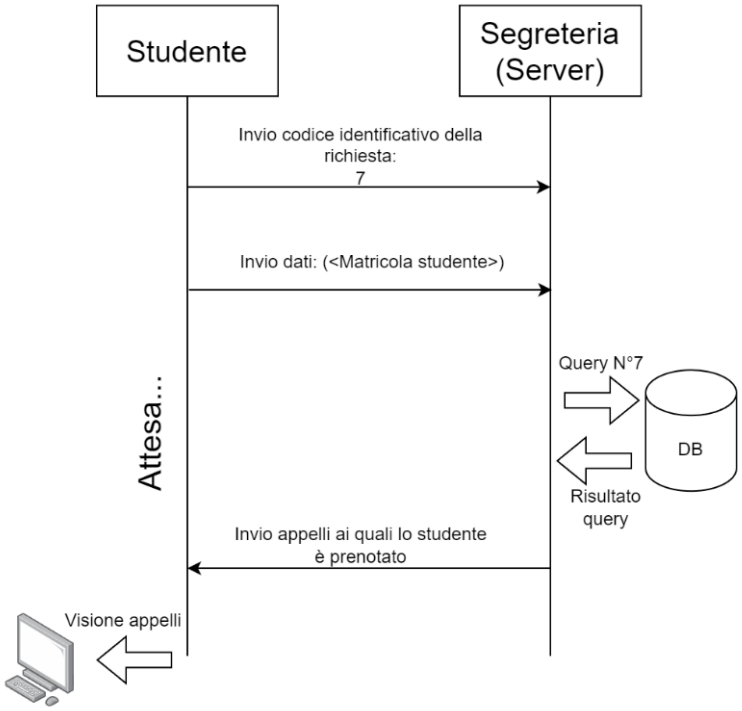


L'operazione inizia con lo studente che sceglie l'operazione. Alla segreteria sono inviati il codice identificativo dell'operazione insieme ai dati: la matricola dello studente e il codice dell'esame selezionato. La segreteria inoltra la richiesta al server universitario che preleva i dati dal database. Infine il server dell'università invia i dati prelevati alla segreteria che, a sua volta, li inoltra allo studente.

Per prelevare gli esami dal database si utilizza la seguente query:

```
SELECT appelli.CodiceAppello AS Codice, appelli.Data, corsi.Nome
FROM appelli
	JOIN esami ON appelli.CodiceEsame = esami.CodiceEsame
	JOIN corsi ON esami.CodiceCorso = corsi.CodiceCorso
WHERE
	NOT EXISTS (
		SELECT prenotazioni.CodiceAppello
		FROM prenotazioni
		WHERE prenotazioni.Studente = '<id studente>'
			AND prenotazioni.CodiceAppello = appelli.CodiceAppello
	)
	AND esami.CodiceEsame = '<id esame>'
```

Operazione 7: Visione appelli prenotati



L'operazione inizia con lo studente che sceglie l'operazione. Alla segreteria sono inviati il codice identificativo dell'operazione insieme alla matricola dello studente. La segreteria inoltra la richiesta al server dell'università che preleva i dati dal database. Infine il server universitario restituisce i dati alla segreteria che, a sua volta, li inoltra allo studente.

I dati sono prelevati dal database con la seguente query:

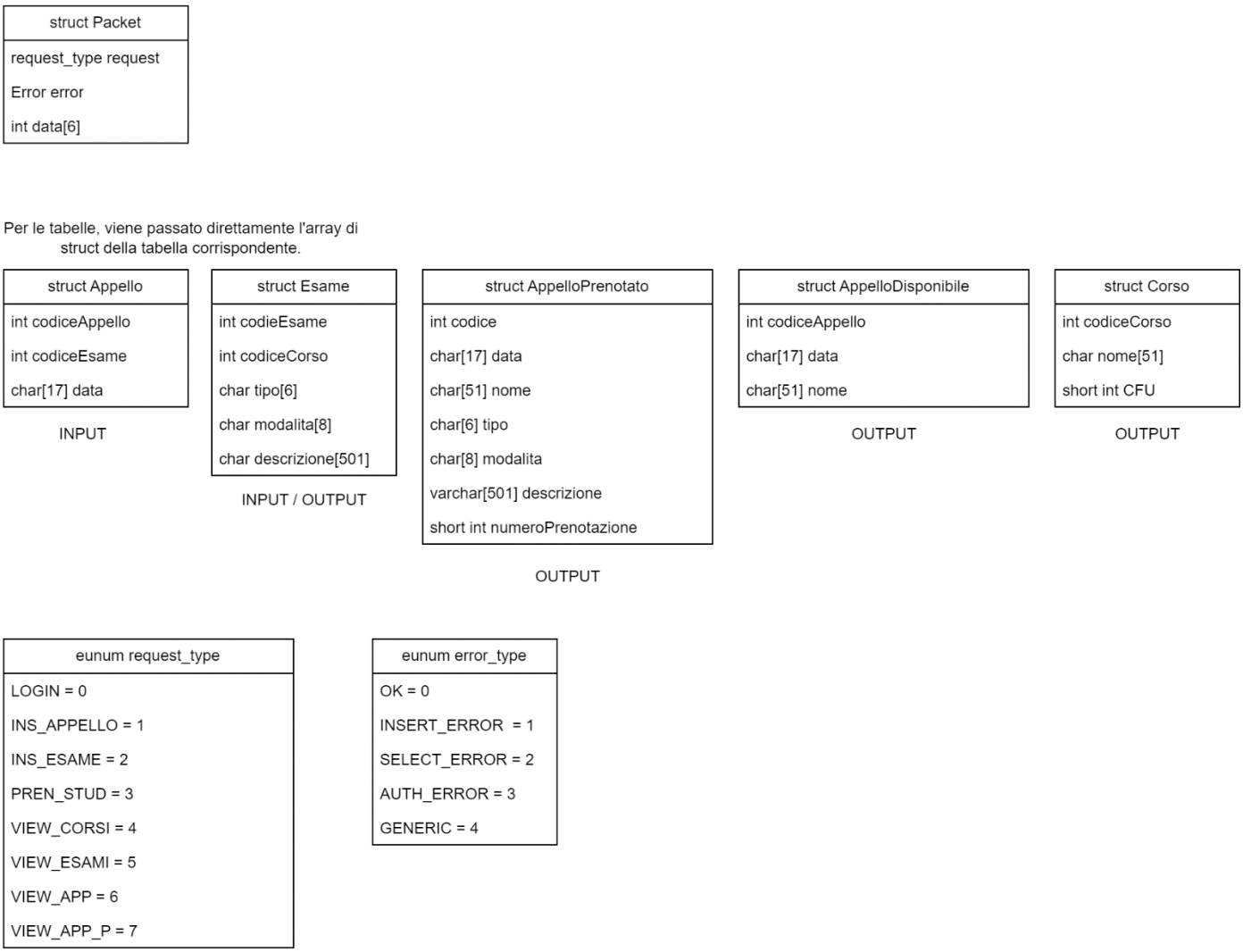
```
SELECT prenotazioni.CodiceAppello AS Codice, appelli.Data, corsi.Nome, esami.Tipo,
esami.Modalita, esami.Descrizione, Prenotazioni. NumeroProgressivo AS NPrenotazione
FROM prenotazioni
    JOIN studenti ON prenotazioni.Studente = studenti.Matricola
    JOIN appelli ON prenotazioni.CodiceAppello = appelli.CodiceAppello
    JOIN esami ON appelli.CodiceEsame = esami.CodiceEsame
    JOIN corsi ON corsi.CodiceCorso = esami.CodiceCorso
WHERE prenotazioni.Studente = '<id studente>'
```

Dettagli implementativi dei client/server

Classi, librerie e Strutture dati

Al fine di poter implementare le operazioni, sono state individuate e modellate classi e strutture dati per la gestione delle richieste

Strutture dati



La prima struttura dati è il *packet*.
Con questa struttura dati si identifica il **pacchetto di richiesta**. I suoi campi sono i seguenti:

- *request*, un'enumerazione (*request_type*) che identifica il tipo di operazione che si vuole eseguire mediante un codice
- *error*, un oggetto della classe *Error* che indica il verificarsi di un errore durante l'esecuzione delle richieste
- *data*, un array di 4 elementi che contiene i dati da inviare nella richiesta e/o nella risposta (matricola dello studente, corso selezionato, esame selezionato, appello a cui lo studente vuole prenotarsi, numero di record restituito dalle query e numero progressivo di prenotazione)

Ogni volta che si vuole eseguire un'operazione, si invia un *packet* inizializzando il campo *request*: esso sarà il discriminante secondo il quale i server soddisferanno le richieste.
Ad ogni risposta dal server, il client controlla l'errore: se non è stato prodotto alcun errore continua con l'operazione, altrimenti gestisce l'errore.

Le strutture dati *Appello*, *Esame*, *AppelloPrenotato*, *AppelloDisponibile* e *Corso* indicano le tabelle del DB. Ogni struttura ha gli stessi campi delle tabelle del DB, sia in tipo che in dimensione.
La struttura *Appello* è una struttura di **input**, cioè rappresenta il record che dovrà essere inserito attraverso una query di inserimento.
La struttura *Esame* è una struttura di **input** e **output**, cioè rappresenta il record che dovrà essere inserito attraverso una query di inserimento e il risultato della SELECT per ottenere gli esami.
Le strutture *AppelloPrenotato*, *AppelloDisponibile* e *Corso* sono strutture di **output**, cioè rappresentano i record dei risultati delle SELECT per la visione degli appelli.

L'enumerazione *request_type* definisce e raggruppa gli identificativi delle richieste delle operazioni. Essi sono:

- *LOGIN* = 0 → Identifica la richiesta di login per lo studente
- *INS_APPELLO* = 1 → Identifica la richiesta di inserimento di un nuovo appello da parte della segreteria
- *INS_ESAME* = 2 → Identifica la richiesta di inserimento di un nuovo esame da parte della segreteria
- *PREN_STUD* = 3 → Identifica la richiesta di prenotazione dello studente ad un appello da parte della segreteria
- *VIEW_CORSI* = 4 → Identifica la richiesta di visione dei corsi
- *VIEW_ESAMI* = 5 → Identifica la richiesta di visione degli esami per un corso
- *VIEW_APP* = 6 → Identifica la richiesta di visione degli appelli disponibili a cui lo studente non è prenotato
- *VIEW_APP_P* = 7 → Identifica la richiesta di visione degli appelli a cui lo studente è prenotato

L'enumerazione *error_type* definisce e raggruppa gli identificativi degli errori che possono verificarsi durante l'esecuzione:

- *OK* = 0 → Non si è verificato nessun errore
- *INSERT_ERROR* = 1 → Errore nel procedimento di inserimento, in particolare durante query SQL
- *SELECT_ERROR* = 2 → Errore nel procedimento di ottenimento delle tabelle, in particolare durante query SQL
- *AUTH_ERROR* = 3 → Errore durante l'autenticazione dello studente
- *GENERIC* = 4 → Errore generico

Classi

DBMS
- db: sqlite3 * - error_connection_db: int
- createMapResults(table_result: vector< map<string,string> >*, statement: sqlite3_stmt*, error: Error* = nullptr): void - executeQuery(sql: string, error_code: ErrorType, error: Error*, table_result: vector< map <string,string> >* = nullptr): void + DBMS(db_name: string, error: Error*=nullptr) + ~DBMS() + select(sql: string, error: Error*=nullptr): vector<map<string,string>>* + insert(sql: string, error: Error*=nullptr): void

La classe *DBMS* permette di interfacciarsi con il database SQLite vero e proprio, in particolare, date le operazioni previste dal sistema tale classe permette di effettuare solo SELECT ed INSERT.

Attributi:

- *db*, rappresenta il puntatore al file del database
- *error_connection_db*, rappresenta il valore intero dell'errore di connessione al DB

Metodi:

- *createMapResults*, metodo che assembla riga per riga la tabella restituita dalla query eseguita
- *executeQuery*, metodo che wrappa le funzioni della libreria sqlite3 per eseguire, effettivamente, la query passata come parametro
- *select*, metodo che esegue una SELECT sul DB
- *insert*, metodo che esegue una INSERT sul DB

Error
- code: error_type - title: char[50] - description: char[500]
+ Error() + Error(error_code: ErrorType) + Error (error_code: ErrorType, title: string, description: string) + ~Error() + setAll(error_code: ErrorType, title: string, description: string): void + setCode(error_code: ErrorType): void + getCode(): ErrorType + setTitle(title: string): void + getTitle(): string + setDescription(description: string): void + getDescription(): string + printError(): void

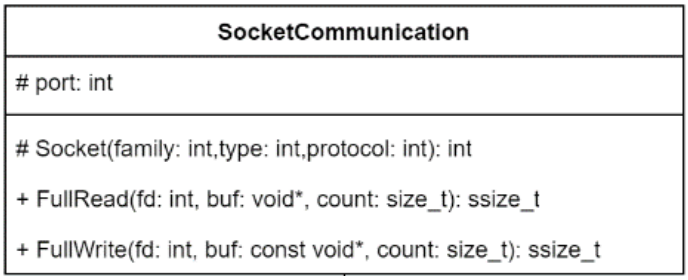
La classe *Error* rappresenta l'errore che può verificarsi durante l'esecuzione.

Attributi:

- *code*, rappresenta il codice dell'errore che si è verificato
- *title*, rappresenta, in formato stringa, il nome dell'errore
- *desription*, rappresenta, in formato stringa, la descrizione dell'errore

Metodi:

- *printError*, stampa tutti i dati relativi all'errore
- setters e getters per settare e per ottenere i relativi dati



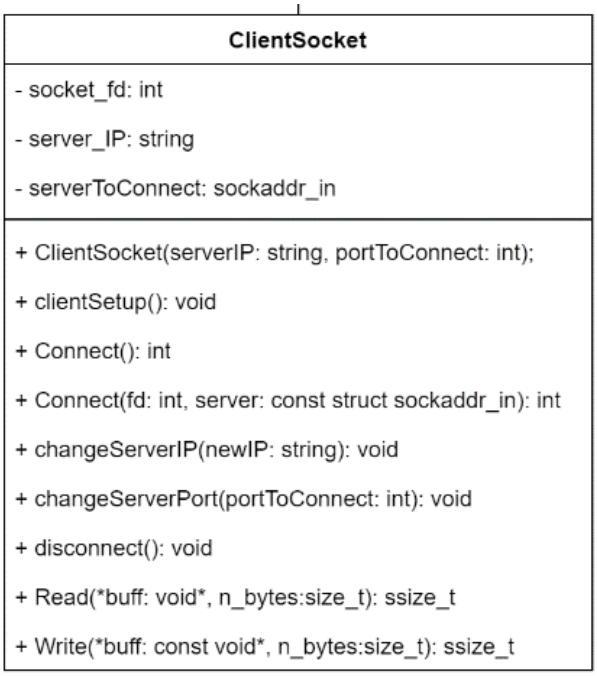
La classe *SocketCommunication* rappresenta la comunicazione attraverso un socket definendo i metodi per l'invio e la ricezione dei dati su di essa.

Attributi:

- *port*, rappresenta la porta della socket

Metodi:

- *Socket*, metodo che wrappa la chiamata di sistema *socket()* per creare un socket e ottenere il file descriptor per usarla
- *FullRead*, funzione che wrappa la lettura completa del buffer della chiamata di sistema *read()*
- *FullWrite*, funzione che wrappa la scrittura completa del buffer della chiamata di sistema *write()*



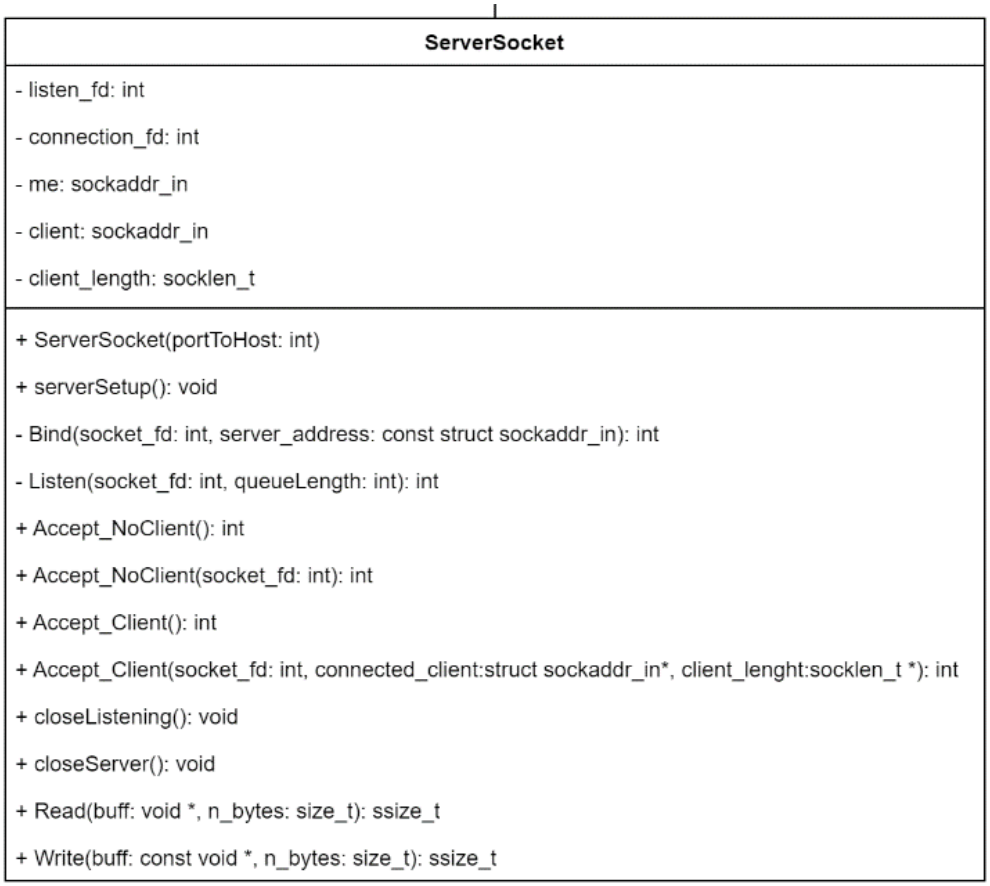
La classe *ClientSocket* eredita da *SocketCommunication* e rappresenta la socket dal lato client.

Attributi:

- *socket_fd*, file descriptor del socket per la comunicazione
- *server_IP*, IP del server a cui il client vuole connettersi
- *serverToConnect*, struttura che rappresenta il server a cui ci si vuole connettere
- *port*, ereditato da *SocketCommunication*, rappresenta la porta a cui il client vuole connettersi

Metodi:

- *clientSetup*, procedura di inizializzazione della comunicazione: permette di ottenere il file descriptor della scoket, imposta i dati del server e rende pronto il client per la comunicazione
- *Connect*, procedura che wrappa la chiamata di sistema *connect()* per connettersi al server
- *changeServerIP*, procedura che permette di cambiare l'IP del server a cui ci si vuole connettere, impostandolo in formato network e aggiornandolo nella struttura *serverToConnect*
- *changeServerPort*, procedura che permette di cambiare la porta del server a cui ci si vuole connettere, impostandolo in formato network e aggiornandolo nella struttura *serverToConnect*
- *disconnect*, procedura che chiude il file descriptor della socket
- *Read* / *Write*, procedure che wrappano le *FullRead* e *FullWrite* sulla connessione stabilita dal client



La classe *ServerSocket* eredita da *SocketCommunication* e rappresenta la socket dal lato server.

Attributi:

- *listen_fd*, file descriptor di ascolto della socket
- *connection_fd*, file descriptor della socket una volta accettata la connessione dal client
- *me*, struttura che rappresenta il server stesso per l'impostazione della comunicazione
- *client,* struttura che rappresenta il client la cui connessione viene accettata
- *client_length*, struttura che rappresenta la lunghezza del client

Metodi:

- *serverSetup*, procedura di inizializzazione della comunicazione: permette di ottenere il file descriptor della scoket, imposta i dati del server e rende pronto il server per l'ascolto e l'accettazione
- *Bind*, procedura che wrappa la chiamata di sistema *bind()*
- *Listen*, procedura che wrappa la chiamata di sistema *listen()*
- *Accept_NoClient*, procedura che wrappa la chiamata di sistema *accept()* salvandosi in *connection_fd* il file descriptor per la comunicazione con il client. Non vengono salvate le informazioni sul client che si connette
- *Accept_Client*, analoga alla precedente con la differenza che si salva le informazioni del client connesso
- *changePort* cambia la porta su cui il server è hostato impostandola in formato network
- *closeListening*, procedura che chiude il file descriptor di ascolto
- *closeServer*, procedura che chiude il file descriptor della socket e della connessione con il client
- *Read* / *Write*, procedure che wrappano le *FullRead* e *FullWrite* sulla connessione stabilita dal server

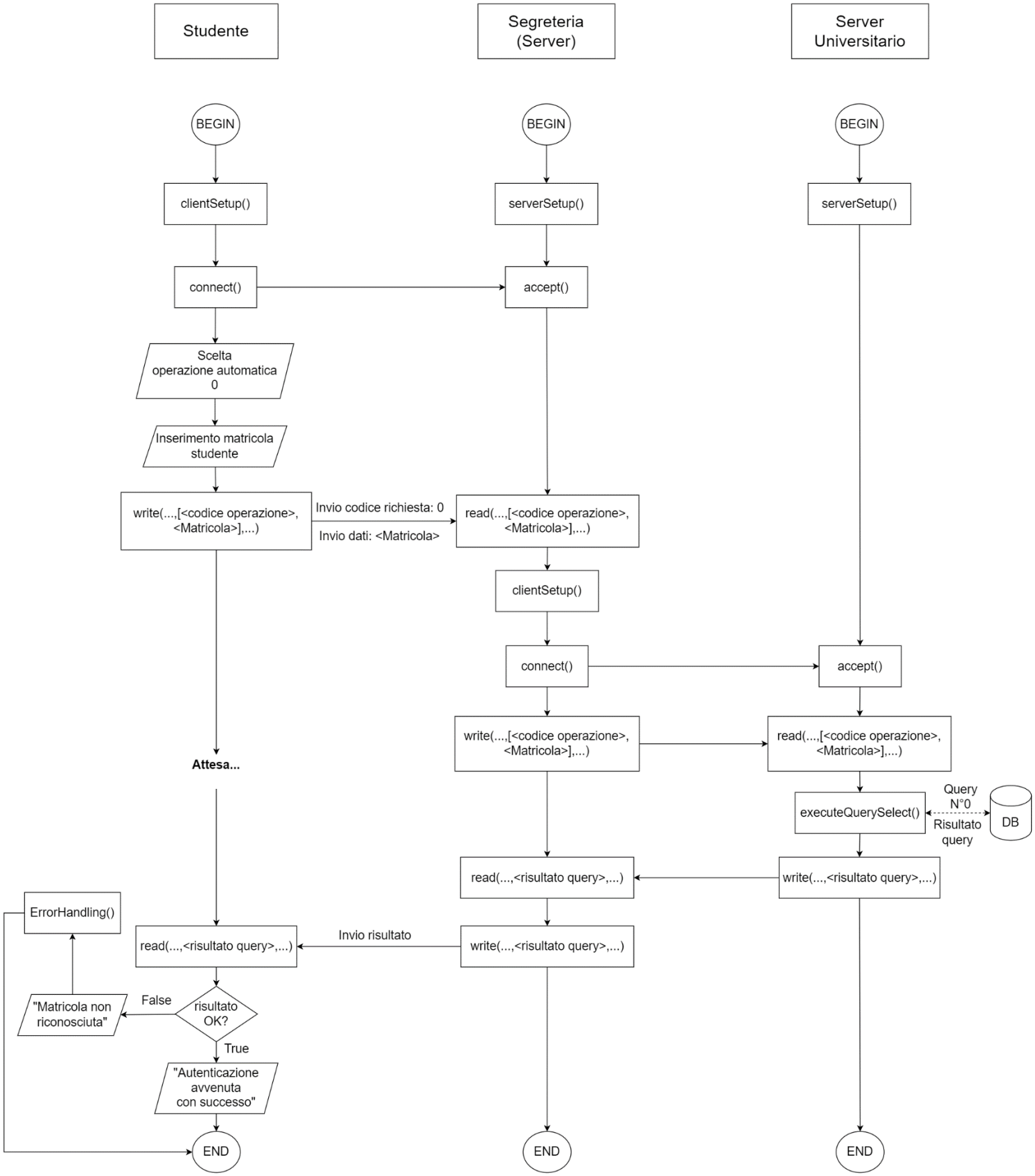
Operazioni

In questa sezione della documentazione vengono trattate le operazioni precedentemente descritte **specificandone** (sempre in modo astratto) la **sequenza effettiva di passi**.
Le direzioni delle frecce rappresentano la **direzione del flusso delle operazioni**.
Le frecce **trasversali** indicano la **comunicazione** tra client/server.
In tale implementazione si è preferito far eseguire le query al DB **sempre e solo al server universitario**, di conseguenza una richiesta che parte dallo studente verso la segreteria, viene inoltrata al server universitario per poter eseguire le query; ne deriva che i risultati/esiti delle query sul server verranno inoltrati alla segreteria la quale, a sua volta, li inoltrerà allo studente.
In linea generale:

- Ogni operazione inizia con l'impostazione della comunicazione socket attraverso il metodo *clientSetup()*, per il lato client, e con l'impostazione dell'ascolto attraverso il metodo *serverSetup()*, per il lato server.
- Per iniziare un'operazione viene inviato un *packet* al cui interno il campo *request* conterrà il codice dell'operazione richiesta

Per gestire la richiesta, i server controllano tale campo *request*.

Operazione 0: Autenticazione Studente



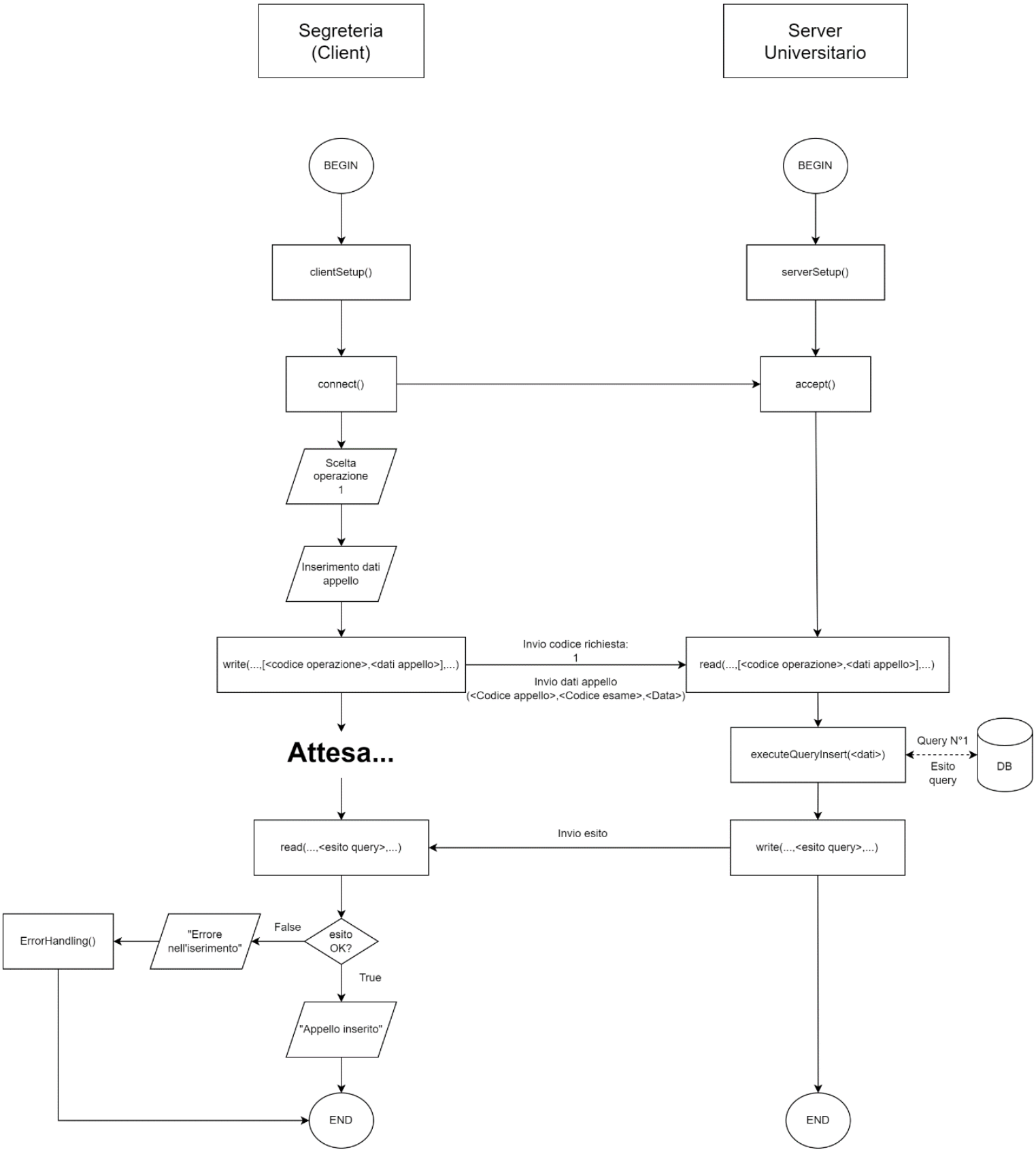
La procedura di autenticazione viene eseguita appena il processo *studente* è avviato.

Tale procedura inizia con la richiesta di autenticazione (identificata dal codice di richiesta 0) da parte dello *studente*: esso, dopo aver fornito in input la propria matricola, invia (attraverso una *write()*) un pacchetto di richiesta al server della segreteria con le seguenti informazioni:

- Codice della richiesta → *request* = *LOGIN* (= 0)
- Matricola dello studente

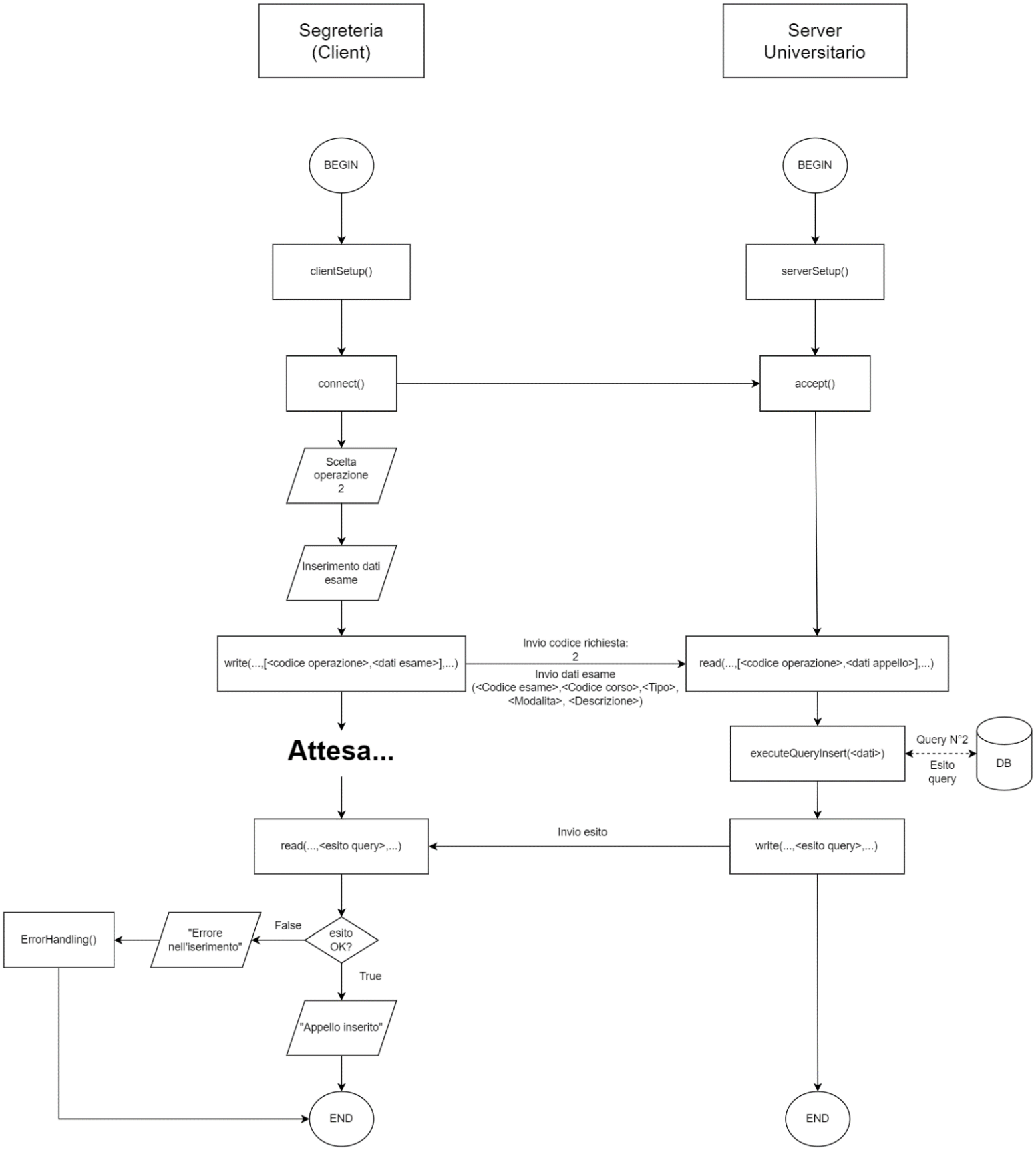
Il server della segreteria attraverso una *read()* riceve il pacchetto di richiesta.
Da questo punto in poi, il server della segreteria assume anche il ruolo di client. Così facendo anch'esso invocherà il metodo *clientSetup()*, si connetterà al server universitario e inoltrerà la richiesta al server universitario inviando la stessa coppia di dati.
Il server universitario, accettata la connessione dal client (segreteria) e ricevuta la richiesta, esegue la query N°0 e ne invia i risultati al server universitario.
Il server della segreteria invierà a sua volta il risultato della query allo studente.
Lo studente, infine, potrà procedere se autenticato oppure, in caso contrario, verrà stampato un messaggio di errore.

Operazione 1: Inserimento nuovo appello



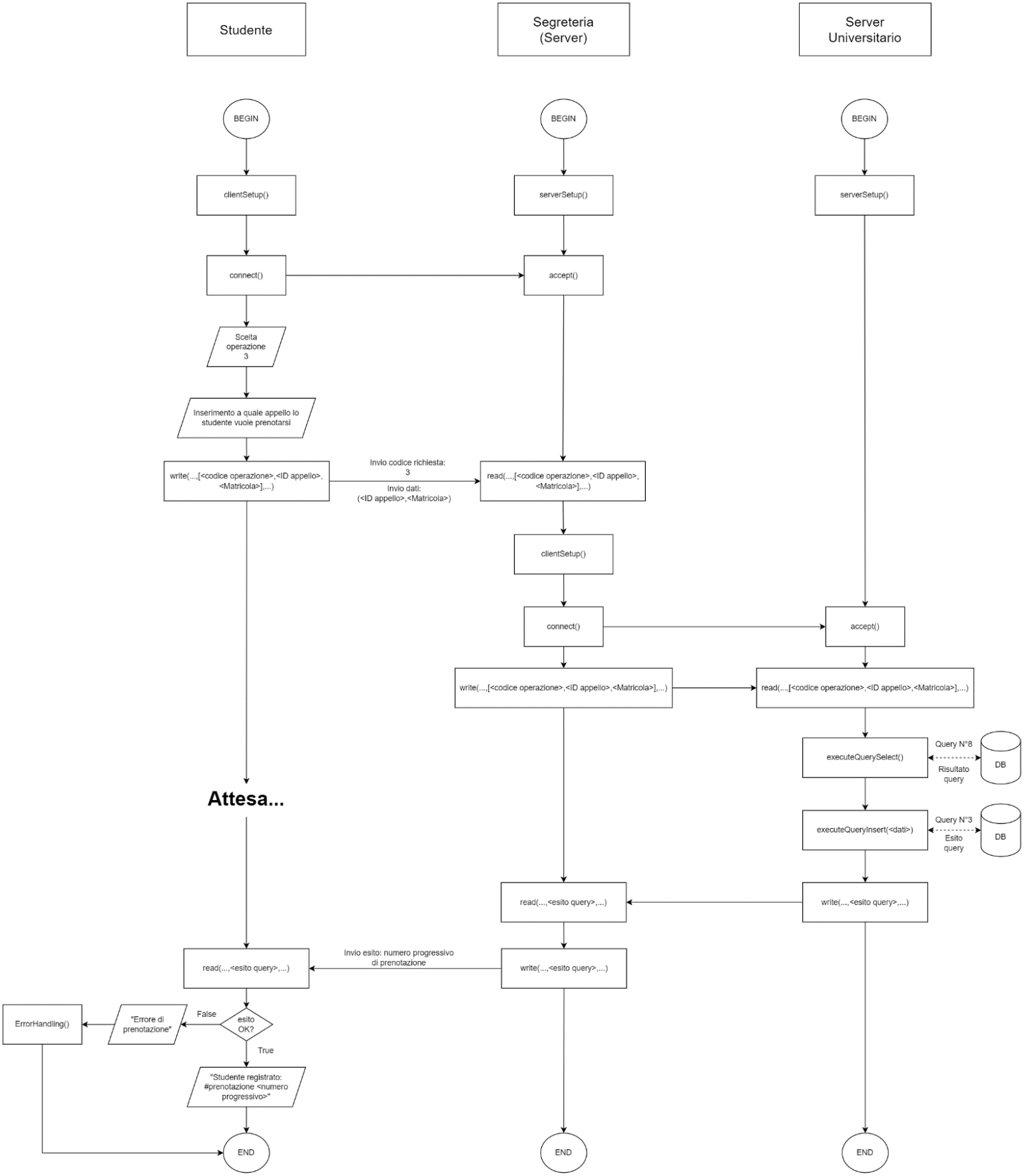
La procedura inizia il con client della segreteria che tenta di connettersi al server universitario.
Una volta che la connessione ha avuto esito positivo, il client della segreteria invia del pacchetto di richiesta con il campo *request* impostato a *INS_APPELLO* (= 1) al server universitario.
Immediatamente dopo l'invio di tale richiesta, sono inviati al server universitario anche i dati dell'appello da inserire; dopodiché il client della segreteria si mette in attesa della risposta dal server universitario.
Il server universitario provvede ad eseguire la query N°1 e il suo esito verrà inviato al client della segreteria.
Quest'ultimo elaborerà tale risultato.

Operazione 2: Inserimento nuovo esame



La procedura inizia il con client della segreteria che tenta di connettersi al server universitario.
Una volta che la connessione ha avuto esito positivo, il client della segreteria invia del pacchetto di richiesta con il campo *request* impostato a *INS_ESAME* (= 2) al server universitario.
Immediatamente dopo l'invio di tale richiesta, sono inviati al server universitario anche i dati dell'esame da inserire; dopodiché il client della segreteria si mette in attesa della risposta dal server universitario.
Il server universitario provvede ad eseguire la query N°2 e il suo esito verrà inviato al client della segreteria.
Quest'ultimo elaborerà tale risultato.

Operazione 3: Inserimento nuova prenotazione dello studente



La procedura inizia il con lo studente che cerca di connettersi al server della segreteria.
Se la connessione ha successo, lo studente inserisce il codice dell'appello a cui intende prenotarsi e invia il pacchetto di richiesta con le seguenti informazioni:

- Codice della richiesta → *request = PREN_STUD* (= 3)
- Matricola dello studente
- Appello selezionato

Il server della segreteria riceve la richiesta e assume il ruolo di client verso il server universitario (ne consegue l'utilizzo del metodo *clientSetup()*) e, dopo essersi connesso con successo al server universitario, invia il pacchetto di richiesta con le stesse informazioni al server universitario.
Il server universitario eseguirà due query:

- La query N°8 per ottenere il numero di studenti prenotati all'appello scelto inizialmente dallo studente
- La query N°3 per tentare di inserire la prenotazione all'interno del DB

Se l'ultima query avrà successo, verrà inviata come risposta alla richiesta della segreteria il numero progressivo di prenotazione dello studente.
La segreteria provvederà ad inoltrare allo studente il numero progressivo se l'inserimento è andato a buon fine.

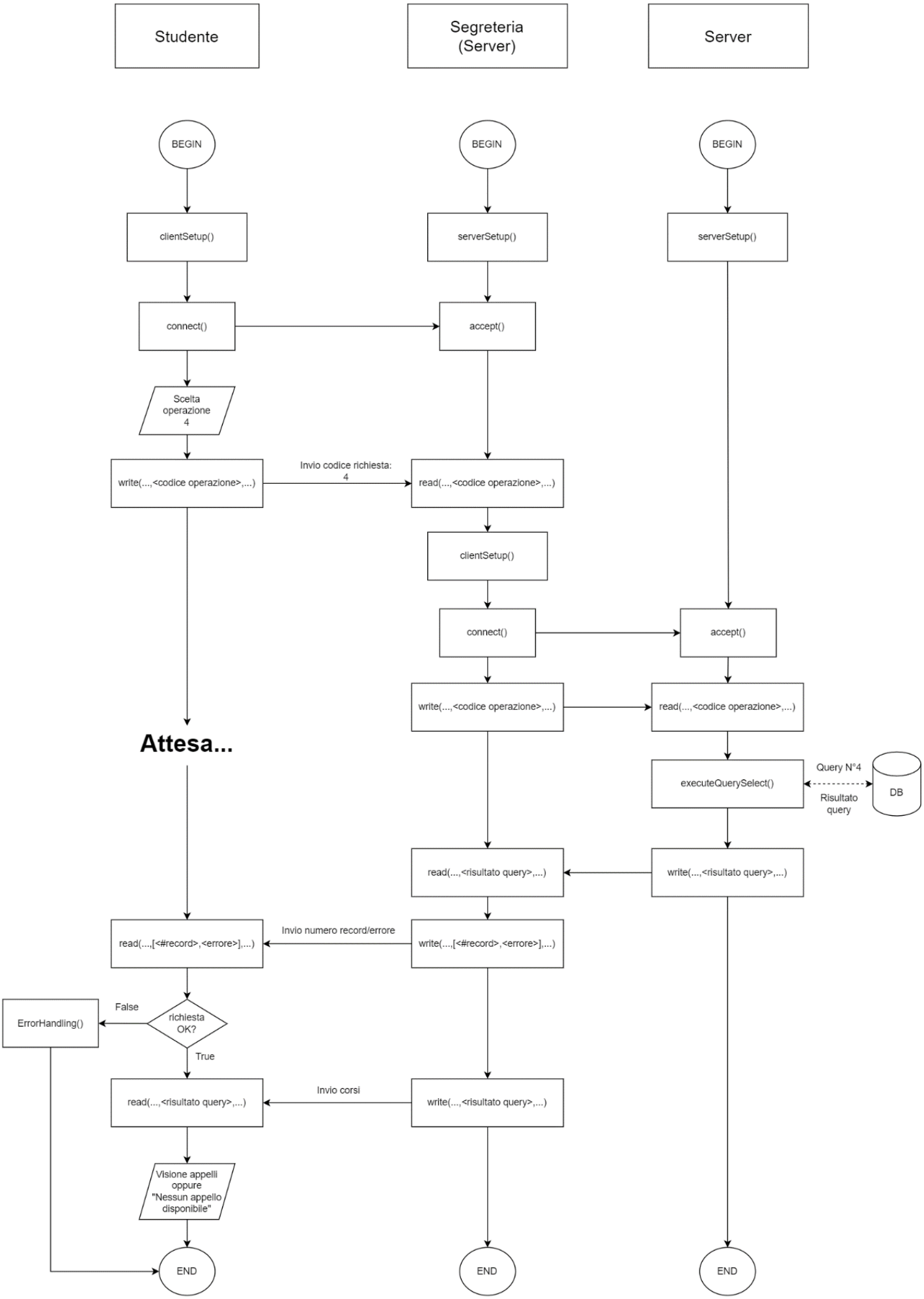
Richiesta di visione degli appelli disponibili

La richiesta di visione degli appelli disponibili per un corso è costituita da 3 macro operazioni:

- Visione corsi
- Visione esami
- Visione appelli

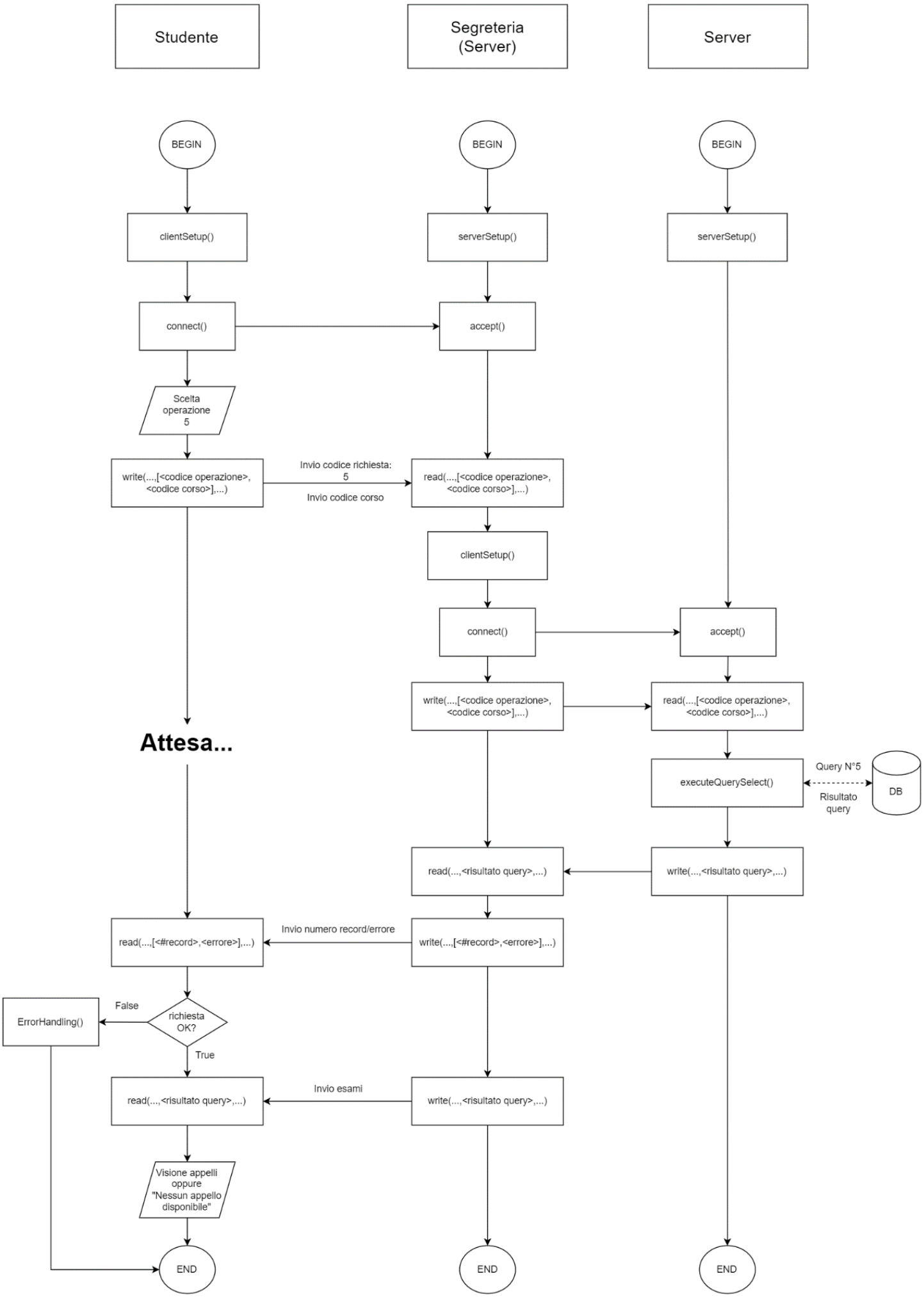
La richiesta di visione degli appelli inizia con la visione dei corsi; l'utente dovrà selezionare il corso di cui vuole visionare gli appelli.
Una volta scelto il corso, si passa alla visione degli esami per quel corso. Qualora fossero presenti degli esami per quel corso, allora si potranno visionare, in caso contrario la procedura si interrompe e si dovrà ricominciare da capo.
Una volta scelto l'esame, si passa alla visione degli appelli per quell'esame ai quali, però, lo studente non è già prenotato. Qualora fossero presenti degli appelli per quell'esame, saranno stampati a schermo, mentre in caso contrario la procedura termina e bisogna ricominciare da capo.

Operazione 4: Richiesta visione corsi



Se la connessione tra *studente* e *segreteria* ha successo, lo studente invia il pacchetto di richiesta con il campo *request* pari a *VIEW_CORSI* (= 4).
Il server della segreteria deve quindi contattare il server universitario per ottenere i corsi e per farlo assume il ruolo di client (ne consegue l'utilizzo del metodo *clientSetup()*). Dopo essersi connesso con successo al *server universitario*, la *segreteria* invia il pacchetto di richiesta con le stesse informazioni al server universitario.
Il server universitario eseguirà la query N°4 e se tale query avrà successo verrà inviato il risultato alla *segreteria* la quale lo inoltrerà allo *studente* che potrà visionare i corsi.
In questo caso, al fine di capire se la query ha prodotto risultati o no (e quindi capire se ci sono corsi), si controlla prima il numero di record ottenuto dalla query: se quest'ultimo è 0 allora non ci saranno corsi, in caso contrario verranno stampati a schermo.

Operazione 5: Richiesta visione esami di un determinato corso

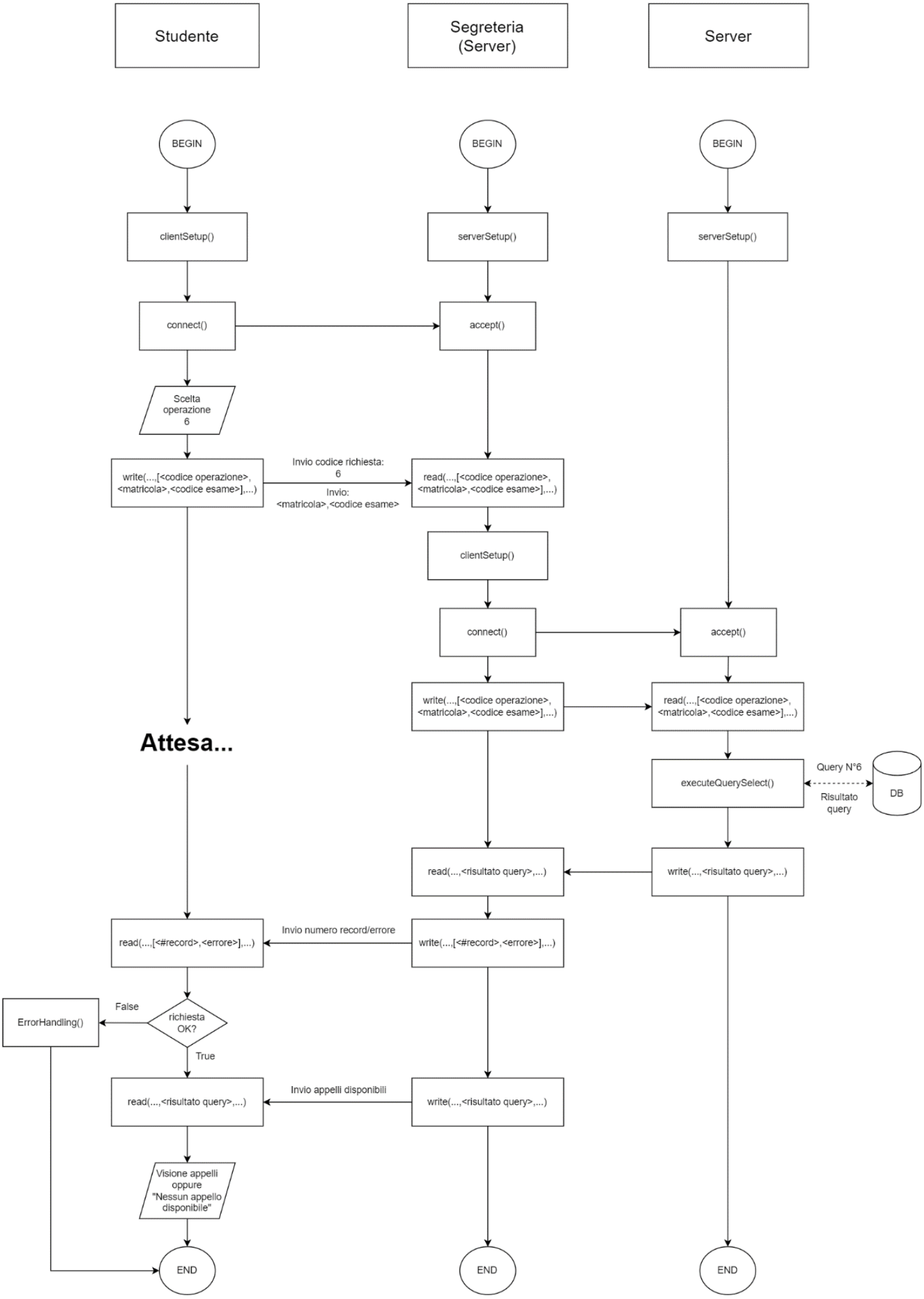


Se la connessione tra *studente* e *segreteria* ha successo, lo studente invia il pacchetto di richiesta con le seguenti informazioni:

- Codice della richiesta → *request =VIEW_ESAMI* (= 5)
- Codice del corso

Il server della segreteria deve quindi contattare il server universitario per ottenere gli esami e per farlo assume il ruolo di client (ne consegue l'utilizzo del metodo *clientSetup()*). Dopo essersi connesso con successo al *server universitario*, la *segreteria* invia il pacchetto di richiesta con le stesse informazioni al server universitario.
Il server universitario eseguirà la query N°5 e se tale query avrà successo verrà inviato il risultato alla *segreteria* la quale lo inoltrerà a sua volta allo *studente* che potrà visionare gli esami del corso scelto.
In questo caso, al fine di capire se la query ha prodotto risultati o no (e quindi capire se ci sono esami), si controlla prima il numero di record ottenuto dalla query: se quest'ultimo è 0 allora non ci saranno esami per il corso selezionato, in caso contrario verranno stampati a schermo.

Operazione 6: Richiesta visione appelli disponibili

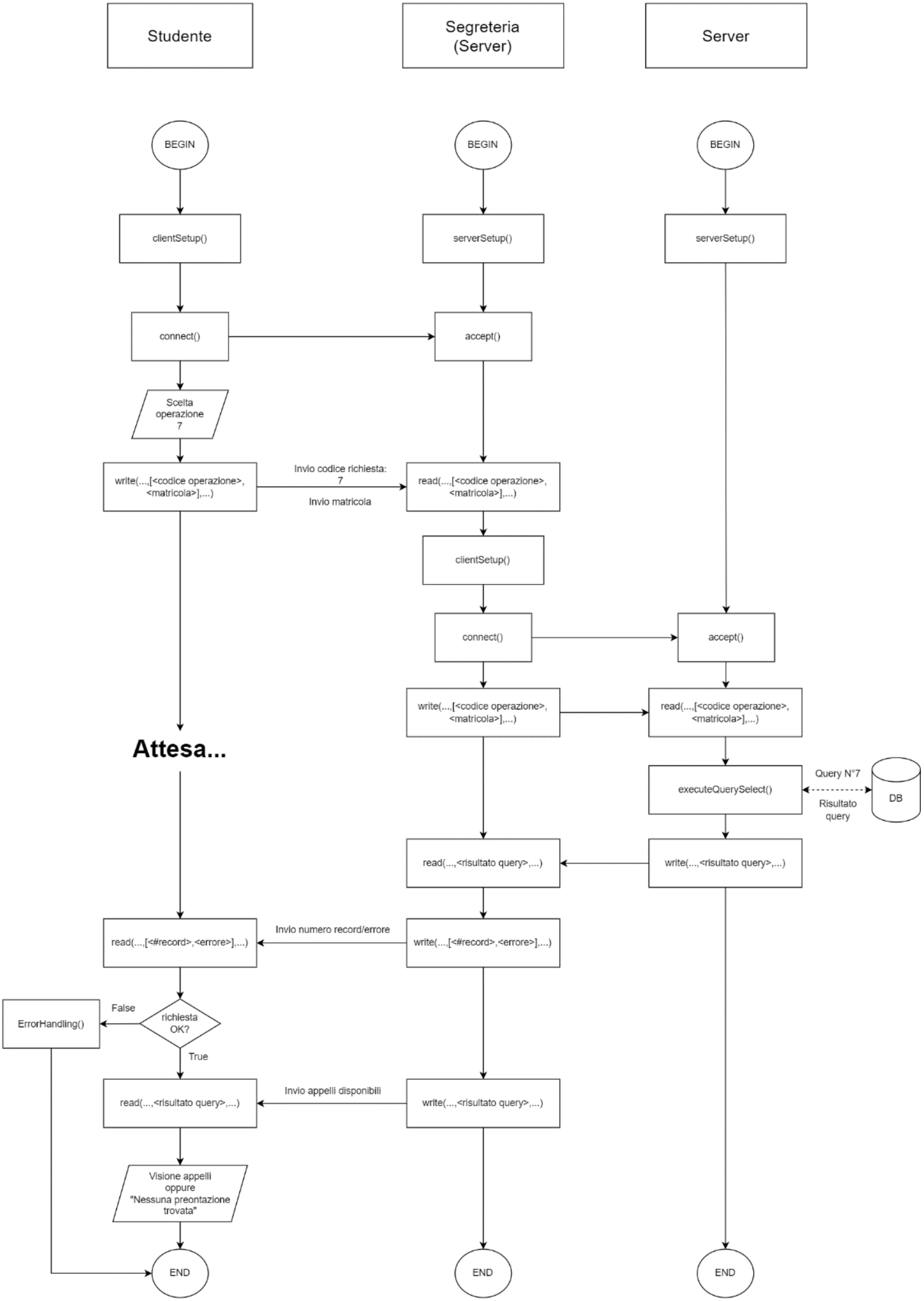


Se la connessione tra *studente* e *segreteria* ha successo, lo studente invia il pacchetto di richiesta con le seguenti informazioni:

- Codice della richiesta → *request =VIEW_APP* (= 6)
- Matricola dello studente
- Codice dell'esame

Il server della segreteria deve quindi contattare il server universitario e per farlo assume il ruolo di client (ne consegue l'utilizzo del metodo *clientSetup()*) e, dopo essersi connesso con successo al server universitario, invia il pacchetto di richiesta con le stesse informazioni al server universitario.
Il server universitario eseguirà la query N°6 e se tale query avrà successo verrà inviato il risultato alla *segreteria* la quale lo inoltrerà allo *studente* che potrà visionare gli appelli disponibili ai quali non si è prenotato.
In questo caso, al fine di capire se la query ha prodotto risultati o no (e quindi capire se ci sono appelli disponibili per lo studente), si controlla prima il numero di record ottenuto dalla query: se quest'ultimo è 0 allora non ci saranno appelli disponibili, in caso contrario verranno stampati a schermo.

Operazione 7: Richiesta visione appelli prenotati dello studente



Se la connessione tra *studente* e *segreteria* ha successo, lo studente invia il pacchetto di richiesta con le seguenti informazioni:

- Codice della richiesta → *request =VIEW_APP_P* (= 7)
- Matricola dello studente

Il server della segreteria deve quindi contattare il server universitario per ottenere gli appelli a cui lo studente è prenotato e per farlo assume il ruolo di client (ne consegue l'utilizzo del metodo *clientSetup()*).
Dopo essersi connesso con successo al server universitario, la *segreteria* invia il pacchetto di richiesta con le stesse informazioni al server universitario.
Il server universitario eseguirà la query N°7 e se tale query avrà successo verrà inviato il risultato alla *segreteria* la quale lo inoltrerà allo *studente* che potrà visionare gli appelli disponibili ai quali non si è prenotato.
In questo caso, al fine di capire se la query ha prodotto risultati o no (e quindi capire se ci sono appelli a lo studente è prenotato), si controlla prima il numero di record ottenuto dalla query: se quest'ultimo è 0 allora non ci saranno appelli prenotati, in caso contrario verranno stampati a schermo.

Parti rilevanti del codice

Il codice è suddiviso in **quattro programmi main**, ognuno dei quali possiede il proprio eseguibile:

- *segreteria_server.cpp*: rappresenta quella parte di segreteria che ha la funzione di inoltrare le richieste dello studente al server universitario;
- *segreteria_client.cpp*: rappresenta quella parte di segreteria che ha la funzione di inserire i nuovi esami e i nuovi corsi;
- *studente.cpp*: rappresenta lo studente che deve effettuare le varie operazioni passando per la segreteria (in particolare per *segreteria_server*);
- *server_uni.cpp*: rappresenta il server universitario che si interfaccia con il database e che elabora le richieste dello studente e della segreteria.

File SocketCommunication

```
class SocketCommunication{
    protected:
        int port;

        int Socket(int family,int type,int protocol);
    public:
        SocketCommunication(int port);
        ssize_t FullRead(int fd, void *buf, size_t count);
        ssize_t FullWrite(int fd, const void *buf, size_t count);
};
```

La classe *SocketCommunication* rappresenta la comunicazione attraverso una socket definendo i metodi per l'invio e la ricezione dei dati su di essa. L'unico attributo è *port* che rappresenta la porta della socket.

Metodi:

- *SocketCommunication()*: costruttore che prende in input una porta e la imposta come porta della socket;
- *Socket()*: metodo che wrappa la system call *socket()* per creare una socket e ottenere il file descriptor per usarla. I parametri di input sono gli stessi della system call;
- *FullRead()*: funzione che wrappa la lettura completa del buffer della chiamata di sistema *read()*. I parametri di input sono gli stessi della system call;
- *FullWrite()*: funzione che wrappa la scrittura completa del buffer della chiamata di sistema *write()*. I parametri di input sono gli stessi della system call;

```
#define UNI_SERVER_PORT 10000
#define SEGRETERIA_SERVER_PORT 20000
#define QUEUE_LENGTH 1024
```

All'interno del file sono definite varie macro che sono utilizzate sia dalle classi che dai programmi main:

- *UNI_SERVER_PORT*: rappresenta la porta del server universitario sulla quale comunicare;
- *SEGRETERIA_SERVER_PORT*: rappresenta la porta del *server_segreteria* sulla quale *studente* e *server_uni* comunicano;
- *QUEUE_LENGTH*: rappresenta il numero di client massimi in coda per essere accettati dal server.

File ClientSocket

```
class ClientSocket: protected SocketCommunication{
    private:
        std::string server_IP;
        int socket_fd;
        sockaddr_in serverToConnect;

    public:
        ClientSocket(std::string serverIP,int portToConnect);
        void clientSetup();
        int Connect();
        int Connect(int fd,const struct sockaddr_in server);
        void changeServerIP(std::string newIP);
        void changeServerPort(int portToConnect);
        void disconnect();
        ssize_t Read(void *buff,size_t n_bytes);
        ssize_t Write(const void *buff,size_t n_bytes);
};
```

La classe ClientSocket rappresenta il lato client della comunicazione utilizzando le socket ereditate dalla classe SocketCommunication.

Attributi:

- *socket_fd*: file descriptor della socket per la comunicazione;
- *server_IP*: IP del server a cui il client vuole connettersi;
- *serverToConnect*: struttura che rappresenta il server a cui ci si vuole connettere;
- *port*: ereditato da *SocketCommunication*, rappresenta la porta alla quale il client vuole connettersi.

Metodi:

- *ClientSocket()*: costruttore che imposta l'ip e la porta del server a cui connettersi. Prende in input l'ip del server (*serverIP*) e la porta sulla quale è hostato il server (*server*);
- *clientSetup()*: procedura che inizializza la comunicazione: permette di ottenere il file descriptor della scoket, imposta i dati del server e rende pronto il client per la comunicazione;
- *Connect():* procedura che wrappa la chiamata di sistema *connect()* per connettersi al server. La funzione senza parametri richiama la system call utilizzando gli attributi salvati nell'oggetto mentre la versione con i parametri richiama sempre la system call con la possibilità di fornire in input direttamente i parametri per la system call;
- *changeServerIP()*: procedura che permette di cambiare l'IP del server a cui ci si vuole connettere, impostandolo in formato network e aggiornandolo nella struttura *serverToConnect*. Prende in input il nuovo ip del server (*newIP*);
- *changeServerPort()*: procedura che permette di cambiare la porta del server a cui ci si vuole connettere, impostandolo in formato network e aggiornandolo nella struttura *serverToConnect*. Prende in input la nuova porta del server (*portToConnect*);
- *disconnect()*: procedura che chiude il file descriptor della socket;
- *Read() / Write()*: procedure che wrappano le *FullRead()* e *FullWrite()* sulla connessione stabilita dal client. I parametri sono gli stessi delle rispettive system call.

File ServerSocket

```
class ServerSocket: protected SocketCommunication{
    private:
        int listen_fd;
        int connection_fd;
        sockaddr_in me;
        sockaddr_in client;
        socklen_t client_length;

        int Bind(int socket_fd,const struct sockaddr_in server_address);
        int Listen(int socket_fd, int queueLength);
    public:
        ServerSocket(int portToHost);
        void serverSetup();
        int Accept_NoClient();
        int Accept_NoClient(int socket_fd);
        int Accept_Client();
        int Accept_Client(int socket_fd,struct sockaddr_in *connected_client,socklen_t
*client_lenght);
        void changePort(int portToHost);
        void closeListening();
        void closeServer();
        ssize_t Read(void *buff,size_t n_bytes);
        ssize_t Write(const void *buff,size_t n_bytes);
};
```

La classe *ServerSocket* eredita da *SocketCommunication* e rappresenta la socket dal lato server.

Attributi:

- *listen_fd*: file descriptor di ascolto della socket;
- *connection_fd*: file descriptor della socket una volta accettata la connessione dal client;
- *me*: struttura che rappresenta il server stesso per l'impostazione della comunicazione;
- *client*: struttura che rappresenta il client la cui connessione viene accettata;
- *client_length*: struttura che rappresenta la lunghezza del client.

Metodi:

- *ServerSocket()*: costruttore che imposta la porta su cui il server sarà hostato. Prende in input la porta (*portToHost*);
- *serverSetup()*: procedura di inizializzazione della comunicazione: permette di ottenere il file descriptor della scoket, imposta i dati del server e rende pronto il server per l'ascolto e l'accettazione;
- *Bind()*: procedura che wrappa la chiamata di sistema *bind()*;
- *Listen()*: procedura che wrappa la chiamata di sistema *listen()*;
- *Accept_NoClient()*: procedura che wrappa la chiamata di sistema *accept()* salvandosi in *connection_fd* il file descriptor per la comunicazione con il client. Non vengono salvate le informazioni sul client che si connette. La versione senza parametri richiama la system call utilizzando gli attributi dell'oggetto mentre quella con i parametri li inoltra alla system call;
- *Accept_Client()*: analoga alla precedente con la differenza che si salvano le informazioni del client connesso. La versione senza parametri salva le informazioni all'interno dell'oggetto mentre quella con i parametri li inoltra alla system call;

- *changePort()*: cambia la porta su cui il server è hostato impostandola in formato network. In input prende la porta (*portToHost*);
- *closeListening()*: procedura che chiude il file descriptor di ascolto;
- *closeServer()*: procedura che chiude il file descriptor della socket e della connessione con il client;
- *Read()* / *Write()*: procedure che wrappano le *FullRead()* e *FullWrite()* sulla connessione stabilita dal client. I parametri sono gli stessi delle rispettive system call.

File DataStructures

Il file *DataStructures* contiene tutte le strutture dati utilizzate dai programmi main e dalle implementazioni delle classi. Queste strutture dati permettono di inviare i vari dati in modo organizzato e non grezzo.

Struct Packet

```
#define MATRICOLA_STUDENTE 0
#define CORSO 1
#define ESAME 2
#define APPELLO 3
#define RIGHE_QUERY 4
#define GENERIC_DATA 5

struct Packet{
    RequestType request = LOGIN;
    Error error;
    int data[6];
};
```

La struct *Packet* rappresenta il pacchetto che viene inviato sempre prima di altri eventuali dati. È utilizzato per inviare i codici delle richieste ed eventuali dati. La struct è composta da:

- *request*: contiene il codice della richiesta da eseguire;
- *error*: contiene l'errore che si è verificato durante l'esecuzione di una richiesta;
- *data*: contiene eventuali dati da inviare insieme alla richiesta (es. quando lo studente effettua il login invia la propria matricola in questo array).

Le macro definite in alto specificano gli indici del vettore all'interno del quale si devono inserire o leggere specifici dati. Queste indicano:

- *MATRICOLA_STUDENTE*: indice nel quale si trova la matricola di uno studente (usato nelle operazioni di login, visione appelli, visione appelli prenotati e alla prenotazione di un appello);
- *CORSO*: indice nel quale si trova il codice di un corso (usato nelle operazioni di visione esami);
- *ESAME*: indice nel quale si trova il codice di un esame (usato nell'operazione di visione appelli);
- *APPELLO*: indice nel quale si trova il codice di un appello (usato nell'operazione di prenotazione di un appello);
- *RIGHE_QUERY*: indice nel quale si trova il numero di record ottenuti dalle operazioni di visione esami, visione appelli e visione corsi;
- *GENERIC_DATA*: indice nel quale si trova un generico valore (usato per il numero progressivo come conferma di prenotazione).

Struct per la gestione delle tabelle

```c
struct Appello{
    int codiceAppello;
    int codiceEsame;
    char data[17];
};
```

```c
struct Esame{
    int codiceEsame;
    int codiceCorso;
    char tipo[6];
    char modalita[8];
    char descrizione[501];
};
```

```c
struct AppelloPrenotato{
    int codice;
    char data[17];
    char nome[51];
    char tipo[6];
    char modalita[8];
    char descrizione[501];
    short int numeroPrenotazione;
};
```

```c
struct AppelloDisponibile{
    int codiceAppello;
    char data[17];
    char nome[51];
};
```

```c
struct Corso{
    int codiceCorso;
    char nome[51];
    short int CFU;
};
```

Queste struct rispecchiano le tabelle salvate nel database. Sono usate sia per inviare dati al server universitario che per riceverli, quindi visualizzarli. Le struct *AppelloDisponibile* e *AppelloPrenotato* sono usate soltanto per ricevere le tabelle dal server universitario nel caso delle operazioni di visione appelli disponibile e di visione appelli prenotati.

Esempio segreteria client/server (visione appelli prenotati)

```
Packet risposta_server;
ClientSocket client=ClientSocket("127.0.0.1",UNI_SERVER_PORT);
client.clientSetup();
client.Connect();
client.Write(&richiesta,sizeof(richiesta));
client.Read(&risposta_server,sizeof(risposta_server));
if(risposta_server.error.getCode()==OK){
    int num_righe=risposta_server.data[RIGHE_QUERY];
    cout<<"Numero appelli trovati per lo
studente"<<risposta_server.data[MATRICOLA_STUDENTE]<<":"<<risposta_server.data[RIGHE_QUERY
]<<endl;
   server.Write(&risposta_server, sizeof(risposta_server));
   if(num_righe!=0){
       AppelloPrenotato *appelli=new AppelloPrenotato[num_righe];
       client.Read(appelli,sizeof(AppelloPrenotato)*num_righe);
       server.Write(appelli,sizeof(AppelloPrenotato)*num_righe);
       delete[] appelli;
   }
} else {
    risposta_server.error.setCode(GENERIC);
    server.Write(&risposta_server,sizeof(risposta_server));
}
client.disconnect();
```

Si può notare che il server della segreteria assume il ruolo di client poiché deve contattare il server universitario per ricevere i dati delle tabelle. La possibilità di mantenere una connessione sia con il server universitario e sia con lo studente è possibile grazie al fatto che la comunicazione con le socket è memorizzata e gestia all'interno degli oggetti *ClientSocket* e *ServerSocket*, quindi si possono utilizzare entrambi senza generare confilitti con i file descriptor.

Istruzioni di utilizzo

1. Nella root directory del progetto bisogna eseguire il file “sqlite-installation.sh” da terminale. Questo permette di installare la libreria sqlite all’interno del sistema;
2. Andare alla cartella “codice” e compilare i file sorgente digitando il comando “make” sul terminale.
3. I file che dovranno essere eseguiti da terminale sono: “server_uni”, “segreteria_server”, “segreteria_client” e “studente”. Ogni file deve essere eseguito su un terminale diverso.