



UNIVERSITÀ DEGLI STUDI  
DI NAPOLI FEDERICO II

# POTHLES

PROGETTAZIONE ED IMPLEMENTAZIONE DI UN SISTEMA CLIENT-  
SERVER PER MANIPOLAZIONE DI INFORMAZIONI SULLA PRESENZA  
DI IRREGOLARITÀ SU UNA SUPERFICIE

a.a. 2021 – 2022

LSO\_2122\_2

Marotta	Vincenzo	N86003005
Muto	Emanuele Ciro	N86003440
Salernitano	Mattia	N86003385



# Sommario

Descrizione del progetto.....	3
1.1 Analisi e funzionalità del problema .....	3
Descrizione del protocollo.....	4
2.1 Introduzione.....	4
2.2 Il Protocollo ECM .....	4
2.2.1 Echo.....	4
2.2.2 Initialization Check .....	5
2.2.3 Registration .....	5
2.2.4 Login.....	6
2.2.5 Near Event.....	6
2.2.6 Get Accelerometer Value.....	7
2.2.7 Save Coords .....	7
2.2.8 Close Connection.....	7
Dettagli implementativi.....	8
3.1 Introduzione.....	8
3.2 Descrizione del server .....	8
3.3 Descrizione ThreadPool .....	13
Descrizione del Client.....	18
4.1 Introduzione.....	18
4.2 Il Client .....	18
Guida d'uso.....	23
5.1 Introduzione.....	23
5.2 Server .....	23
5.3 Client.....	24

# Capitolo 1

## Descrizione del progetto

### 1.1 ANALISI E FUNZIONALITÀ DEL PROBELMA

Si progetterà ed implementerà un sistema client-server che consenta la raccolta e l'interrogazione di informazioni riguardanti la presenza di irregolarità (buche) su una superficie.

Il sistema dovrà memorizzare una lista di eventi generati dal client riguardanti la rilevazione di un improvviso cambiamento lungo l'asse verticale dell'accelerazione, che superi una determinata soglia comunicata dal server in fase di connessione.

Il client dovrà:

- garantire una sezione per la registrazione e una sezione per effettuare l'accesso. Ogni client sarà identificato da un nickname scelto dall'utente in fase di registrazione;
- permettere all'utente di avviare una sessione di registrazione eventi, durante la quale il client si connette al server, riceve i parametri di soglia e comunica al server la posizione ed il valore del cambiamento ogni volta che viene registrato un nuovo evento;
- mostrare all'utente la lista di tutti gli eventi registrati dal server in un certo raggio dalla propria posizione;
- mostrare all'utente gli eventi vicini su mappa.

# Capitolo 2

## Descrizione del protocollo

### 2.1 INTRODUZIONE

Dopo aver analizzato e descritto per sommi capi le funzionalità del sistema, si passa alla descrizione del protocollo al livello di applicazione utilizzato nelle comunicazioni tra client e server.

### 2.2 IL PROTOCOLLO ECM

Il protocollo ECM (Enhanced Coordinate Messaging Protocol) permette la comunicazione tra il server e il client in modo semplice ed efficace.

È stato ideato con uno sguardo al futuro, in modo che sia facilmente scalabile senza stravolgere il codice già scritto.

La struttura di un comando ECM segue il seguente pattern:

**HEADER -> PACKET\_SIZE -> COMMAND -> OTHER\_DATA\_ACCORDING\_TO\_THE\_COMMAND**

Dalle analisi delle specifiche commissionate, si è deciso un approccio dove è sempre il client a comunicare per primo.

Si è scelto anche di utilizzare una nomenclatura dei comandi esplicita per permettere una facile comprensione degli stessi.

I comandi sono strutturati in questo modo:

#### 2.2.1 Echo

La richiesta *ECHO* viene utilizzata sia dal client che dal server per verificare che la connessione tra di loro sia ancora attiva.

#### Client – Server

```
ECM_PROT_V1
<packet size> : uint_32
ECHO_REQUEST\n
```

#### Client – Server

```
ECM_PROT_V1
<packet size> : uint_32
ECHO_REPLY\n
```

### 2.2.2 Initialization Check

La richiesta *CHECK* permette di conoscere lo stato di disponibilità del server.

Ad esempio, se il server ha il database in manutenzione, è sempre possibile ricevere una notifica di indisponibilità a offrire il servizio, senza che il client si ritrovi solo un errore di irraggiungibilità del server.

#### Client

```
ECM_PROT_V1
<packet size> : uint_32
CHECK\n
```

#### Server

```
ECM_PROT_V1
<packet size> : uint_32
CHECK_OK\n
```

oppure

```
ECM_PROT_V1
<packet size> : uint_32
CHECK_FAIL\n
```

### 2.2.3 Registration

La registrazione di un nuovo utente avviene con il client che invia le richieste *USR\_REG\_REQUEST* e i dati necessari. Il server può rispondere *USR\_REG\_OK* in caso di successo, ovvero *USR\_REG\_FAIL* in caso ci siano problemi con la registrazione.

#### Client

```
ECM_PROT_V1
<packet size> : uint_32
USR_REG_REQUEST\n
<user>\n : char
<password>\n : char
<email>\n : char
<name>\n : char
<surname>\n : char
```

#### Server

```
ECM_PROT_V1
<packet size> : uint_32
USR_REG_OK\n
```

oppure

```
ECM_PROT_V1
<packet size> : uint_32
USR_REG_FAIL\n
```

### 2.2.4 Login

Il login avviene con la richiesta *LOGIN\_REQUEST* e i suoi dati. Il server può rispondere con *LOGIN\_OK* in caso di successo, ovvero con *LOGIN\_FAIL* nel caso l'utente non risulti registrato. La risposta *USR\_NOT\_LOGGED* viene inviata al client qual ora quest'ultimo effettui una qualsiasi altra richiesta, diversa da *ECHO*, dalla *CHECK* e dalla registrazione, senza che l'utente abbia prima effettuato il login.

#### Client

```
ECM_PROT_V1
<packet size> : uint_32
LOGIN_REQUEST\n
<user>\n : char
<password>\n : char
```

#### Server

```
ECM_PROT_V1
<packet size> : uint_32
LOGIN_OK\n
```

oppure

```
ECM_PROT_V1
<packet size> : uint_32
LOGIN_FAIL\n
```

oppure

```
ECM_PROT_V1
<packet size> : uint_32
USR_NOT_LOGGED\n
```

### 2.2.5 Near Event

Con la richiesta *NEAR\_EVENT\_REQUEST* il client richiede al server la presenza di buche nelle vicinanze delle coordinate allegate, con un certo raggio. Nel caso siano presenti buche corrispondenti alla richiesta allora il server risponde con *NEAR\_EVENT\_REPLY* e la lista delle buche (posizione e valore dell'accelerometro), ovvero risponde *NEAR\_EVENT\_NOT\_FOUND* nel caso non ci siano buche nelle vicinanze delle coordinate inviate.

#### Client

```
ECM_PROT_V1
<packet size> : uint_32
NEAR_EVENT_REQUEST\n
<max_distanza> : uint32;
<coords_payload> : struct
ecm_accelerometer_coords_t
```

#### Server

```
ECM_PROT_V1
<packet size> : uint_32
NEAR_EVENT_REPLY\n
<num_coords> : uint32
<coords_payload> : struct
ecm_accelerometer_coords_t
```

oppure

```
ECM_PROT_V1
<packet size> : uint_32
NEAR_EVENT_NOT_FOUND\n
```

### 2.2.6 Get Accelerometer Value

Quando il client chiede il valore soglia per l'accelerometro con la richiesta *ACCELEROMETER\_REQUEST*, il server risponde con *ACCELEROMETER\_RESPONSE* e il valore soglia desiderato.

#### Client

```
ECM_PROT_V1
<packet size> : uint32
ACCELEROMETER_REQUEST\n
```

#### Server

```
ECM_PROT_V1
<packet size> : uint_32
ACCELEROMETER_RESPONSE\n
<accelerometer threshold> :
double
```

### 2.2.7 Save Coords

Quando il cliente è pronto per salvare la posizione di nuove buche, viene utilizzata la richiesta *SAVE\_COORDS\_REQUEST* seguita dalla lista delle coordinate da salvare. Da notare che, anche se nelle specifiche di progetto è stato richiesto che la connessione al server rimanesse attiva per tutto il tempo di salvataggio e, quindi, si sarebbe potuto evitare l'invio di una lista di posizioni, si è scelta la lista proprio in caso di cambiamenti futuri delle specifiche.

#### Client

```
ECM_PROT_V1
<packet size> : uint_32
SAVE_COORDS_REQUEST\n
<num_coords> : uint32
<coords_payload> : struct
ecm_accelerometer_coords_t
```

#### Server

```
ECM_PROT_V1
<packet size> : uint_32
SAVE_COORDS_OK\n

oppure

ECM_PROT_V1
<packet size> : uint_32
SAVE_COORDS_FAIL\n
```

### 2.2.8 Close Connection

Quando il client ha finito, invia al server la comunicazione che sta per chiudere la connessione con la richiesta *CONN\_CLOSE\_ANNOUNCEMENT*. Il server ricevuto questa richiesta, invia una conferma al client con *CONN\_CLOSE\_OK*.

#### Client

```
ECM_PROT_V1
<packet size> : uint_32
CONN_CLOSE_ANNOUNCEMENT\n
```

#### Server

```
ECM_PROT_V1
<packet size> : uint_32
CONN_CLOSE_OK\n
```



# Capitolo 3

## Dettagli implementativi

### 3.1 INTRODUZIONE

Dopo aver fornito una descrizione più accurata del protocollo, si passa ad una descrizione più accurata di alcuni dettagli implementativi più interessanti.

### 3.2 DESCRIZIONE DEL SERVER

Per la gestione delle connessioni multiple si è deciso di creare un server con architettura multi-thread, dove invece di assegnare ogni connessione ad uno specifico thread, si è scelto un approccio più raffinato implementando un semplice thread pool da noi scritto.

Il server per la gestione dei dati si affida a un database *Postgress*.

Il server, una volta avviato, creerà la socket:

```
// Creo la socket
int socketDescriptor = socket(PF_INET, SOCK_STREAM, 0);
if (socketDescriptor < 0)
{
    time(&currentTime);
    fprintf(stdout, "%s -> Socket creation error.\n\n",
ctime(&currentTime));
    exit(EXIT_FAILURE);
}
```

setterà la struttura dati per ascolto:

```
// Setto i dati per l'ascolto
struct sockaddr_in myAddress;
myAddress.sin_family = AF_INET;
myAddress.sin_port = htons(5200);
myAddress.sin_addr.s_addr = htonl(INADDR_ANY);
```

eseguirà il binding

```
// Eseguo il bind

if (bind(socketDescriptor, (struct sockaddr *)&myAddress, sizeof(myAddress)) < 0)
{
    time(&currentTime);
    fprintf(stdout, "%s -> Bind error.\n\n", ctime(&currentTime));
    close(socketDescriptor);
    exit(EXIT_FAILURE);
}
```

inizierà l'ascolto sulla porta indicata

```
// Avvio l'ascolto.

if (listen(socketDescriptor, 1) < 0)
{
    time(&currentTime);
    fprintf(stdout, "%s -> Listen error.\n\n", ctime(&currentTime));
    close(socketDescriptor);
    exit(EXIT_FAILURE);
}
```

Una volta che il tutto è pronto un ciclo while procederà ad accettare, di volta in volta, le nuove connessioni, inserire il file descriptor e le informazioni della connessione in una struttura dati di tipo *clientDescription\_t*; a questo punto, questa struttura contenente la descrizione del client verrà inviato aggiunta alla coda del pool, come argomenti, insieme al puntatore alla funzione di gestione del client.

I thread del pool, di volta in volta, preleveranno i dati dalla coda e gestiranno la comunicazione con il client.

Nel caso la coda del pool di thread fosse piena, il server chiuderà subito la connessione con il nuovo client: quest'ultimo dovrà effettuare una nuova connessione quando la coda avrà spazio disponibile.

```

// Ciclo principale - accettazione client.

clientDescription_t *client;
while (1)
{
    client = (clientDescription_t *)calloc(1, sizeof(clientDescription_t));
    client->clientSocketlength = sizeof(client->clientAddress);
    client->clientFileDescriptor = accept(socketDescriptor, (struct sockaddr *)&(client->clientAddress), &(client->clientSocketlength));
    char ipStr[INET_ADDRSTRLEN];
    inet_ntop(AF_INET, &(client->clientAddress.sin_addr), ipStr, INET_ADDRSTRLEN);
    if ((strcmp(ipStr, "127.0.0.1") == 0) || (strcmp(ipStr, "0.0.0.0") == 0))
    {
        close(client->clientFileDescriptor);
        free(client);
        break;
    }
    time(&currentTime);
    _printMessageStdOutput(&currentTime, "Accepted connection", ipStr);
    if (threadpool_addWork(myPool, manageClient, client) == 0){
        //Chiudo connessione in caso di coda piena
        close(client->clientFileDescriptor);
        free(client)
    }
}
}

```

La funzione di gestione del client è un ciclo while che, ad ogni iterazione, legge dal file descriptor associato alla socket una richiesta proveniente dal client e poi risponde. Quando viene ricevuto l'annuncio dal client della chiusura della connessione, si uscirà da questo ciclo while e verrà chiusa la connessione.

Il server è anche in grado di gestire un contatore di timeout: se il server non riceve una richiesta dal client o, al più, una richiesta ECHO entro un intervallo stabilito, allora chiuderà la connessione e si passerà a servire il prossimo client.

Tale procedura è possibile grazie alle seguenti variabili:

```

struct pollfd pollFileDescription;

pollFileDescription.fd = client->clientFileDescriptor;
pollFileDescription.events = POLLIN;

```

ed alla funzione poll()

```
if(poll(&pollFileDescription, 1, TIMEOUT_CONNECTION) == 0)
    {time(&currentTime);
      _printMessageStdOutput(&currentTime, "Timeout", ipString);
      break;
    }
```

che permette di sbloccare il file descriptor associato alla socket passato un tempo di timeout senza che vi sia letto o scritto qualcosa dal predetto file descriptor.

Per quanto riguarda la chiusura del server, basta inviare un segnale di chiusura del processo (ctrl + c) per avviare la procedura tramite l'handle del segnale.

Poiché la funzione di accept() delle connessioni è bloccante, si è scelto di utilizzare una connessione fake da localhost.

```
// Gestico i segnali
signal(SIGINT, closeServerHandler);
signal(SIGTERM, closeServerHandler);
```

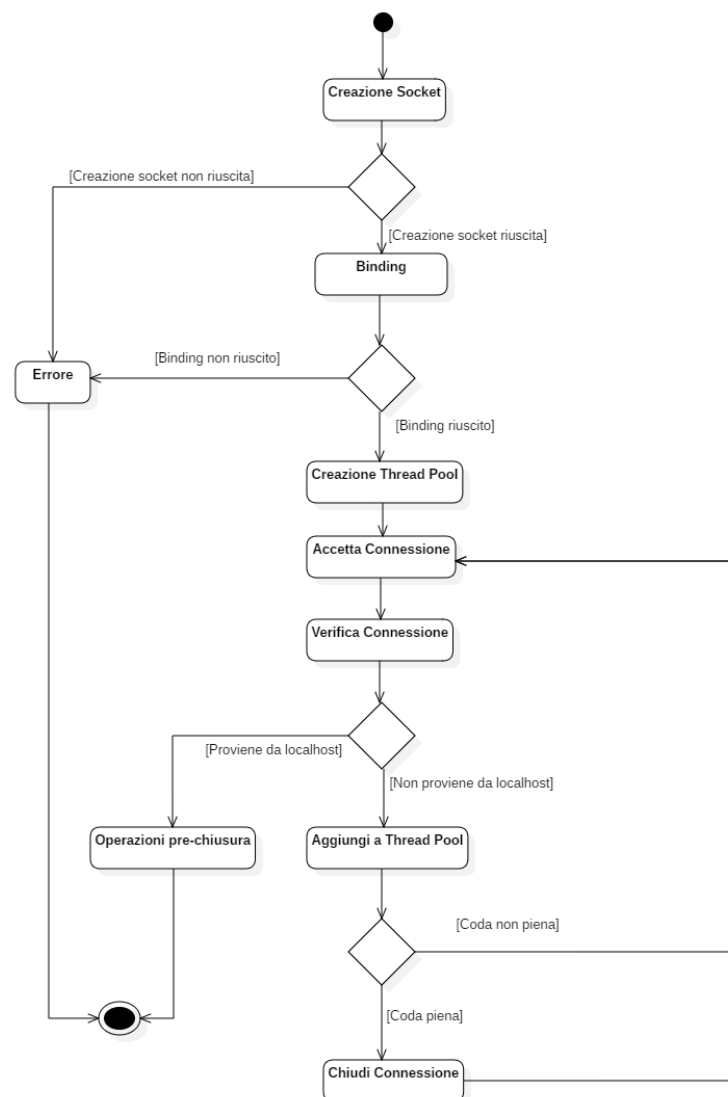
```
/*
Specifica le azioni da compiere quando viene rilevato un segnale di terminazione.
Crea una finit connessione che viene interpretato come flag per uscire dal ciclo
while del
thread principale.
*/
void closeServerHandler(int sigID)
{
    time_t currentTime;
    time(&currentTime);
    char message[50] = "Riceived server termination signal (";
    char id[10];
    sprintf(id, "%d", sigID);
    strcat(message, id);
    strcat(message, ")");
    _printMessageStdOutput(&currentTime, message, "localhost");
    runServer = 0;
}
```

```
// Connessione fake per chiudere il server uscendo dal while.
```

```
struct sockaddr_in serv_addr;  
int sock = socket(PF_INET, SOCK_STREAM, 0);  
serv_addr.sin_family = AF_INET;  
serv_addr.sin_port = htons(5200);  
inet_pton(AF_INET, "127.0.0.1", &serv_addr.sin_addr);  
int fd = connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr));  
close(fd);  
close(sock);  
}
```

Quindi se al momento dell'accettazione della nuova connessione, la connessione proviene da localhost, si uscirà dal ciclo while di accettazione delle connessioni, verrà chiamata la chiusura del pool di thread e il server terminerà la sua esecuzione.

A lato viene riportato un Activity Diagram che sintetizza e mostra in modo immediato il funzionamento del server.



### 3.3 DESCRIZIONE THREADPOOL

È stato scelto l'utilizzo di un pool di thread per avvicinarci a una soluzione meno accademica e più reale, quindi, sarà possibile accettare un numero di connessioni parallele di client pari al numero di thread scelto per il pool.

Senza complicare troppo la soluzione proposta, è stato deciso di implementare un semplice pool di thread, dove ogni thread esegue una funzione, con i suoi argomenti, passata per riferimento e presa da una struttura dati di tipo "*threadpool\_work\_t*" (da qui in poi chiamata "work"); il pool ha una coda di lunghezza fissa (la cui lunghezza è scelta a priori), può fermare e avviare i thread del pool ma non prevede un sistema di controllo dello stato di esecuzione dei singoli thread assegnati al già menzionato pool.

Partiamo con l'introdurre la struttura dati work che permette lo storage delle funzioni da far eseguire ai thread del pool.

```
/* Struttura di un lavoro. */
typedef struct threadpool_work {
    thread_func_t threadFunction;    // Funzione che deve essere eseguita.
    void *argument;                 // Argomenti della funzione che deve essere
    eseguita.
    struct threadpool_work *next;    // Puntatore al prossimo elemento della coda
    di lavori.
} threadpool_work_t;
```

Dove:

- il tipo `thread_func_t` è la ridenominazione tramite typedef del codice per il passaggio di una funzione tramite riferimento;
- `void *argument` è il puntatore alla struttura contenente gli argomenti per la funzione che eseguirà il thread;
- `struct threadpool_work *next` è il puntatore alla seguente struttura dati work contenuta nella coda dei lavori

Adesso passiamo al cuore del pool di thread: la struttura dati stessa del pool.

```

/* Struttura di un thread pool */
typedef struct threadpool {
    threadpool_work_t *firstWork;           // Puntatore al primo elemento della coda dei
    lavori.
    threadpool_work_t *lastWork;            // Puntatore all'ultimo elemento della coda
    dei lavori.
    size_t maxNumberOfParallelThreads;      // Numero massimo dei thread eseguiti in
    parallelo.
    size_t maximumNumberOfWork;             // Dimensione massima della coda dei lavori.
    size_t currentNumberOfWork;             // Dimensione corrente della coda dei lavori.
    pthread_mutex_t workMutex;              // Mutex per la modifica della coda dei
    lavori.
    pthread_cond_t workCondition;           // Cond per i thread. Serve per fermare i
    thread se non hanno lavoro o se il flag di stop è settato.
    volatile bool run;                      // Booleano per impostare lo stato di
    elaborazione dei thread.
    volatile bool killAll;                  // Booleano per impostare lo stato di
    terminazione dei thread.
    pthread_mutex_t endMutex;               // Mutex per incrementare il contatore di
    thread terminati.
    volatile size_t numEndedThread;         // Numero dei thread terminati.
    pthread_cond_t endCondition;            // Cond per fermare il thread principale in
    modo da attendere la terminazione dei thread del pool.
} threadpool_t;

```

Tralasciando i puntatori per la lista delle strutture work e le variabili di dimensionamento del pool di thread, abbiamo:

- il mutex "workMutex" per gestire l'accesso alla coda di work;
- la variabile condizionale "workCondition", con il mutex workMutex, che serve per sospendere i thread quando non ci sono work da elaborare;
- il booleano "run" che serve per indicare lo stato di run dell'intero pool: quando i thread vedono che questa variabile è impostata a false, si sospendono fino a che non ricevono il broadcast di cambiamento della variabile;
- il booleano "killAll" serve per indicare ai thread che devono terminarsi: in pratica serve per uscire dal ciclo while del codice di base che esegue ogni thread;
- il mutex "endMutex" che serve a gestire la concorrenza, dei thread del pool, di incrementare la variabile "numEndedThred";

- la variabile "numEndedThread" è il contatore de numero di thread del pool che sono terminati: finché questo contatore non assume lo stesso valore del numero di thread paralleli del pool, il thread principale rimane sospeso.
- la variabile condizionale "endCondition" che serve a sospendere il thread principale nell'attesa che tutti i thread del pool terminino.



Vediamo adesso come ogni thread lavora dando un'occhiata al loro codice base:

```
/*
Codice per i thread che eseguono le funzioni.
*/
static void *_threadpool_threadWorkFuncion(void *args)
{
    threadpool_t *myThreadPool = (threadpool_t *)args;
    threadpool_work_t *workToDo;

    while (myThreadPool->killAll == false)
    {
        pthread_mutex_lock(&(myThreadPool->workMutex));
        while ((myThreadPool->firstWork == NULL) || (myThreadPool->run ==
THREADPOOL_STOP))
        {
            if (myThreadPool->killAll == true)
            {
                pthread_mutex_unlock(&(myThreadPool->workMutex));
                break;
            }
            pthread_cond_wait(&(myThreadPool->workCondition), &(myThreadPool-
>workMutex));
        }

        if (myThreadPool->killAll == true)
        {
            pthread_mutex_unlock(&(myThreadPool->workMutex));
            break;
        }

        workToDo = _threadpool_getWork(myThreadPool);
        pthread_mutex_unlock(&(myThreadPool->workMutex));
        // Esegue la funzione passando gli argomenti.
        workToDo->threadFunction(workToDo->argument);
        _threadpool_destoryWork(workToDo);
    }

    pthread_mutex_lock(&(myThreadPool->endMutex));
    (myThreadPool->numEndedThread)++;
    pthread_mutex_unlock(&(myThreadPool->endMutex));
    pthread_cond_broadcast(&(myThreadPool->endCondition));

    return NULL;
}
```

Al thread viene passato come argomento il puntatore al suo pool in modo da poter accedere alle predette variabili (coda dei work, booleani, mutex e condition), in modo che il tutto funzioni.

Il thread eseguirà un ciclo while finché la variabile booleana killAll non sarà uguale a true.

Nel ciclo while il thread acquisirà il mutex della coda, ma potrebbe sospendersi nel caso in cui non ci siano work nella coda, ovvero il booleano di run è settato a false; una volta soddisfatta una delle condizioni, cioè che sia presente un work nella coda o il booleano run sia stato impostato a true, il thread riprenderà il lavoro.

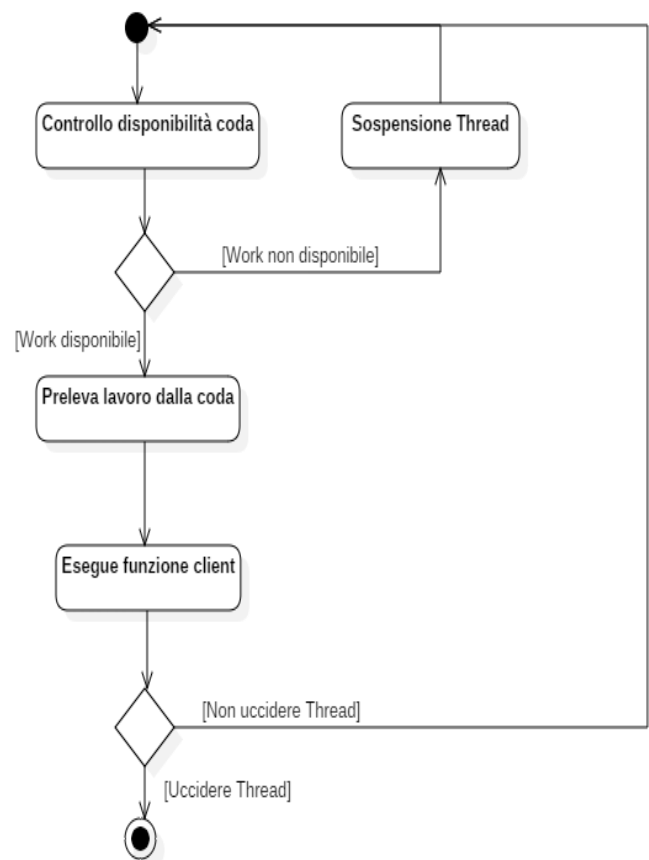
A questo punto il thread preleverà una struttura, sbloccherà il mutex, eseguirà la funzione con gli argomenti contenuti nel work e, infine, a elaborazione completata, deallocherà la struttura work.

A questo punto il ciclo riprenderà daccapo.

Invece, nel caso un thread debba uscire dal ciclo while, acquisirà il mutex per aggiungere il contatore dei thread terminati, eseguirà l'incremento del predetto contatore, rilascerà il mutex e, infine, eseguirà il broadcast sulla variabile condizionale che interrompe il thread principale.

Questa impostazione del codice base di un thread implica che, anche se si fosse deciso di interrompere i thread, questi esisteranno finché il lavoro corrente non sarà terminato.

Di seguito viene riportato un Activity Diagram che sintetizza e mostra in modo immediato il funzionamento del pool di thread.



# Capitolo 4

## Descrizione del Client

### 4.1 INTRODUZIONE

Dopo aver descritto più dettagliatamente alcuni dettagli implementativi, passiamo ad una descrizione un po' più nel dettaglio del client.

### 4.2 IL CLIENT

Dal lato client, il cuore delle comunicazioni di rete avviene grazie a due classi che, lavorando insieme, permettono una di inviare/ricevere le informazioni dal server e, l'altra, codificare/decodificare le informazioni stesse.

La classe ECMServerConnector, la classe che si occupa delle comunicazioni con il server, viene inizializzata con la creazione della socket, l'associazione dei data stream, l'inizializzazione del thread di ricezione (vedi più avanti) e la chiamata al metodo di callback in base al risultato della creazione della socket.

```
try {
    socket = new Socket(ip, port);
    input = new DataInputStream(socket.getInputStream());
    output = new DataOutputStream(socket.getOutputStream());
    while (!socket.isConnected())
        ;
    startReceiver();
    callback.onConnectionReady();
} catch (ConnectException e) {
    callback.onConnectionRefused();
} catch (IOException e2) { // UnknownHostException viene catturato qui.
    callback.onConnectionError();
}
```

L'invio dei dati avviene con un semplice metodo che scrive nella socket tramite lo stream di output:

```
try {
    synchronized (output) {
        output.write(dataToSend);
        output.flush();
    }
} catch (IOException e) {
    e.printStackTrace();
}
```

La ricezione dei dati, invece, viene affidata ad un thread separato:

```
receiverThread = new Thread() -> {
    while ((socket.isConnected()) && (!socket.isClosed()) &&
(!Thread.currentThread().isInterrupted())) {
        try {
            if (input.available() > 0) {
                byte[] protocol = new byte[11];
                input.read(protocol, 0, 11);
                System.out.println("Protocol -> " + new String(protocol));
                if (new String(protocol).equals("ECM_PROT_V1")) {
                    byte[] temp = new byte[4];
                    input.read(temp, 0, 4);
                    ByteBuffer packetSizeBuffer = ByteBuffer.wrap(temp)
                        .order(ByteOrder.LITTLE_ENDIAN);
                    int payloadSize = packetSizeBuffer.getInt();
                    byte[] payload = new byte[payloadSize];
                    input.read(payload, 0, payloadSize);
                    notifyAllReceiverCallback(payload, payloadSize);
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
};
receiverThread.start();
```

quindi il metodo, prima verifica se il pacchetto ricevuto inizia con l'header giusto e, nel caso, prepara il buffer e invia il payload alla seconda e importante classe ECMServerDAO.

La codifica delle informazioni dirette al server e la decodifica delle informazioni ricevute da quest'ultimo, viene affidata alla classe ECMServerDAO.

Questa classe deve essere registrata alla predetta classe ECMServerConnector da cui riceve le informazioni dal server, ovvero, invia le informazioni al server.

Le informazioni ricevute vengono smistate ai vari metodi di decodifica tramite l'analisi del comando contenuto nel payload del pacchetto:

```
connectorCallback = new ECMServerConnectorReceiverCallback() {

    @Override
    public void onReceived(byte[] payload, int payloadSize) {
        ByteBuffer buffer = ByteBuffer.wrap(payload).order(ByteOrder.LITTLE_ENDIAN);
        int commandLen = 0;
        while (payload[commandLen] != '\n')
            commandLen++;
        byte[] command = new byte[commandLen];
        buffer.get(command, 0, commandLen);
        String commandString = new String(command);
        buffer.get(); //Serve per togliere \n dopo aver rimosso il comando dal buffer.
    }
};
```

```

switch (commandString) {
    case "CHECK_OK":
        receivedCheckOk();
        break;
    case "CHECK_FAIL":
        receivedCheckFail();
        break;
    case "USR_REG_OK":
        receivedUserRegistrationOk();
        break;
    case "USR_REG_FAIL":
        receivedUserRegistrationFail();
        break;
    case "LOGIN_OK":
        receivedLoginOk();
        break;
    case "LOGIN_FAIL":
        receivedLoginFail();
        break;
    case "USR_NOT_LOGGED":
        receivedUserNotLogged();
        break;
    case "NEAR_EVENT_REPLY":
        receivedNearEventReply(buffer);
        break;
    case "NEAR_EVENT_NOT_FOUND":
        receivedNearEventNotFound();
        break;
    case "ACCELEROMETER_RESPONSE":
        receivedAccelerometerThreshold(buffer);
        break;
    case "SAVE_COORDS_OK":
        receivedSaveCoordsOk();
        break;
    case "SAVE_COORDS_FAIL":
        receivedSaveCoordsFail();
        break;
    case "CONN_CLOSE_OK":
        receivedCloseConnectionConfirmed();
        break;
    case "ECHO_REQUEST":
        receivedEchoRequest();
        break;
    case "ECHO_REPLY":
        receivedEchoReply();
        break;
    default:
        callback.onNotValidCommand();
        break;
}
}

};
connector.registerECMServerConnectorReceiverCallback(connectorCallback);

```

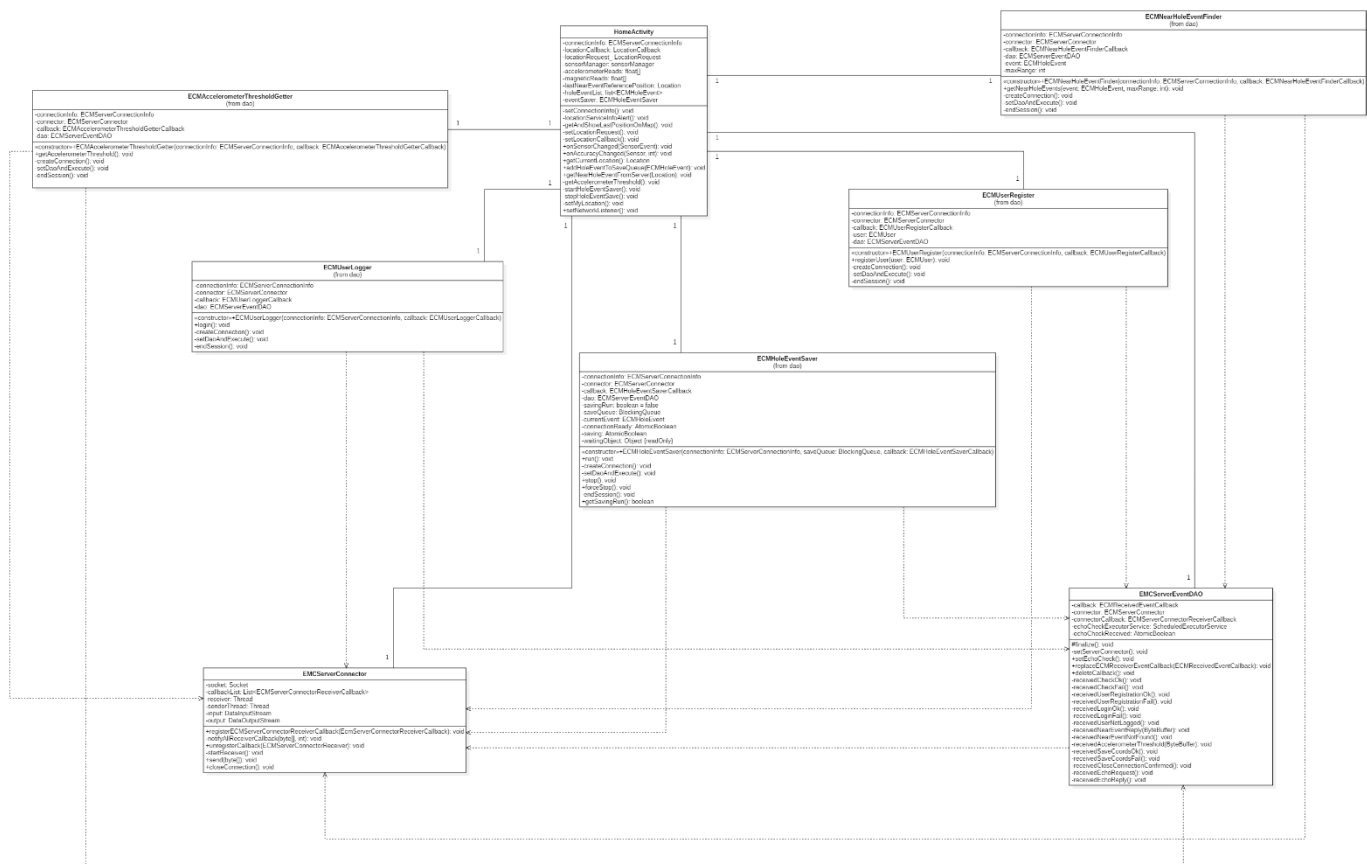
Per quanto riguarda la verifica della connessione al server, uno specifico metodo setta un esecutore che, a intervalli regolari, invia una richiesta ECHO al server e alla mancata risposta del server chiama il metodo di callback per avvisare che la connessione è stata persa.

```
void setEchoCheck(){
    echoCheckExecutorService = Executors.newSingleThreadScheduledExecutor();
    echoCheckExecutorService.scheduleAtFixedRate(() -> {
        if(!echoCheckReceived.get()){
            callback.onConnectionLost();
            echoCheckExecutorService.shutdown();
            return;
        }
        echoCheckReceived.set(false);
        sendEcho();
    }, 2, 2, TimeUnit.SECONDS);
}
```

Infine, sono state implementate 5 classi per automatizzare e facilitare le comunicazioni con il server, dedicando a ognuna di queste classi l'assolvimento di una specifica funzionalità.

Le classi in questione sono:

- ECMAccelerometerThresholdGetter: recupera il valore soglia dal server;
- ECMHoleEventSaver: si occupa di salvare le nuove buche;
- ECMNearHoleEventFinder: recupera le buche nelle vicinanze di una specifica posizione;
- ECMUserLogger: si occupa del login dell'utente;
- ECMUserRegister: si occupa di registrare un nuovo utente.



CD 1 - Classi del client descritte tramite Class Diagram

# Capitolo 5

## Guida d'uso

### 5.1 INTRODUZIONE

Dopo aver descritto in maniera più dettagliata anche il client, si passa ad una semplice e breve guida d'uso sia per il server che per il client.

### 5.1 SERVER

Per compilare il server bisogna usare il seguente comando di compilazione da terminale:

```
gcc -Wall -Wextra -g -pthread -o main.run main.c ecmprotocol.c  
ecmdb.c threadpool.c -I/usr/include/postgresql/  
-L/usr/lib/postgresql/14/lib/ -lpq
```

Per avviare il programma, invece:

```
./main.run [argument] <option> ...
```

dove i possibili argomenti sono:

- `-t` : numero di thread da assegnare al pool per la gestione delle connessioni-client (*default 4*);
- `-q` : grandezza della coda del pool di thread (*default 20*);
- `-a` : valore soglia dell'accelerometro (*default 10.0*).


**Esempi di avvio:**


- `./main.run -t 4 -q 20 -a 10,0`
- `./main.run`





## 5.3 CLIENT


Dopo aver fatto la registrazione ed aver effettuato correttamente l'accesso


 Name

 Surname

 Username


 Email


 Password


 Confirm Password

REGISTER

Password must contain 8 characters and a symbol like .,-\_

 Username

 Password



☐ Remember me

LOGIN

NEW ACCOUNT

*Screen 1 - Registrazione*

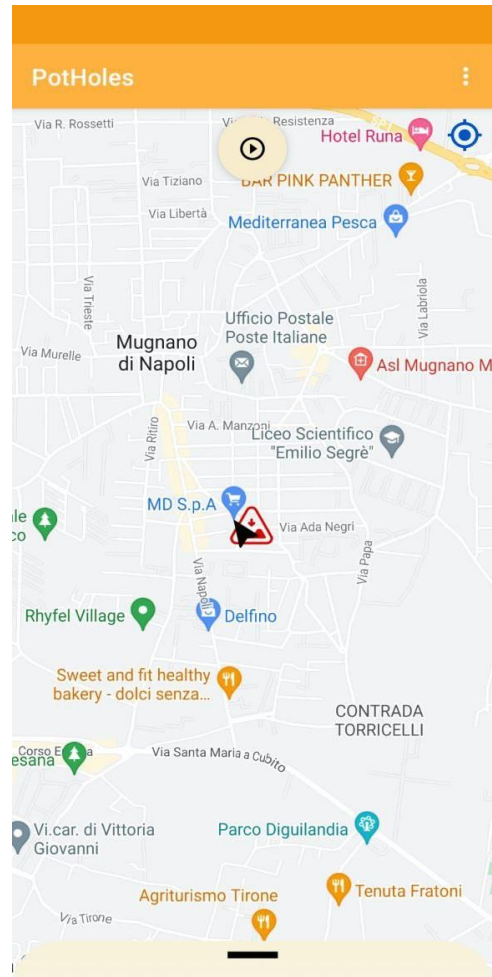
*Screen 2 - Login*

L'utente dovrà decidere se garantire o meno i permessi di localizzazione prima di continuare (*Screen 3*).

Una volta garantiti, l'utente troverà la sua posizione su una mappa (*Screen 4*).



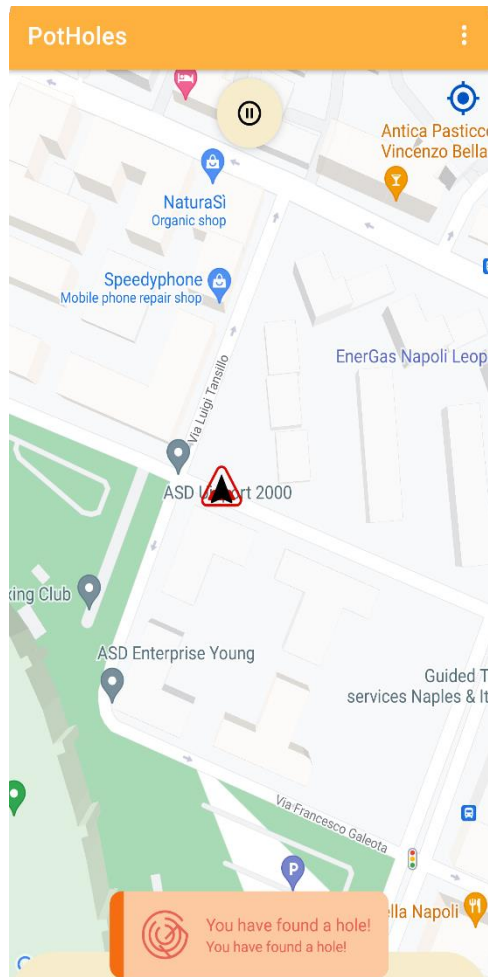
*Screen 3 - Permessi localizzazione*



*Screen 4 - Schermata principale*

L'utente qui può iniziare una sessione di registrazione per segnalare (nel caso ci fossero) delle buche semplicemente cliccando sul pulsante in alto di registrazione (*Screen 5*).

Sarà anche possibile visualizzare le buche più vicine scorrendo dal basso verso l'alto (*Screen 6*).



**Screen 5 - Pulsante di registrazione e buca trovata**



**Screen 6 - Visualizzazione buche vicine**

Nel caso in cui i permessi di localizzazione non vengano garantiti l'utente sarà in grado di tenere premuto sulla mappa per aggiornarla e visualizzare le buche più vicine.



*Screen 7 - Permessi non garantiti*