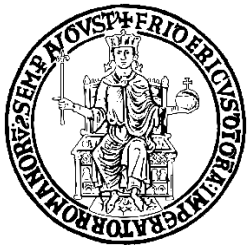
UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

SCUOLA POLITECNICA E DELLE SCIENZE DI BASE

DIPARTIMENTO DI INGEGNERIA ELETTRICA E TECNOLOGIE DELL'INFORMAZIONE



CORSO DI LAUREA IN INFORMATICA

INSEGNAMENTO DI LABORATORIO DI SISTEMI OPERATIVI

ANNO ACCADEMICO 2021/2022

Potholes

Gruppo LSO_2122_54

Andrea PEPE

MATRICOLA N86003197

andrea.pepe2@studenti.unina.it

Marcello RUSSO

MATRICOLA N86003235

marcello.russo@studenti.unina.it

Sommario

Introduzione

Richiesta

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

Il sistema deve memorizzare e rendere disponibile una lista di eventi generati dai clients riguardanti la rilevazione di una un repentino cambiamento dell'accelerazione lungo l'asse verticale che superi una soglia comunicata dal server in fase di connessione. Ciascun client è identificato da un nickname, scelto dall'utente.

Cosa consente di fare il client

Il client android sviluppato consente di:

- impostare un nickname di identificazione;
- collegarsi al server per ricevere i valori di soglia;
- registrare gli eventi alla ricerca di una particolare accelerazione lungo l'asse verticale;
- inviare al server i dati registrati per consetirne il salvataggio;
- comunicare con il server per ricevere una lista dei dati memorizzati.

Server: Guida alla compilazione e all'uso

Tecnologie

Il server è composto da tre elementi fondamentali: il server effettivo, la base di dati e il server setup. Per la memorizzazione abbiamo utilizzato SQLite3, un database locale memorizzato all'interno di un file, nel nostro caso il DB si trova nella cartella *Database*.

Il server setup è un piccolo programma in C il cui scopo è quello di inizializzare la base di dati o crearla se non presente. Infine, abbiamo il server effettivo che analizzeremo a breve.

Requisiti

Affinché sia possibile procedere con l'installazione è necessario rispettare i seguenti requisiti:

- avere un ambiente Unix;
- avere SQLite3 installato;
- compilatore GCC installato.

Compilazione ed esecuzione

Una volta clonato il repository nella destinazione di interesse possiamo procedere in due modi. Il primo è procedere manualmente alla compilazione dei file e la successiva esecuzione; il secondo è utilizzare lo script Bash messo a disposizione per automatizzare il processo.

Metodo #1

Il primo passo è compilare il server setup

```
>gcc Setup/setup.c -o potholes_server_setup –lsqlite3
```

Successivamente eseguire l'output ricevuto

```
>./potholes_server_setup
```

Una volta completato il setup possiamo procede alla compilazione ed esecuzione del server

```
>gcc src/* -o potholes_server -lsqlite3 –pthread -lm
```

Poi

```
>./potholes_server
```

Metodo #2

Come anticipato abbiamo messo a disposizione uno script per velocizzare il processo di installazione e reiniziallizazione dell'app.

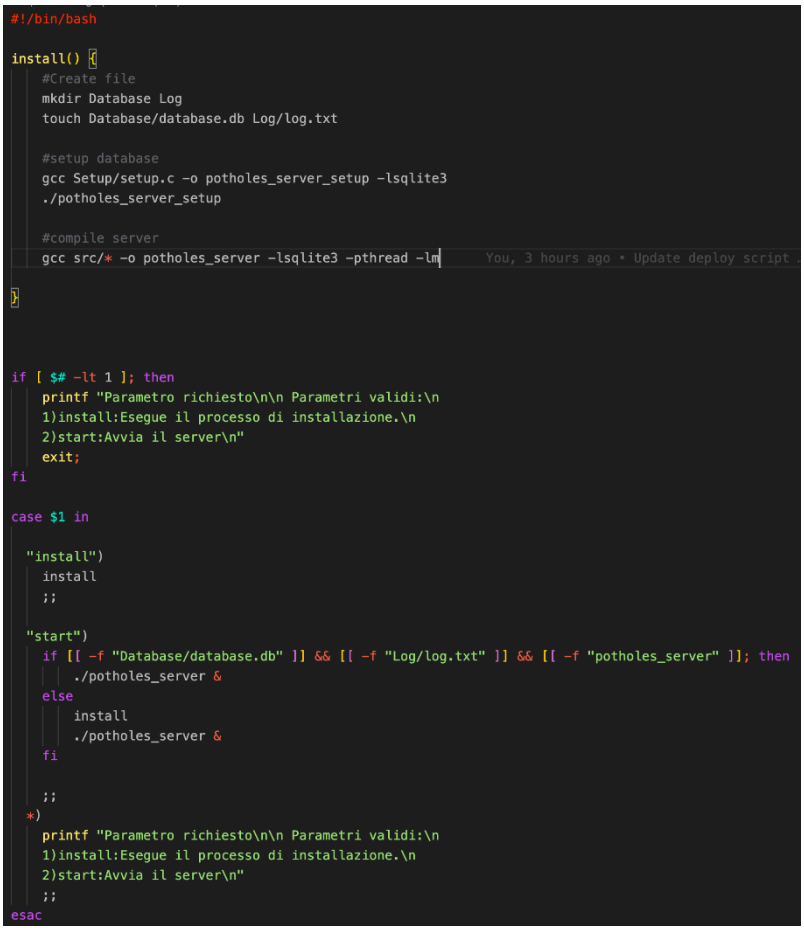
Per lanciare lo script è necessario eseguire il seguente comando

```
>./deploy.sh comando
```

comando può assumere due valori:

- *install*: esegue il processo di installazione (o di reinstallazione);
- *start*: avvia il server effettuando prima un controllo per verificare che i file necessari siano presenti, se non lo sono effettua l'installazione.

Di seguito lo script deploy.sh



```bash
#!/bin/bash

install() {
    #Create file
    mkdir Database Log
    touch Database/database.db Log/log.txt

    #setup database
    gcc Setup/setup.c -o potholes_server_setup -lsqlite3
    ./potholes_server_setup

    #compile server
    gcc src/* -o potholes_server -lsqlite3 -pthread -lm     You, 3 hours ago • Update deploy script

}



if [ $# -lt 1 ]; then
    printf "Parametro richiesto\n\n Parametri validi:\n
    1)install:Esegue il processo di installazione.\n
    2)start:Avvia il server\n"
    exit;
fi

case $1 in

  "install")
    install
    ;;

  "start")
    if [[ -f "Database/database.db" ]] && [[ -f "Log/log.txt" ]] && [[ -f "potholes_server" ]]; then
        ./potholes_server &
    else
        install
        ./potholes_server &
    fi

    ;;
  *)
    printf "Parametro richiesto\n\n Parametri validi:\n
    1)install:Esegue il processo di installazione.\n
    2)start:Avvia il server\n"
    ;;
esac
```

Protocollo di comunicazione

Nella seguente sezione analizziamo come avviene la comunicazione tra client e server e come quest'ultima viene gestita da entrambe le parti.

Come anticipato il server è stato realizzato in linguaggio C, il quale espone i suoi servizi mediante delle socket. In particolare, abbiamo optato per una comunicazione orientata alla connessione e non permanente. Tale scelta è stata necessaria principalmente per essere certi che le informazioni venissero scambiate correttamente tra le parti e per ridurre al minimo il

tempo di connessione. Questo significa dunque che il client apre una connessione con il server solo nel momento in cui ha bisogno di effettuare uno scambio di dati con quest'ultimo. Sulla base di quanto indicato, riportiamo di seguito la system call utilizzata per l'apertura della socket:

```
socket(AF_INET, SOCK_STREAM, 0)
```

Da notare i parametri utilizzati:

- *AF_INET*: selezione del dominio IPv4 e IPv6;
- *SOCK_STREAM*: selezione della tipologia di socket TCP;
- il parametro *Protocol* è impostato a *0* in modo tale che venga usata la configurazione di default del sistema per la combinazione dominio-tipo.

Passando al lato client, si tratta di una app nativa Android scritta in linguaggio Java. Per questo motivo ci siamo serviti della classe built-in di Java Socket per aprire la connessione con il server. Tale classe, inoltre, consente di accedere agli oggetti *BufferedReader* e *OutputStream* per poter eseguire le operazioni di lettura e scrittura, dunque invio e ricezione dei dati. Infine, descriviamo di seguito la formattazione che abbiamo adottato per i messaggi per uno scambio efficiente dei dati. Dal momento che le socket lavorano per byte è stato necessario implementare un sistema che fosse in grado di semplificare l'interpretazione questi ultimi. Ispirandoci alla formattazione dei file CSV, abbiamo applicato un meccanismo simile per separare ogni campo, vale a dire attraverso il delimitatore ";". Così facendo, ciò che succede sia lato client che server è che il messaggio viene tokenizzato, cioè viene separato in più parti e analizzate singolarmente,

semplificando il processo di validazione sia del messaggio che dei dati effettivi.

Implementazione server

Nella seguente sezione andiamo a descrivere l'intera implementazione del server, come è stato strutturato e come viene gestita l'erogazione dei servizi.

Iniziamo definendo il workflow del server elencando le varie fasi per poi analizzarle nel dettaglio:

1. Apertura socket
2. Configurazione spazio indirizzi
3. Binding della socket
4. Impostare la socket in modalità di ascolto
5. Accettare connessione dal client
6. Gestire richiesta

Apertura socket

Il primo step del server è quello di creare la socket ed inizializzarla impostando i valori di dominio e tipologia. In questa fase ci serviamo della system call *socket*.

```
//! Create socket
if ((server_sd = socket(AF_INET, SOCK_STREAM,0)) == 0) {
    log_e("socket","Errore durante la creazione del socket");
    exit(EXIT_FAILURE);
}else{
    log_m("Socket Success", "Socket creata con successo");
}
```

Configurazione spazio indirizzi

Successivamente, andiamo a popolare la struttura dati che gestisce lo spazio degli indirizzi della socket.

```
//! Set address
address.sin_family = AF_INET;
address.sin_port = htons(PORT);
address.sin_addr.s_addr = htonl(INADDR_ANY);
```

Le righe di codice sopra riportate configurano rispettivamente: lo spazio di indirizzi IPv4, il numero di porto da associare alla socket (nel nostro caso 8585) e lo spazio di indirizzi accettati in ingresso, vale a dire qualunque.

Binding della socket

La fase di binding configura la socket creata in prima battuta con i parametri impostati nella struttura dati degli indirizzi, in modo da renderla utilizzabile.

```
//! Binding socket
if (bind(server_sd, (struct sockaddr*)&address,addrlen)< 0) {
    log_e("bind","Errore durante il binding");
    exit(EXIT_FAILURE);
}else{
    log_m("Bind Success", "Bind eseguito con successo");

}
```

Impostare la socket in modalità di ascolto

Questo passaggio abilita la socket all'ascolto, definendo inoltre il numero massimo di client che può gestire.

```
//!Set socket on listening
if (listen(server_sd,15) < 0) {
    log_e("listen","Errore durante l'ascolto");
    exit(EXIT_FAILURE);
}else{
    log_m("Listen Success", "Listen eseguito con successo");
}
```

Accettare connessione dal client

La system call usata in questa fase consente alla socket di comunicare effettivamente con il client connesso. Dal momento che la nostra socket deve rimanere attiva in fase di accettazione del client, abbiamo chiuso la funzione in un ciclo *while*, in questo modo potrà accettare nuove connessioni fino a quando il server non verrà spento.

```
//! Accept connetion by client.
while ((client_sd = accept(server_sd,(struct sockaddr*)&client_address, (socklen_t*)&client_addrlen)) != -1) {
```

Gestire richiesta

```
//! Get client ip
char client_ip[25];
if(inet_ntop(AF_INET, &client_address.sin_addr, client_ip, sizeof(client_ip))) {…

//! Setup Thread
thread_attr_status = pthread_attr_init(&thread_attr);
if(thread_attr_status!=0)
    log_e("pthread_attr_init", "Errore durante l'iniziallizzazione degli attributi");
else
    log_m("pthread_attr_init", "Attributi inizializzati con successo");

//?Set Thread Detach State
thread_attr_status = pthread_attr_setdetachstate(&thread_attr, PTHREAD_CREATE_DETACHED);
if(thread_attr_status!=0)
    log_e("pthread_attr_setdetachstate", "Errore durante l'impostazione dello stato detach");
else
    log_m("pthread_attr_setdetachstate", "Stato detach impostato");

//? Memory allocation for socket descriptor
thread_client_sd = malloc(sizeof(int));
*(thread_client_sd) = client_sd;

//! Start request thread
create_request_thread_status = pthread_create(&requets_thread, &thread_attr, requestDispatcher, (void *) thread_client_sd);

if(create_request_thread_status < 0)
    log_e("pthread_create", "Errore durante la creazione di un nuovo thread");
else
    log_m("pthread_create", "Thread creato con successo");

pthread_attr_destroy(&thread_attr);
```

La prima operazione che viene effettuata in questo blocco di codice non appena il client si connette alla socket è quella di recupero e tracciamento

dell'indirizzo IP del dispositivo connesso, in modo da poter loggare le operazioni svolte.

In seguito, procediamo alla configurazione degli attributi con i quali configureremo il nostro thread. Nello specifico, inizializziamo gli attributi e impostiamo la modalità *detached*. Una volta che gli attributi sono pronti procediamo con l'allocare dinamicamente un nuovo socket descriptor copiando al suo interno il contenuto della socket descriptor del client ricevuta dalla funzione *accept*.

Questo passaggio è fondamentale in quanto consente al server di smistare su un thread secondario la gestione della suddetta richiesta e procedere all'elaborazione di una altra. Questo si traduce in un sistema in cui non vi è attesa per il client in quanto le operazioni vengono gestite contemporaneamente.

La system call *pthread_create* crea il nuovo thread in modo tale che esegua la funzione *requestDispatcher* il cui ruolo è quello collegarsi al DB, analizzare il messaggio del client e indirizzare la socket verso la corretta operazione.

```c
//? Callback di partenza del thread
void *requestDispatcher(void *thread_client_sd) {

    //? Set client socket descriptor
    int client_sd = *(int *) thread_client_sd;
    //? Socket Buffer
    char buffer[MAX_BUFFER] = { 0 };
    bzero(buffer,sizeof(buffer));
    //? SQLite DB
    sqlite3 *database;
    int sqlite3_status;

    //! Open db
    sqlite3_status = sqlite3_open_v2("Database/database.db", &database, SQLITE_OPEN_READWRITE |SQLITE_OPEN_NOMUTEX, NULL);

    if(sqlite3_status != SQLITE_OK){
        send(client_sd, "Impossibile collegarsi al Database\n", strlen("Impossibile collegarsi al Database\n"), 0);
        log_e("sqlite3_open_v2","Impossibile collegarsi al database");
        return NULL;
    }
    log_m("sqlite3_open_v2","Connessione al database avvenuta con successo");

    log_m("Thread start", "Request Thread avviato");

    //! Read Action Request
    if(recv(client_sd, buffer, sizeof(buffer),0)==-1)
        log_e("recv error","Errore durante la riczione della richiesta");
    else{
        log_m("Richiesta ricevuta dal client",buffer);

        //! Dispatch Action
        if(strcmp(buffer, "getAllPotholes") == 0){
            getAllPotholes(client_sd, database);
        }else if(strcmp(buffer, "getNearPotholes") == 0){
            getNearPotholes(client_sd, database);
        }else if(strcmp(buffer, "insertPotholes") == 0){
            insertPotholes(client_sd, database);
        }else if(strcmp(buffer, "getThreshold") == 0){
            getThreshold(client_sd, database);
        }else{
            log_e("404", "Action non gestibile dal server");
            send(client_sd, "ERROR 404 - Action non gestibile\n", strlen("ERROR 404 - Action non gestibile\n"), 0);
            sleep(2);
        }
    }

    //! Free Socket memory allocation
    free(thread_client_sd);
    //! Close client connection socket
    close(client_sd);
    return NULL;
}
```

Logging e SignalHandler

Per concludere la trattazione del server parliamo del sistema di logging e la gestione dei segnali per l'arresto del server.

Per tener traccia di quanto accade durante l'esecuzione del programma, abbiamo realizzato una variante semplificata del sistema di logging integrato

in Android. In particolare, abbiamo ricopiato i metodi *Log.i* e *Log.e* per indicare rispettivamente un messaggio normale ed un messaggio di errore. Tali messaggi vengono salvati in un file di log in modo da non perderne traccia.

```
char *logfile = "Log/log.txt";

void log_m(char *tag, char *message){

    char toWrite[MAX_LOG];
    int file = open(logfile,O_RDWR|O_CREAT|O_APPEND,S_IRWXU);
    if (file != -1){
        sprintf(toWrite,"%s: %s\n",tag,message);
        write(file,toWrite,strlen(tag)+strlen(message)+3);
        close(file);
    }
}

void log_e(char *tag, char *message){

    char toWrite[MAX_LOG];
    int file = open(logfile,O_RDWR|O_CREAT|O_APPEND,S_IRWXU);
    if (file != -1){
        sprintf(toWrite,"ERRORE - %s: %s\n",tag,message);
        write(file,toWrite,strlen(tag)+strlen(message)+12);
        close(file);
    }
}
```

Infine, la gestione dei segnali consente di acquisire i segnali USR1, USR2 e INT e terminare correttamente l'esecuzione.

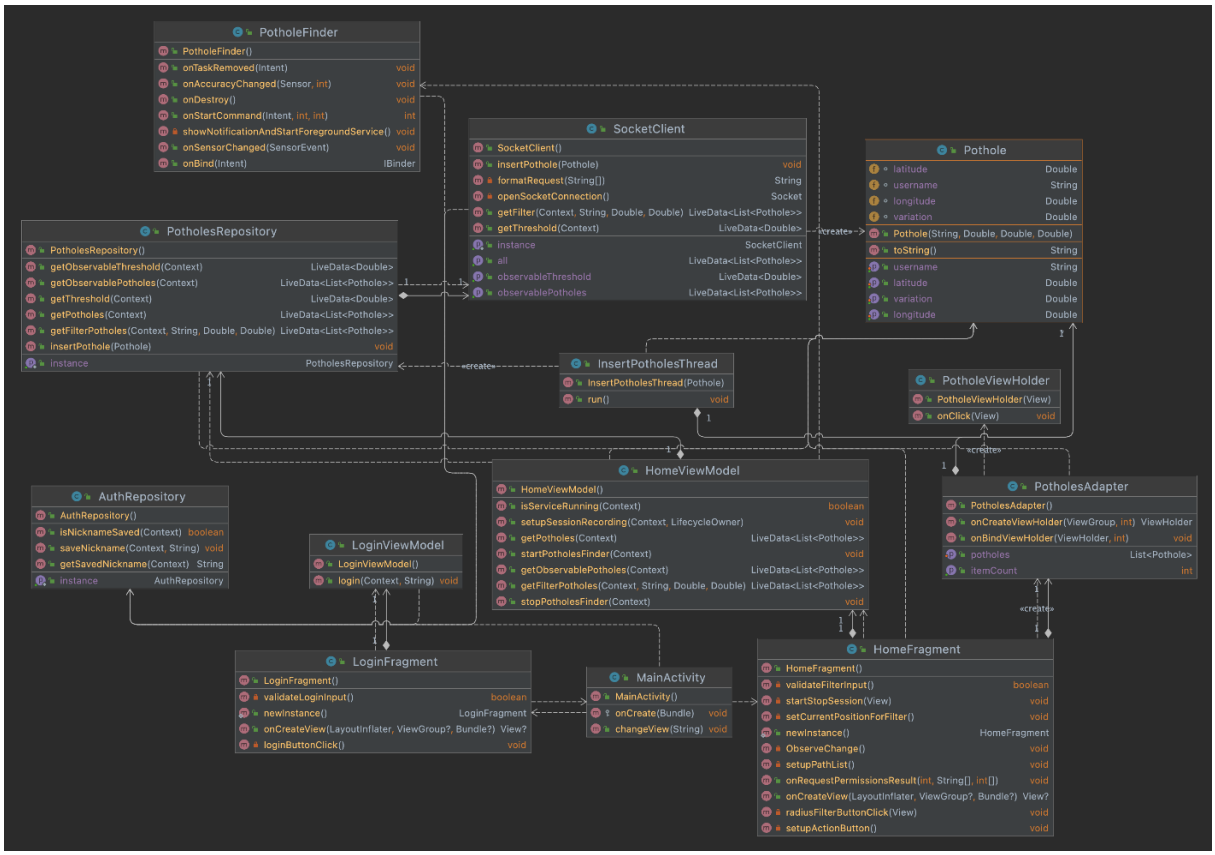
Implementazione client

Il client è costituito da una app nativa Android scritta in linguaggio Java. Tale app è stata realizzata adottando il design pattern MVVM (Model-View-ViewModel) in quanto tale pattern consente di realizzare un prodotto estremamente flessibile e manutenibile.

Struttura

L’app è strutturata in modo tale da avere una sola activity, la quale si occupa di gestire i fragment in essa ospitati. In particolare, i fragment sono due, il primo consente di impostare un nickname che verrà poi salvato nella memoria del dispositivo, il secondo consente di svolgere tutte le funzioni richieste di Potholes.

Di seguito il class diagram dell’intera app Android.



Classe di connessione

La classe responsabile della connettività con il server è situata nel Package *DataAccess.SocketClient*.

Al suo interno sono presenti i metodi essenziali per gestire la comunicazione con la socket ospitata dal server ed eseguire le operazioni richieste. Al suo interno, inoltre, risiedono i LiveData, i quali consentono all'app di lavorare in multithreading e non creare rallentamenti lato UI.

Di seguito mostriamo una porzione di codice di tale classe. In particolare, il metodo di apertura della connessione ed il metodo che si occupa della ricezione di Potholes entro un dato raggio, in quanto al suo interno vi sono sia operazioni di lettura che di scrittura verso la socket.

```
private Socket openSocketConnection(){
    Socket socket = null;
    try {
        socket = new Socket(server_address,server_port);
        socket.setSoTimeout(6000);
        Log.i( tag: "Socket creata", msg: "Socket connessa correttamente");
        return socket;
    } catch (IOException e) {
        Log.e( tag: "Errore Socket", msg: "Non è stato possibile stabilire una connessione con la socket");
        e.printStackTrace();
    }
    return socket;
}
```

```
public LiveData<List<Pothole>> getFilter(Context context, String radius, Double latitude, Double longitude) {
    String msg = "getNearPotholes",result;
    List<Pothole> resultList = new ArrayList<>();
    try{
        Socket socket = this.openSocketConnection();
        BufferedReader reader =
                new BufferedReader(
                        new InputStreamReader(socket.getInputStream()));
        socket.getOutputStream().write(msg.getBytes());
        Thread.sleep( millis: 2000);

        String request = this.formatRequest(AuthRepository.getInstance().getSavedNickname(context),
                                            String.valueOf(latitude),String.valueOf(longitude),radius);
        socket.getOutputStream().write(request.getBytes());
        Thread.sleep( millis: 2000);

        while(reader.ready()) {
            result = reader.readLine();
            String[] tokens = result.split( regex: ";");
            resultList.add(new Pothole(tokens[0],Double.valueOf(tokens[1]),Double.valueOf(tokens[2]),Double.valueOf(tokens[3])));
        }

        socket.close();
        potholesList.postValue(resultList);
    }catch (Exception e){
        Log.e( tag: "Errore comunicazione con il socket", msg: "Impossibile dialogare con il socket");
        e.printStackTrace();
    }

    return potholesList;
}
```

Multithreading

Nell'ultima sezione di questo documento, mostriamo come abbiamo reso l'app multithreading in modo da renderla estremamente reattiva e non creare mai rallentamenti.

Lavorando con il pattern MVVM abbiamo potuto sfruttare le potenzialità dei LiveData in modo da poter registrare un comportamento automatico dell'app ad ogni aggiornamento del set di dati. Per rendere completamente asincrona la parte di fetch dei dati e non bloccare la UI in attesa di una risposta, abbiamo fatto modo che ogni operazione di rete avvenisse su thread secondari in background, di seguito un esempio.

```
new Thread(new Runnable() {
    @Override
    public void run() { mViewModel.getPotholes(getContext()); }
}).start();
```

Inoltre, abbiamo realizzato anche la parte di registrazione del percorso su thread secondari. Nello specifico la gestione del sensore è stata affidata ad un service che viene eseguito in background anche quando l'app è sospesa, in questo modo una volta avviata la registrazione è possibile mettere in standby il dispositivo. All'interno del service, a sua volta, ogni qual volta viene rilevata la buca, la comunicazione con la socket *Client* viene affidata ad un ulteriore thread in modo da non creare attesa nel service.