

# UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II



CORSO DI LABORATORIO DI SISTEMI OPERATIVI  
DIPARTIMENTO DI INGEGNERIA ELETTRICA E TECNOLOGIE  
DELL'INFORMAZIONE

## POTHOLES

<LSO\_2122\_42>

---

### Candidati:

Luca Bianco - N86003200  
Antonio Garofalo - N86003129

ANNO ACCADEMICO 2021/2022

# Indice

<b>1</b>	<b>Introduzione</b>	<b>2</b>
1.1	Cosa offre POTHOLEES? . . . . .	2
1.2	Tecnologie adoperate . . . . .	2
<b>2</b>	<b>Guida alla compilazione e all'uso</b>	<b>3</b>
2.1	Inizializzazione del server . . . . .	3
2.1.1	Compilazione . . . . .	3
2.1.2	Esecuzione . . . . .	3
2.2	Avvio del server . . . . .	3
<b>3</b>	<b>Protocollo</b>	<b>5</b>
<b>4</b>	<b>Dettagli implementativi - Server</b>	<b>7</b>
4.1	Struttura . . . . .	7
4.2	Test . . . . .	10
<b>5</b>	<b>Dettagli implementativi - Client</b>	<b>11</b>
5.1	Thread Centering . . . . .	11
5.2	OSM - Open Street Map . . . . .	12
5.3	Diagramma di classi . . . . .	13
5.4	Networking . . . . .	14
<b>6</b>	<b>Mockup</b>	<b>19</b>
<b>7</b>	<b>Artefatti</b>	<b>20</b>
7.1	State Chart . . . . .	20
7.2	Activity Diagram . . . . .	21
7.3	Sequence diagram . . . . .	22

# 1 Introduzione

L'applicativo che ci è stato richiesto di sviluppare consiste in un sistema **Client-Server** che come da traccia:

*"Consenta la raccolta e l'interrogazione di informazioni riguardanti la presenza di irregolarità (buche) su di una superficie."*

## 1.1 Cosa offre POTHOLE?

L'applicativo è un sistema semplice basato su un'interrogazione lato client e una response lato server, in particolare le funzionalità sono le seguenti:

- 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 e il valore del cambiamento ogni volta che registra un nuovo evento.
- Mostrare all'utente la lista di tutti gli eventi registrati dal server in un certo raggio dalla propria posizione.
- Permettere all'utente di visualizzare gli eventi vicini su mappa.

## 1.2 Tecnologie adoperate

Per la realizzazione del progetto abbiamo come prima istanza creato una macchina virtuale sulla piattaforma cloud **Microsoft Azure**, la macchina virtuale monta un sistema operativo Linux con distro **Ubuntu Server 20.04**.

Il server (sviluppato in linguaggio C) è stato caricato sulla macchina virtuale e tramite protocollo TCP/IP mette a disposizione i servizi prima elencati, per la precisione il server compie le operazioni di interrogazione e salvataggio su di un file database SQLite (**.db**).

Il client (sviluppato in linguaggio Java su Android) sfrutta le più moderne system call e librerie di supporto per interfacciarsi in modo semplice con il server remoto.

Source vari: [\[1\]](#) [\[2\]](#) [\[3\]](#) con integrazione delle slide del corso 2021/22.

## 2 Guida alla compilazione e all'uso

Il server è strutturato da 6 file totali, 3 file con estensione .c e 3 file con estensione .h, per facilitare la build del server abbiamo creato un makefile che facilmente può essere usato per compilare ESCLUSIVAMENTE" il server.c e il file utils.c dove abbiamo raccolto la maggior parte delle chiamate e servizi del server.

### 2.1 Inizializzazione del server

#### 2.1.1 Compilazione

Per compilare il file starter è necessario sulla console BASH (nota bene: bisogna trovarsi nella directory "Server" del progetto):

Listing 1: Compilazione Starter

```
#!/bin/bash
gcc [-o nomeEseguibile] Other/startserver.c Other/utils.c -lsqlite3
```

#### 2.1.2 Esecuzione

Compilato lo starter sarà solo necessario eseguire l'eseguibile creato (se non specificato il nome con -o nomeEseguibile di default sarà a.out):

Listing 2: Esecuzione Starter

```
#!/bin/bash
./nomeEseguibile
```

### 2.2 Avvio del server

Come annunciato prima, per il server è stato creato un makefile apposito per compilazione, memory check e avvio, infatti per il server basta eseguire il seguente comando in BASH:

Listing 3: Server makefile

```
#!/bin/bash
make makefile clean all [valgrind || start]
```

```

CC = gcc
CFLAGS = -O2 -Wall
PTHREAD_FLAG = -pthread
SQL_FLAG = -lsqlite3
LDLIBS = -lm
OBJECTFILES =
TARGET = server_pothole
SRC = server.c Other/utils.c

all: $(TARGET)
    @printf "\e[32m\b\e[0m" "\n\tFile Creation...\n"

$(TARGET): $(OBJECTFILES)
    $(CC) $(CFLAGS) -o $(TARGET) $(OBJECTFILES) $(LDLIBS) $(SRC) $(PTHREAD_FLAG) $(SQL_FLAG)

.PHONY: depend clean

clean:
    @printf "\e[32m\b\e[0m" "\n\tClean...\n"
    rm -f $(OBJECTFILES) $(TARGET)

start:
    @printf "\e[32m\b\e[0m" "\n\tStarting Server...\n"
    ./server_pothole

valgrind:
    @printf "\e[32m\b\e[0m" "\n\tValgrind MemCheck...\n"
    valgrind --leak-check=full \
        --show-leak-kinds=all \
        --track-origins=yes \
        --verbose \
        --log-file=valgrind-out.txt \
        ./server_pothole

```

Figura 1: Makefile

struttura:

- clean: pulizia di eventuali file inutili.
- all: compilazione (anche utilizzabile a se stante) dei file.
- start: start del server.
- valgrind: start del server con alla chiusura un file .txt che elenca eventuali memory leak, quindi uno start con memorycheck a runtime.

### 3 Protocollo

Questa sezione è dedicata alla struttura della comunicazione client server che abbiamo deciso di adoperare. Questa sezione verrà divisa in due parti: una dedicata al Client e un'altra dedicata al Server.

**Iniziamo con la parte Server:** Il server inizia con la creazione di una socket, questa socket permetterà di avere una lista di richieste che arriveranno dai client e verranno man mano soddisfatte. Come primo passo la socket che abbiamo creato è una socket che ha le seguenti caratteristiche:

- Dominio: AF\_INET per l'utilizzo d'Ipv4 e Ipv6;
- Tipo: SOCK\_STREAM quindi basata sul protocollo TCP per avere un tipo di connessione connection-oriented.
- Protocol: 0

Abbiamo deciso di fare in modo che la comunicazione client server avvenga in modo tale da non avere una connessione permanente e fissa tra questi due.

Quindi il nostro obiettivo è stato quello di fare in modo che il client richiedesse servizi al server solo quando ne aveva bisogno e quindi che a ogni richiesta venisse aperta una nuova connessione, per poi essere chiusa subito dopo averla soddisfatta. Dopodiché il server continua con la creazione classica di una socket mettendosi in ascolto grazie a una listen con una coda di attesa non più grande di 10, per poi effettuare una accept in un ciclo while.

A ogni richiesta accettata il server effettua tre semplici passaggi:

1. Crea una nuova socket.
2. Crea un thread.
3. Passa al nuovo thread la nuova socket Grazie a questa implementazione.

abbiamo la socket principale che si occupa di accettare le richieste da parte dei client per poi smistarle su altri thread con una nuova socket dedicata a quella specifica connessione, così da avere un server che è in grado di soddisfare più richieste in parallelo e non iterativamente rischiando di avere problemi di lentezza in caso di molte richieste al server.

Ogni thread è in grado di soddisfare le richieste da parte del client perché grazie alla nuova socket, che gli è stata passata, possiede anche l'indirizzo IP del client con cui comunicare. Il primo step che compie ogni thread è quello di leggere, grazie a una chiamata alla funzione read, qual è il servizio che

il client vuole effettuare, quindi la stringa di caratteri proveniente dal client viene confrontata ed eventualmente la richiesta viene smistata nuovamente a una opportuna funzione. Se il thread ha bisogno di ricevere dei parametri da parte del client, allora grazie a una chiamata alla funzione send con il messaggio “START r”, informa il client che è pronto per ricevere i dati, subito dopo effettua una chiamata a recv così da ricevere i parametri. Una volta ricevuti i parametri questi saranno formattati in modo tale che i dati siano separati dal separatore “:”. Per quanto riguarda una eventuale risposta da parte del server, allora allo stesso modo ogni record di dati viene inviato separandoli con “:” aggiungendo alla fine di ogni record anche “r”.

Al termine il server invia una send con messaggio “END” per segnalare al client che non ci sono più dati da comunicare. Ogni thread, una volta soddisfatta la richiesta, rilascia tutte le risorse che stava utilizzando prima di terminare effettuando prima una close sul database che stava utilizzando, poi dealloca la socket passatagli, e infine effettua una close sulla socket descriptor.

**Da parte del Client:** Il client ovviamente è visto da una prospettiva più ad alto livello, quindi a ogni operazione che viene richiesta dall’utente viene effettuata una nuova connessione al server, questa connessione viene creata grazie a una nuova Socket ed effettuando una chiamata al metodo openConnection che connette la socket al corretto indirizzo IP e alla corretta porta. A differenza del server, nel client, per scambiare dati con il server abbiamo bisogno di due buffer diversi, quindi se bisogna scrivere dati da mandare al server questo viene fatto chiedendo un oggetto di tipo OutputStream alla socket e grazie al metodo write possiamo scrivere i nostri dati in questo buffer (ancora separati tramite “:”) a patto che siano, ovviamente, Bytes.

Per quanto riguarda la lettura dei dati da parte del Server questo viene effettuato grazie a un oggetto di tipo BufferedReader, ricavato tramite la socket, che permette di leggere fino a quando il server non invia il messaggio di “END”, una volta arrivati i dati devono essere estratti separandoli dai “:”.

Al termine dello scambio di messaggi il client chiude la connessione grazie a una chiamata alla funzione close della socket.

## 4 Dettagli implementativi - Server

Il server è sviluppato in C in particolare con le seguenti regole implementative:

- Client e server devono comunicare tramite socket TCP o UDP.
- Oltre alle system call UNIX, il server può utilizzare solo la libreria standard C.
- Il server deve essere di tipo concorrente, ed in grado di gestire un numero arbitrario di client contemporaneamente.
- Il server effettua il log delle principali operazioni (nuove connessioni, sconnessioni, richieste da parte dei client) su standard output.

### 4.1 Struttura

La struttura del server è sviluppata da un **inizializzazione** del server (creazione della **socket** e relativo **bind**), fase di **ascolto** (si aspettano le eventuali richieste e connessioni), creazione di un **thread** di esecuzione per svolgere le chiamate ai servizi, gestione di **signal** e chiusura del server.

Particolare attenzione è richiesta sulla fase di ascolto e quella di thread, infatti abbiamo cercato di unire le due fasi in poche linee di codice, il server resta in ascolto su un while con condizione la SYS CALL **accept** e all'interno entrerà soltanto quando ci sarà una SYS CALL **connect** da parte di un client, il server loggerà il tutto (compreso l'ip del client) e creerà un thread con la funzione **pthread\_create** passando una funzione chiamata **manageRequest** (che vedremo successivamente).



```

50 printf("
51 printf("
52 printf("
53 printf("
54 printf("
55 printf("
56
57 /*Create socket*/
58 sockfd = socket(AF_INET, SOCK_STREAM, 0);
59
60 if(sockfd == -1)
61     logging(tag, "Socket creation error", false);
62 else
63     logging(tag, "Socket creation success", true);
64
65 bzero(&serveraddr, sizeof(serveraddr));
66
67 /*Assigning IP, Port and sin_family */
68 serveraddr.sin_family = AF_INET;
69 serveraddr.sin_addr.s_addr = inet_addr(hostname);
70 serveraddr.sin_port = htons(PORT);
71
72 /*Binding*/
73 if(bind(sockfd, (struct sockaddr *) &serveraddr, sizeof(serveraddr)) == -1)
74     logging(tag, "Socket bind error", false);
75 else
76     logging(tag, "Socket bind success", true);
77
78 if(listen(sockfd, 10) == -1)
79     logging(tag, "Listen status: online", true);
80
81 len = sizeof(clientaddr);
82
83 while((connfd = accept(sockfd, (struct sockaddr *) &clientaddr, (socklen_t*)&len)) != -1) {
84
85     /*Init attribute*/
86     err = pthread_attr_init(&attr);
87
88     if(err!=0)
89         logging(tag, "Attribute init error", false);
90
91     /*Set detach state*/
92     err = pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
93
94     if(err!=0)
95         logging(tag, "Setting detach state error", false);
96
97     logging(tag, "Accept success", true);
98
99     personal_socket = malloc(sizeof(int));
100     *(personal_socket) = connfd;
101
102     /*IP logging*/
103     char ip[25];
104     if(inet_ntop(AF_INET, &clientaddr.sin_addr, ip, sizeof(ip))) {
105         char msg[100];
106         sprintf(msg, "Client connect from: %s", ip);
107         logging(tag, msg, true);
108     } else
109         logging(tag, "Invalid Client IP", false);
110
111     /*Start a new manage request*/
112     status_create_thread = pthread_create(&thread, &attr, manageRequest, (void *) personal_socket);
113
114     if(status_create_thread < 0)
115         logging(tag, "Pthread creation error", false);
116
117     pthread_attr_destroy(&attr);
118 }
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133 /*Service selector*/
134 if(strcmp(buffer, "getAll") == 0)
135     getAllPotholesRequest(socket_descriptor, database);
136 else if(strcmp(buffer, "getNear") == 0)
137     getNearPotholesRequest(socket_descriptor, database);
138 else if(strcmp(buffer, "post") == 0)
139     postRequest(socket_descriptor, database);
140 else if(strcmp(buffer, "threshold") == 0)
141     send(socket_descriptor, "17\n", 2, 0);
142 else {
143     logging(tag, "Invalid service", false);
144     send(socket_descriptor, "Invalid service:", 17, 0);
145 }
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999

```

Figura 2: Operazioni

La funzione che compie il thread è molto semplice, inanzitutto si prende il client descriptor in modo da avere un riferimento a tale connessione, successivamente si compie (attraverso una semplice **read**) il controllo su quale operazione il client ha richiesto (FIG. 3), se l'operazione non esiste non accade nulla (Invalid service).

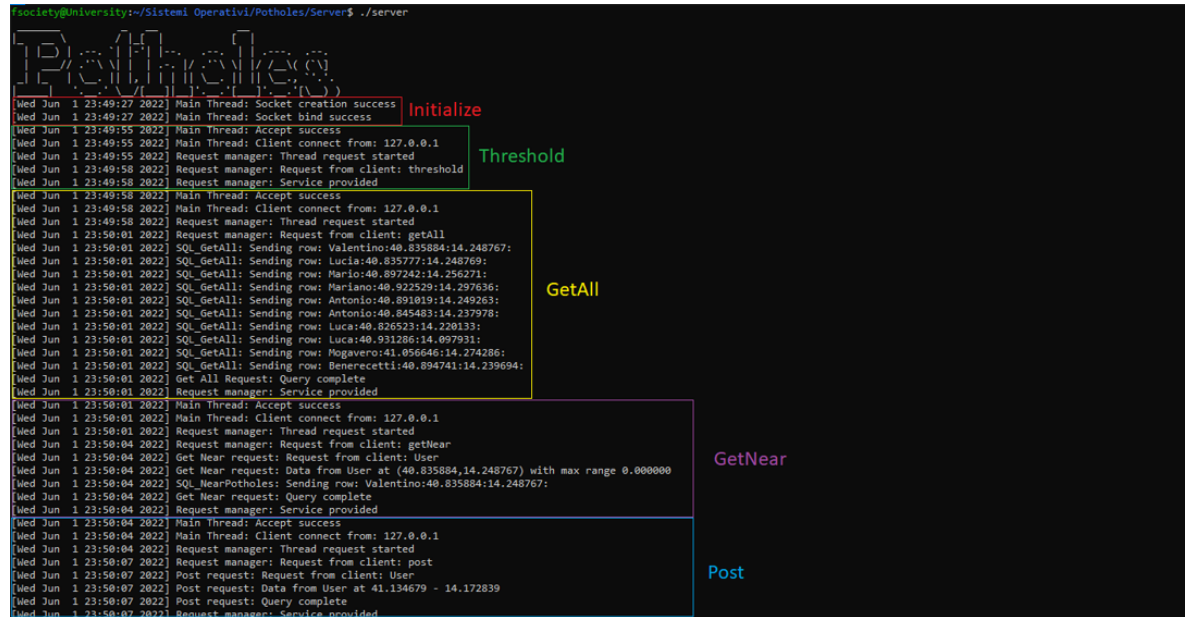
le operazioni chiamano a loro volta funzioni a doc create per il servizio in questione (a parte l'operazione di threshold che compie una semplice **send**).

Ogni operazione formatta l'input del client che può essere:

- Nickname:Latitude:Longitude:Range se viene chiamata una getNear (getAll non necessita di Range).
- Nickname:Latitude:Longitude se viene chiamata una insert.
- Nulla se è chiamata una threshold.

## 4.2 Test

Per facilitare la comprensione degli output del server abbiamo creato una funzione di logging che permette un logging costante per ogni operazione compiuta:



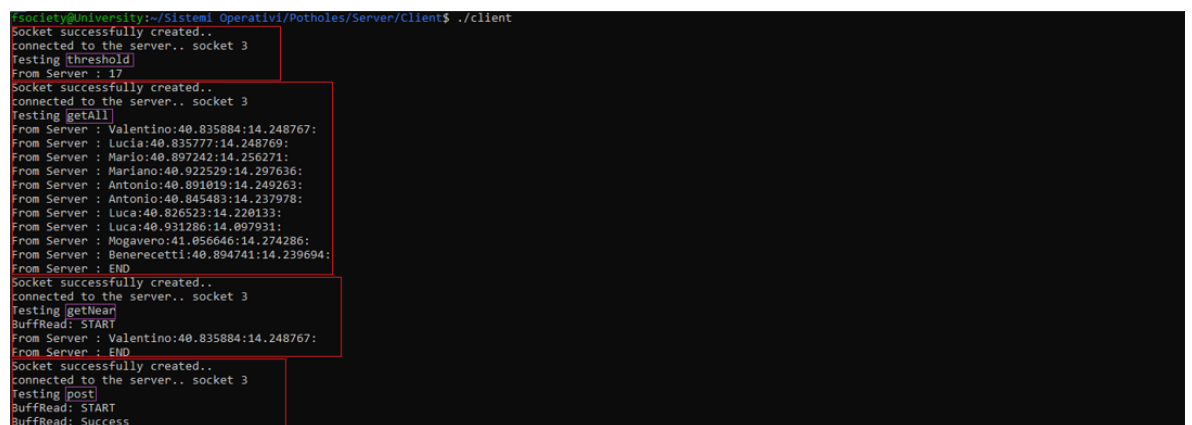
```
fsociety@University:~/Sistemi Operativi/Potholes/Server$ ./server
Potholes
[Wed Jun 1 23:49:27 2022] Main Thread: Socket creation success
[Wed Jun 1 23:49:27 2022] Main Thread: Socket bind success
[Wed Jun 1 23:49:55 2022] Main Thread: Accept success
[Wed Jun 1 23:49:55 2022] Main Thread: Client connect from: 127.0.0.1
[Wed Jun 1 23:49:55 2022] Request manager: Thread request started
[Wed Jun 1 23:49:58 2022] Request manager: Request from client: threshold
[Wed Jun 1 23:49:58 2022] Request manager: Service provided
[Wed Jun 1 23:49:58 2022] Main Thread: Accept success
[Wed Jun 1 23:49:58 2022] Main Thread: Client connect from: 127.0.0.1
[Wed Jun 1 23:49:58 2022] Request manager: Thread request started
[Wed Jun 1 23:50:01 2022] Request manager: Request from client: getAll
[Wed Jun 1 23:50:01 2022] SQL_GetAll: Sending row: Valentino:40.835884:14.248767:
[Wed Jun 1 23:50:01 2022] SQL_GetAll: Sending row: Lucia:40.835777:14.248769:
[Wed Jun 1 23:50:01 2022] SQL_GetAll: Sending row: Mario:40.897242:14.256271:
[Wed Jun 1 23:50:01 2022] SQL_GetAll: Sending row: Mariano:40.922529:14.297636:
[Wed Jun 1 23:50:01 2022] SQL_GetAll: Sending row: Antonio:40.891019:14.249263:
[Wed Jun 1 23:50:01 2022] SQL_GetAll: Sending row: Antonio:40.845483:14.237978:
[Wed Jun 1 23:50:01 2022] SQL_GetAll: Sending row: Luca:40.826523:14.220133:
[Wed Jun 1 23:50:01 2022] SQL_GetAll: Sending row: Luca:40.931286:14.097931:
[Wed Jun 1 23:50:01 2022] SQL_GetAll: Sending row: Mogavero:41.056646:14.274286:
[Wed Jun 1 23:50:01 2022] SQL_GetAll: Sending row: Benerecetti:40.894741:14.239694:
[Wed Jun 1 23:50:01 2022] Get All Request: Query complete
[Wed Jun 1 23:50:01 2022] Request manager: Service provided
[Wed Jun 1 23:50:01 2022] Main Thread: Accept success
[Wed Jun 1 23:50:01 2022] Main Thread: Client connect from: 127.0.0.1
[Wed Jun 1 23:50:01 2022] Request manager: Thread request started
[Wed Jun 1 23:50:04 2022] Request manager: Request from client: getNear
[Wed Jun 1 23:50:04 2022] Get Near request: Request from client: User
[Wed Jun 1 23:50:04 2022] Get Near request: Data from User at (40.835884,14.248767) with max range 0.000000
[Wed Jun 1 23:50:04 2022] SQL_NearPotholes: Sending row: Valentino:40.835884:14.248767:
[Wed Jun 1 23:50:04 2022] Get Near request: Query complete
[Wed Jun 1 23:50:04 2022] Request manager: Service provided
[Wed Jun 1 23:50:04 2022] Main Thread: Accept success
[Wed Jun 1 23:50:04 2022] Main Thread: Client connect from: 127.0.0.1
[Wed Jun 1 23:50:04 2022] Request manager: Thread request started
[Wed Jun 1 23:50:07 2022] Request manager: Request from client: post
[Wed Jun 1 23:50:07 2022] Post request: Request from client: User
[Wed Jun 1 23:50:07 2022] Post request: Data from User at 41.134679 - 14.172839
[Wed Jun 1 23:50:07 2022] Post request: Query complete
[Wed Jun 1 23:50:07 2022] Request manager: Service provided
```

Annotations in the image:

- Initialize** (red box) points to the first 'Main Thread: Accept success' line.
- Threshold** (green box) points to the 'Request from client: threshold' line.
- GetAll** (yellow box) points to the 'Request from client: getAll' line.
- GetNear** (purple box) points to the 'Request from client: getNear' line.
- Post** (blue box) points to the 'Request from client: post' line.

Figura 3: Server

per il testing abbiamo creato un semplice client in C per simulare il funzionamento del server (nella directory Client).



```
fsociety@University:~/Sistemi Operativi/Potholes/Server/Client$ ./client
socket successfully created..
connected to the server.. socket 3
Testing [threshold]
From Server : 17
socket successfully created..
connected to the server.. socket 3
Testing [getAll]
From Server : Valentino:40.835884:14.248767:
From Server : Lucia:40.835777:14.248769:
From Server : Mario:40.897242:14.256271:
From Server : Mariano:40.922529:14.297636:
From Server : Antonio:40.891019:14.249263:
From Server : Antonio:40.845483:14.237978:
From Server : Luca:40.826523:14.220133:
From Server : Luca:40.931286:14.097931:
From Server : Mogavero:41.056646:14.274286:
From Server : Benerecetti:40.894741:14.239694:
From Server : END
socket successfully created..
connected to the server.. socket 3
Testing [getNear]
BuffRead: START
From Server : Valentino:40.835884:14.248767:
From Server : END
socket successfully created..
connected to the server.. socket 3
Testing [post]
BuffRead: START
BuffRead: Success
```

Annotations in the image:

- threshold** (red box) points to the 'Testing [threshold]' line.
- getAll** (red box) points to the 'Testing [getAll]' line.
- getNear** (red box) points to the 'Testing [getNear]' line.
- post** (red box) points to the 'Testing [post]' line.

Figura 4: Test client

## 5 Dettagli implementativi - Client

Il client è stato sviluppato su Android Studio con linguaggio Java. Abbiamo seguito il design pattern **MVP - Model View Presenter** e reso la logica di base molto semplice:

- MainActivity - ospita i fragment.
- SplashScreenActivity - utilizza un handler thread per mostrare il logo per circa 3 secondi e fa un setup veloce di alcune funzionalità (i.e Toasty).
- LoginFragment - semplice schermata di login (non essendoci logica di autenticazione (se non rispettare semplici pattern per semplificare la comunicazione ASCII).
- HomepageFragment - schermata principale dell'app che raccoglie tutte le funzionalità di base.

### 5.1 Thread Centering

Onde evitare freeze e crash dell'applicativo android l'app riduce le attività del main thread al minimo e ogni funzione (Recupero della location, spotting di buche e altro) girano su thread anonimi e non, i thread principali che utilizziamo sono:

- Thread Spotter - utilizzato per lo spotting.
- Thread Potholes - per le attività di networking con il server.

Infatti le attività di networking (creazione di socket, comunicazione con send e read) non possono essere eseguite sul main thread, ciò ci ha spinto ad usare il metodo activity **"runOnUiThread(Runnable r)"** per le modifiche dell'interfaccia.

```
@Override
public void run() {
    Log.i(LOG, "Thread Start: " + Thread.currentThread().getName());

    if (homePagePresenter != null) {
        if (!CheckService.checkConnection(homePagePresenter.getContext().getActivity()))
            return;

        try { otherThread.join(); } catch (InterruptedException ignored) {}
        if (otherThread.isAlive() && homePagePresenter.getThreadPool().getPotholes().length > 0)
            homePagePresenter.getContext().getActivity().runOnUiThread(() -> homePagePresenter.viewLoadPotholesList(location));
    } else if (loginPresenter != null) {
        if (!CheckService.checkConnection(loginPresenter.getContext().getActivity()))
            return;

        Network network = new Network(loginPresenter.getContext().getActivity());
        network.getThreshold();
        if (Network.THRESHOLD > 1) {
            loginPresenter.getContext().getActivity().runOnUiThread(() -> {
                loginPresenter.getContext().reset();
                loginPresenter.getContext().goHomePage();
            });
        } else
            Handler.handleException(new IOException(), loginPresenter.getContext().getActivity());
    }
}
```

```
@Override
public void run() {
    Log.i(LOG, "Thread Start: " + Thread.currentThread().getName());
    try { otherThread.join(); } catch (InterruptedException ignored) {}
    homePagePresenter.sendData(location);
}
```

## 5.2 OSM - Open Street Map

La libreria API che abbiamo utilizzato per quanto riguarda tutte le azioni che hanno bisogno di una MAPPA. OSM è stata una scelta ponderata con Google Maps come alternativa, la scelta è stata finalizzata dopo aver visto i seguenti dettagli:

- Licenza: Open Source (Google Maps è di Google).
- Costo: Gratis (Google Maps limitante sul numero di richieste).
- Supporto: librerie degli utenti e della community (Google Maps ha solo la propria documentazione).

Fondamentale prendere quanto più precise le location dei fossi per avere una buona consistenza e precisione nel calcolo e rappresentazione tramite mappa con OSM, abbiamo creato questa funzionalità a doc:

```
/**Tasks*/
public void getLocation(Map<String, Double> loc) {
    Log.i(LOG, "Getting GPS Position");
    if (lgpsManager())
        return;

    if (ActivityCompat.checkSelfPermission(mContext.getActivity(),
        Manifest.permission.ACCESS_FINE_LOCATION) != PackageManager.PERMISSION_GRANTED
        &&
        ActivityCompat.checkSelfPermission(mContext.getActivity(),
        Manifest.permission.ACCESS_COARSE_LOCATION) != PackageManager.PERMISSION_GRANTED) {
        CheckService.checkGpsEnabled(mContext.getActivity(), 333);
    } else {
        Log.i(LOG, "Getting Location");
        fusedLocationClient.getLastLocation().addOnSuccessListener(mContext.getActivity(), location -> {
            if (location != null) {
                Log.i(LOG, "Latitudine: " + location.getLatitude());
                Log.i(LOG, "Longitudine: " + location.getLongitude());
                loc.put("Latitude", location.getLatitude());
                loc.put("Longitude", location.getLongitude());
            } else {
                Log.i("GPS", "Nessuna location.");
                new MaterialAlertDialogBuilder(mContext.getActivity(), R.style.MyThemeOverlay_MaterialComponents_MaterialAlertDialog)
                    .setTitle("Errore GPS")
                    .setMessage("Il GPS è stato disattivato, verrai reindirizzato alla homepage.")
                    .setPositiveButton("Riprova", (dialogInterface, i) -> {
                        getLocation(loc);
                    })
                    .setNegativeButton("Chiudi", (dialog, i) -> {
                    })
                    .show();
            }
        });
    }

    //Wait getting coordinates
    try { Thread.sleep(1000); } catch (InterruptedException ignored) {}
}
}
```

Figura 5: getLocation

### 5.3 Diagramma di classi

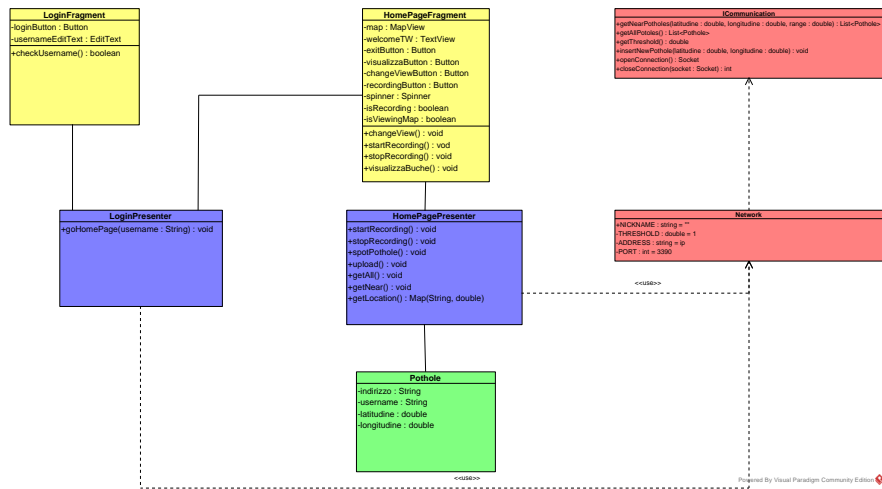


Figura 6: Class Diagram

## 5.4 Networking

La classe che si occupa delle operazioni di networking tra client e server è la classe **Network** che estende un'interfaccia a doc chiamata **ICommunication**, tale classe semplicemente compie quelle che sono le operazioni principali di networking dalla creazione delle socket, alla comunicazione con buffer fino alla loro chiusura.

Particolare attenzione è richiesta alle routine che vengono fatte per ogni operazione (descritte in parte nella sezione "**Protocollo**"). Infatti è possibile notare le fasi descritte di creazione delle socket (tramite l'operazione `new Socket(ADDRESS, PORT)`). successivamente la creazione di un **READER** e **WRITER** per comunicare con la socket, si invia l'operazione da compiere (identificata con la stringa "operation") e successivamente si fanno i vari controlli per scorrere l'eventuale stream di dati. Infine c'è la chiusura della socket con la classica operazione di `close()`.

```

//Methods GET
@Override
public List<Pothole> getNearPotholes(double latitude, double longitude, double range) {
    Log.i(LOG, "Get Near Potholes by "
        + NICKNAME + " at lat "
        + latitude + " and long "
        + longitude + " with range "
        + range);

    String operation = "getNear";
    ArrayList<Pothole> potholes = new ArrayList<>();

    try {
        //Connessione aperta
        Log.i(LOG, "Open connection...");
        Socket socket = openConnection();
        socket.setSoTimeout(5*1000);

        //Richiede operazione getNear
        Log.i(LOG, "Request output stream...");
        socket.getOutputStream().write(operation.getBytes());

        //Apro un buffer reader
        Log.i(LOG, "Request input stream...");
        BufferedReader reader =
            new BufferedReader(
                new InputStreamReader(socket.getInputStream()));

        //Invio informazioni al server
        Log.i(LOG, "Sending information...");
        String tracking = NICKNAME + ":" + latitude + ":" + longitude + ":" + range;
        socket.getOutputStream().write(tracking.getBytes());

        //Attesa informazioni
        Log.i(LOG, "Waiting information...");
        Thread.sleep(1000);

        //Recupero informazioni
        Log.i(LOG, "Reading information...");
        String responseBuffer;
        while(reader.ready()) {
            responseBuffer = reader.readLine();
            responseBuffer = responseBuffer.replace("\u0000", "");
            Log.i(LOG, "Reading: " + responseBuffer);
            if(responseBuffer.equals("START"))
                continue;
            if(responseBuffer.isEmpty() || responseBuffer.equals("END"))
                break;
            String[] fields = responseBuffer.split(":");
            String nickname = fields[0];
            double latitude = Double.parseDouble(fields[1]);
            double longitude = Double.parseDouble(fields[2]);

            potholes.add(new Pothole(nickname, latitude, longitude));
        }

        //Connessione chiusa
        Log.i(LOG, "Open connection...");
        closeConnection(socket);
    } catch (Exception e) {
        Log.e(LOG, "Errore in getNearPotholes");
        Handler.handleException(e, guiRef);
        return null;
    }
    return potholes;
}

```

Figura 7: Network - getNear



```

//Methods GET
@Override
public List<Pothole> getNearPotholes(double latitudine, double longitudine, double range) {
    Log.i(LOG, "Get Near Potholes by "
        + NICKNAME + " at lat "
        + latitudine + " and long "
        + longitudine + " with range "
        + range);

    String operation = "getNear";
    ArrayList<Pothole> potholes = new ArrayList<>();

    try {
        //Connessione aperta
        Log.i(LOG, "Open connection...");
        Socket socket = openConnection();
        socket.setSoTimeout(5*1000);

        //Richiede operazione getNear
        Log.i(LOG, "Request output stream...");
        socket.getOutputStream().write(operation.getBytes());

        //Apro un buffer reader
        Log.i(LOG, "Request input stream...");
        BufferedReader reader =
            new BufferedReader(
                new InputStreamReader(socket.getInputStream()));

        //Invio informazioni al server
        Log.i(LOG, "Sending information...");
        String tracking = NICKNAME + ":" + latitudine + ":" + longitudine + ":" + range;
        socket.getOutputStream().write(tracking.getBytes());

        //Attesa informazioni
        Log.i(LOG, "Waiting information...");
        Thread.sleep(1000);

        //Recupero informazioni
        Log.i(LOG, "Reading information...");
        String responseBuffer;
        while(reader.ready()) {
            responseBuffer = reader.readLine();
            responseBuffer = responseBuffer.replace("\u0000", "");
            Log.i(LOG, "Reading: " + responseBuffer);
            if(responseBuffer.equals("START"))
                continue;
            if(responseBuffer.isEmpty() || responseBuffer.equals("END"))
                break;
            String[] fields = responseBuffer.split(":");
            String nickname = fields[0];
            double latitude = Double.parseDouble(fields[1]);
            double longitude = Double.parseDouble(fields[2]);

            potholes.add(new Pothole(nickname, latitude, longitude));
        }

        //Connessione chiusa
        Log.i(LOG, "Open connection...");
        closeConnection(socket);
    } catch (Exception e) {
        Log.e(LOG, "Errore in getNearPotholes");
        Handler.handleException(e, guiRef);
        return null;
    }
    return potholes;
}

```

Figura 8: Network - getAll

```

@Override
public double getThreshold() {
    Log.i(LOG, "Get Threshold by " + NICKNAME);

    String operation = "threshold";
    String result = "1";

    try {
        //Connessione aperta
        Log.i(LOG, "Open connection...");
        Socket socket = openConnection();
        socket.setSoTimeout(5*1000);

        //Apri un buffer reader
        Log.i(LOG, "Request input stream...");
        BufferedReader reader =
            new BufferedReader(
                new InputStreamReader(socket.getInputStream()));

        //Richiede operazione threshold
        Log.i(LOG, "Request output stream...");
        socket.getOutputStream().write(operation.getBytes());

        //Attesa informazioni
        Log.i(LOG, "Waiting information...");
        Thread.sleep(1000);

        //Recupero informazioni
        Log.i(LOG, "Reading information...");
        if(reader.ready()) {
            result = reader.readLine();
            result = result.replace("\u0000", "");
            result = result.replace(":", "");
            Log.i(LOG, "Reading: " + result);
        }

        //Connessione chiusa
        Log.i(LOG, "Closing connection...");
        closeConnection(socket);

        THRESHOLD = Double.parseDouble(result);
    } catch (Exception e) {
        Log.e(LOG, "Error in getThreshold");
        Handler.handleException(e, guiRef);
    }

    return THRESHOLD;
}

```

Figura 9: Network - threshold

```

//Methods POST
@Override
public void insertNewPothole(double latitudine, double longitudine) {
    Log.i(LOG, "Post new Pothole by "
        + NICKNAME + " at lat "
        + latitudine + " and long "
        + longitudine);

    String operation = "post";

    try {

        //Connessione aperta
        Log.i(LOG, "Opening connection...");
        Socket socket = openConnection();
        socket.setSoTimeout(5*1000);

        //Richiede operazione threshold
        Log.i(LOG, "Request output stream...");
        socket.getOutputStream().write(operation.getBytes());

        //Attesa informazioni
        Log.i(LOG, "Waiting information...");
        Thread.sleep(1000);

        //Invio nuovi dati
        Log.i(LOG, "Sending data...");
        String newData = NICKNAME + ":" + latitudine + ":" + longitudine;
        socket.getOutputStream().write(newData.getBytes());

        //Connessione chiusa
        Log.i(LOG, "Closing connection...");
        closeConnection(socket);

    } catch (Exception e) {
        Log.e(LOG, "Error in insertNewPothole");
        Handler.handleException(e, guiRef);
    }
}

```

Figura 10: Network - post

## 6 Mockup

Il mockup del client è messo a disposizione con il link diretto alla pagina Axure<sup>1</sup> e mostrato qui quanto segue:

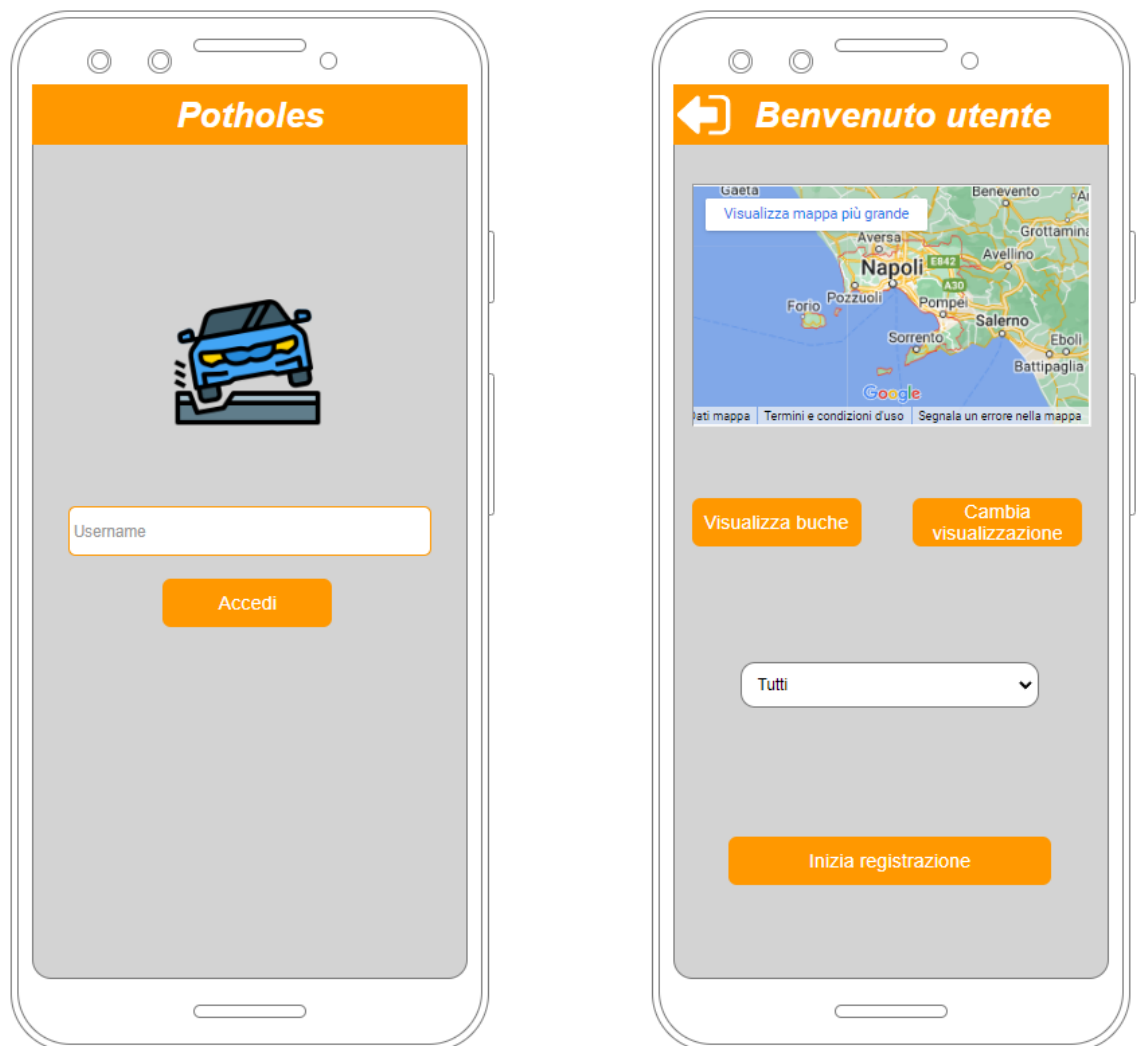


Figura 11: Mockup

---

<sup>1</sup><https://gqqdqx.axshare.com/>

## 7 Artefatti

Ultima sezione contenente gli artefatti per dimostrare l'intero sistema, in particolare state chart, activity diagram e sequence diagram delle funzionalità richieste.

### 7.1 State Chart

Lo state chart dimostra lo stato evolutivo del sistema in tutte le fasi.

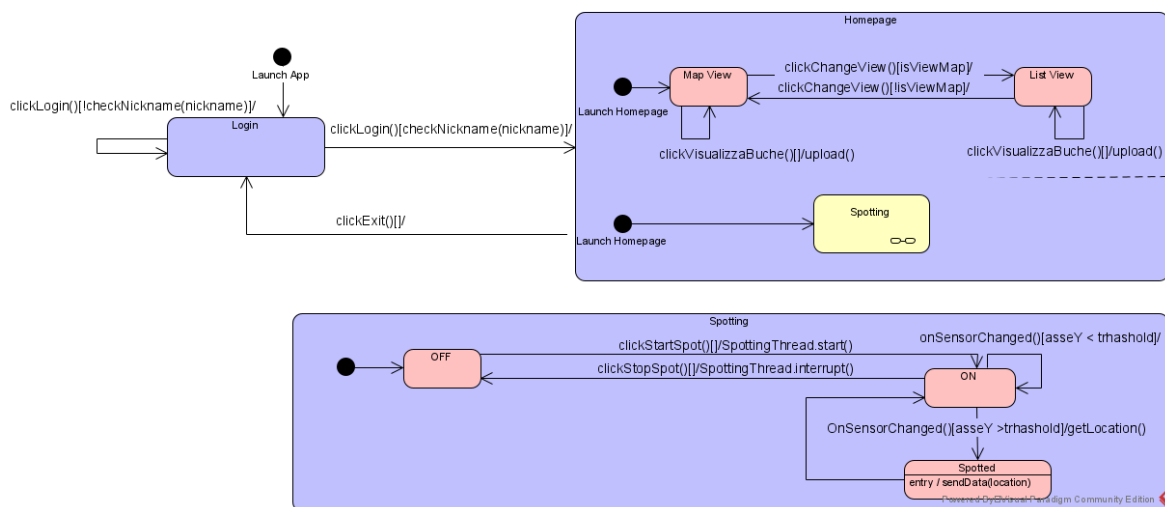
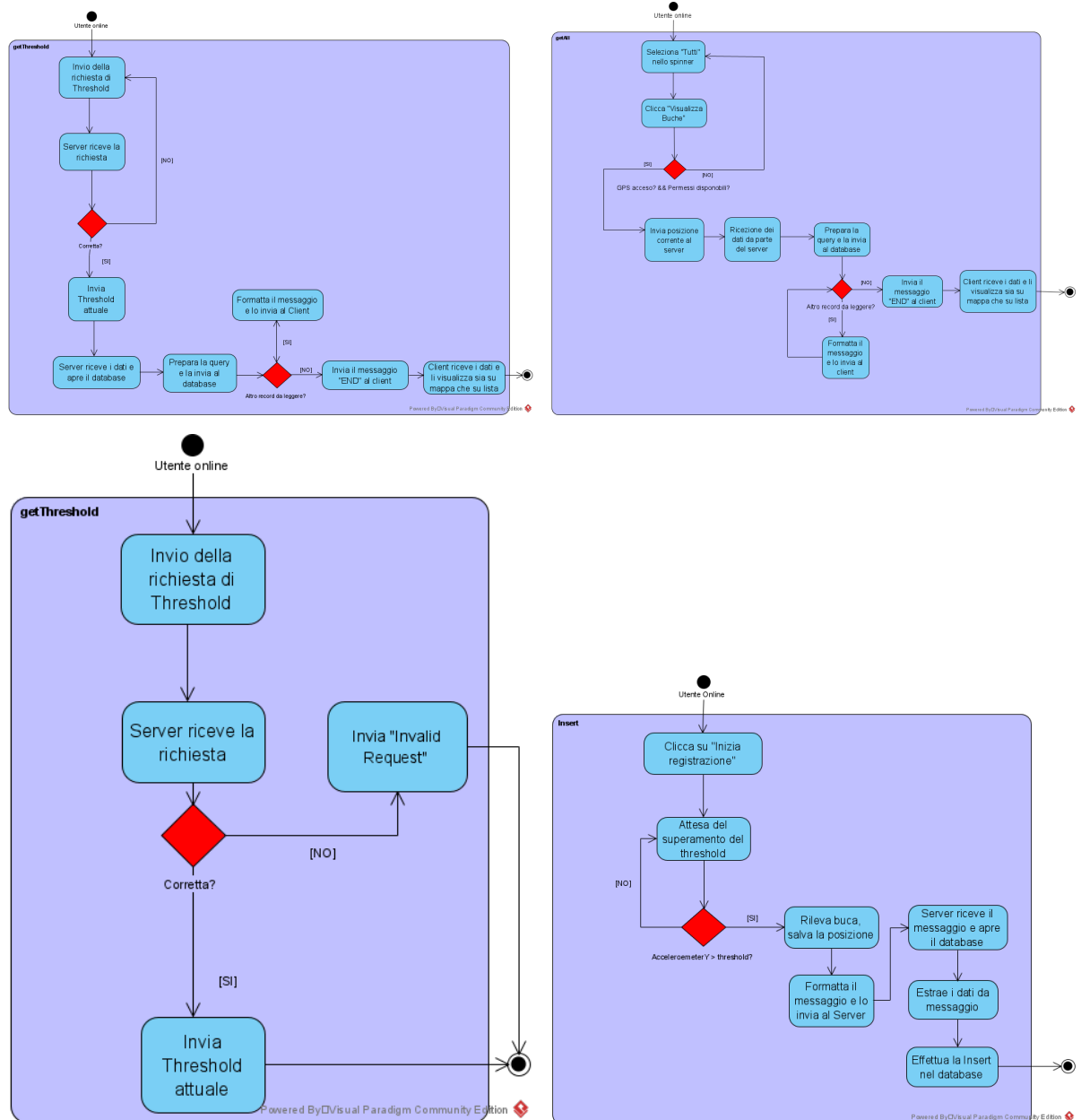


Figura 12: State Chart

## 7.2 Activity Diagram



## 7.3 Sequence diagram

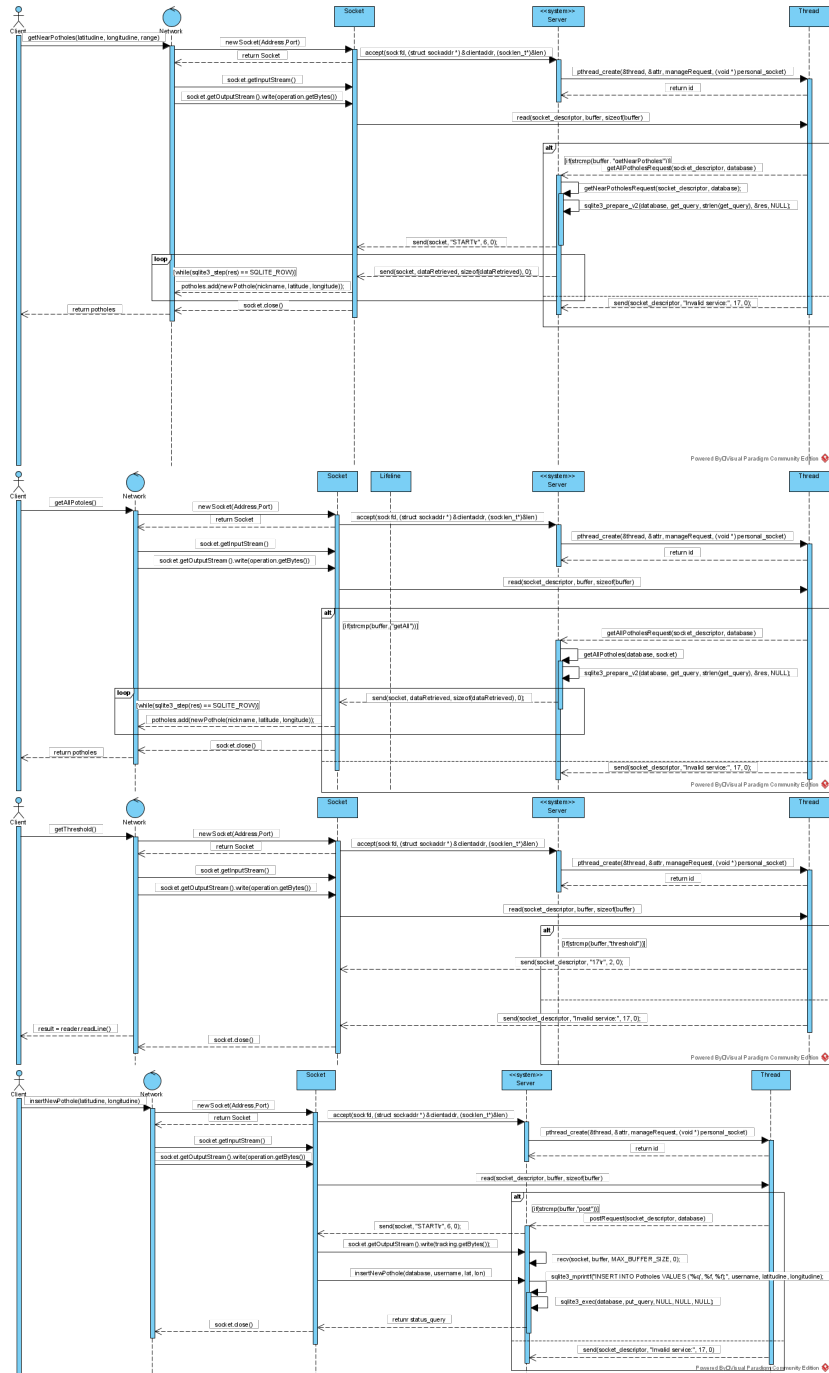


Figura 13: Sequence Diagram

## Riferimenti bibliografici

- [1] C. Negus, *Linux Bible: Boot Up to Ubuntu, Fedora, KNOPPIX, Debian, openSUSE, and 11 Other Distributions*. Bible, Wiley, 2008.
- [2] J. Gustedt, *Modern C*. Manning, 2019.
- [3] M. Murphy, *Beginning Android*. Apress, 2009.