



INSTITUTO POLITÉCNICO NACIONAL

ESCUELA SUPERIOR DE CÓMPUTO

Desarrollo de Sistemas Distribuidos

Protocolo Solicitud-Respuesta Confiable

Alumno:

Bastida Prado Jaime Armando

Ortiz Rodríguez Salvador Alejandro

Rojas Montaña Marcell Douglas

Sánchez Cuevas Esteban

Profesor: Ukranio Coronilla Contreras

Grupo: 4CM4 **Equipo:** Número 7

5 de Abril de 2020

Índice

1. Protocolo Solicitud-Respuesta Confiable.....	2
2. Desarrollo de la Práctica.....	3
2.1 Servidor.....	3
2.2 Cliente.....	3
3. Comprobación de Funcionamiento.....	4
3.1 Pruebas.....	11
4. Conclusiones.....	17

1. Protocolo Solicitud-Respuesta Confiable

De acuerdo con las prácticas anteriores sobre el Protocolo Solicitud-Respuesta y el uso de los *timeouts*; una vez comprendido el uso de los protocolos para la comunicación entre el cliente y el servidor, así como los tiempos de ejecución en estos procesos, procederemos a la optimización de todo ello mediante el Protocolo Solicitud-Respuesta confiable, es decir, mediante la búsqueda de errores y la respectiva solución de estos.

La finalidad de los protocolos es permitir que componentes heterogéneos de sistemas distribuidos puedan desarrollarse independientemente, y por medio de las capas que componen el protocolo, exista una comunicación transparente entre ambos componentes. La confiabilidad en los sistemas distribuidos se refiere a que si una computadora se descompone, el sistema puede sobrevivir como un todo.

En esta práctica haremos uso de nuestro protocolo Solicitud-Respuesta confiable, lo desarrollaremos y observaremos cual es el error que se presenta para darle solución.

2. Desarrollo de la Práctica

El desarrollo de esta práctica consta de dos ejercicios en los cuales se desarrollará una pequeña aplicación cliente-servidor simulando un cajero automático para observar las fallas y errores posibles en el mismo, encontrando los errores y soluciones como lo recomienda Coulouris en el libro “Distributed Systems”.

2.1 Servidor

El servidor mantiene una nano base de datos que solo almacena la cuenta en pesos de un cliente en la variable entera *nbd* y cuyo valor inicial es cero.

2.2 Cliente

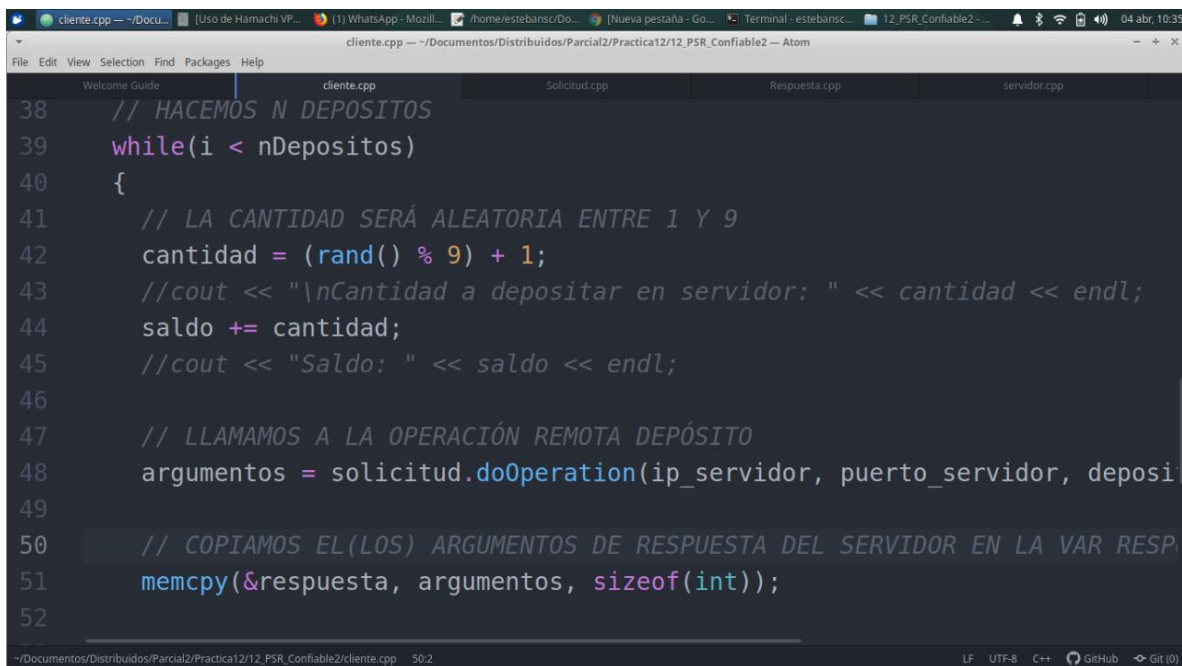
Por otro lado, tendremos un cliente que recibe en la línea de comandos un entero *n*, y va a ejecutar ese número de depósitos de una cantidad aleatoria de pesos comprendida entre \$1 y \$9 sobre su cuenta en el servidor.

Al llegar una solicitud del cliente, el servidor deposita la cantidad recibida en la cuenta, lo cual incrementa *nbd*, y le regresa al cliente el monto actual en su cuenta.

Como el cliente es muy cuidadoso con su dinero debe validar para cada deposito que el valor devuelto por el servidor sea el correcto. En caso de que ocurra un error el programa cliente deberá terminar e imprimir la razón por la que está terminando.

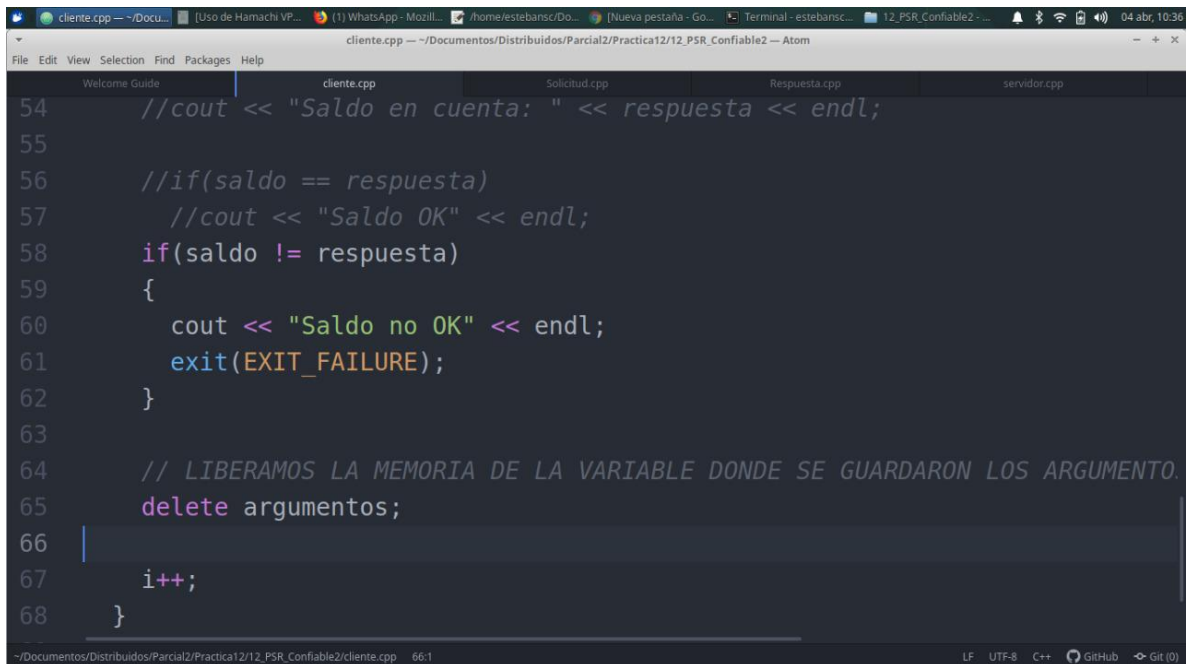
3. Comprobación de Funcionamiento

Para el análisis del código se han comentado las impresiones no necesarias, dejando solo la que nos notifica una incongruencia en el saldo.



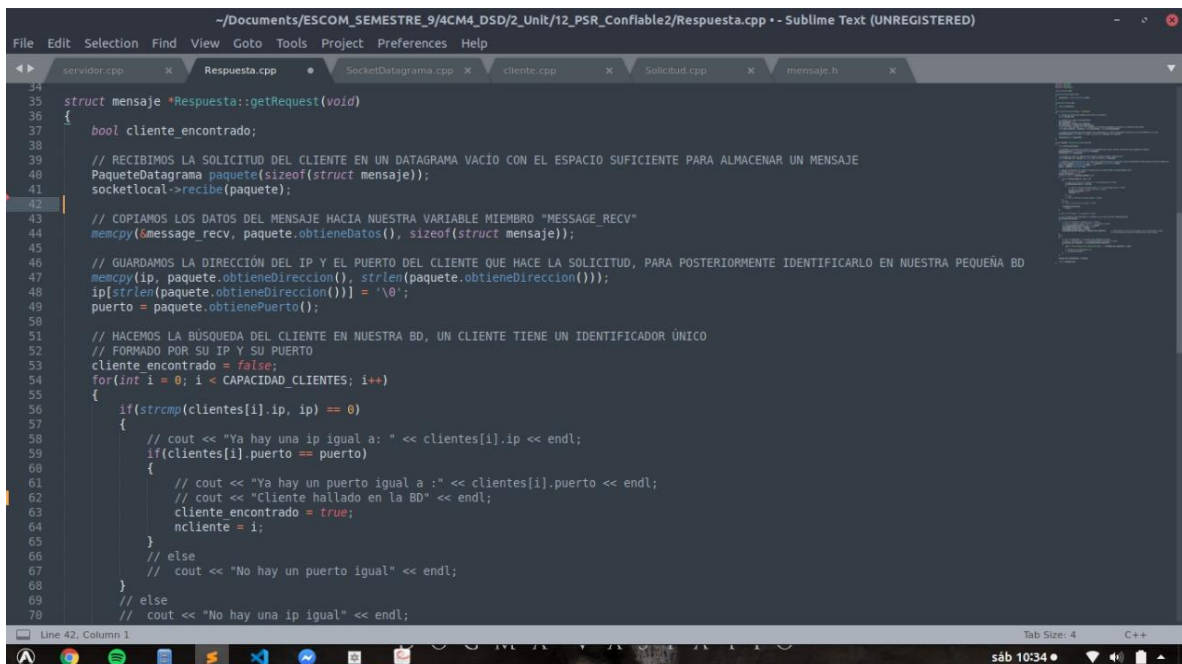
```
38 // HACEMOS N DEPOSITOS
39 while(i < nDepositos)
40 {
41     // LA CANTIDAD SERÁ ALEATORIA ENTRE 1 Y 9
42     cantidad = (rand() % 9) + 1;
43     //cout << "\nCantidad a depositar en servidor: " << cantidad << endl;
44     saldo += cantidad;
45     //cout << "Saldo: " << saldo << endl;
46
47     // LLAMAMOS A LA OPERACIÓN REMOTA DEPÓSITO
48     argumentos = solicitud.doOperation(ip_servidor, puerto_servidor, deposi
49
50     // COPIAMOS EL(LOS) ARGUMENTOS DE RESPUESTA DEL SERVIDOR EN LA VAR RESP
51     memcpy(&respuesta, argumentos, sizeof(int));
52
```

Figura 1



```
54 //cout << "Saldo en cuenta: " << respuesta << endl;
55
56 //if(saldo == respuesta)
57 //cout << "Saldo OK" << endl;
58 if(saldo != respuesta)
59 {
60     cout << "Saldo no OK" << endl;
61     exit(EXIT_FAILURE);
62 }
63
64 // LIBERAMOS LA MEMORIA DE LA VARIABLE DONDE SE GUARDARON LOS ARGUMENTO.
65 delete argumentos;
66
67 i++;
68 }
```

Figura 2



```
34 struct mensaje *Respuesta::getRequest(void)
35 {
36     bool cliente_encontrado;
37
38     // RECIBIMOS LA SOLICITUD DEL CLIENTE EN UN DATAGRAMA VACIO CON EL ESPACIO SUFICIENTE PARA ALMACENAR UN MENSAJE
39     PaqueteDatagrama paquete(sizeof(struct mensaje));
40     socketlocal->recibe(paquete);
41
42     // COPIAMOS LOS DATOS DEL MENSAJE HACIA NUESTRA VARIABLE MIEMBRO "MESSAGE_RECV"
43     memcpy(&message_recv, paquete.obtieneDatos(), sizeof(struct mensaje));
44
45     // GUARDAMOS LA DIRECCIÓN DEL IP Y EL PUERTO DEL CLIENTE QUE HACE LA SOLICITUD, PARA POSTERIORMENTE IDENTIFICARLO EN NUESTRA PEQUEÑA BD
46     memcpy(ip, paquete.obtieneDireccion(), strlen(paquete.obtieneDireccion()));
47     ip[strlen(paquete.obtieneDireccion())] = '\0';
48     puerto = paquete.obtienePuerto();
49
50     // HACEMOS LA BÚSQUEDA DEL CLIENTE EN NUESTRA BD, UN CLIENTE TIENE UN IDENTIFICADOR ÚNICO
51     // FORMADO POR SU IP Y SU PUERTO
52     cliente_encontrado = false;
53     for(int i = 0; i < CAPACIDAD_CLIENTES; i++)
54     {
55         if(strcmp(clientes[i].ip, ip) == 0)
56         {
57             // cout << "Ya hay una ip igual a: " << clientes[i].ip << endl;
58             if(clientes[i].puerto == puerto)
59             {
60                 // cout << "Ya hay un puerto igual a: " << clientes[i].puerto << endl;
61                 // cout << "Cliente hallado en la BD" << endl;
62                 cliente_encontrado = true;
63                 ncliente = i;
64             }
65             // else
66             // cout << "No hay un puerto igual" << endl;
67         }
68         // else
69         // cout << "No hay una ip igual" << endl;
70     }
```

Figura 3

```

103         }
104         // else
105         // cout << "No hay un puerto igual" << endl;
106     }
107     // else
108     // cout << "No hay una ip igual" << endl;
109
110     if(cliente_encontrado)
111         break;
112 }
113
114 // cout << "N cliente: " << ncliente << endl;
115
116 // SI EL CLIENTE NO FUE ENCONTRADO LO ANEXAMOS A LA BD CON SUS DATOS CORRESPONDIENTES
117 if(!cliente_encontrado)
118 {
119     // cout << "cliente no hallado en BD" << endl;
120     // cout << "Agregando nuevo cliente: " << ncliente << endl;
121     memcpy(clientes[ncliente].ip, ip, strlen(ip));
122     clientes[ncliente].ip[strlen(ip)] = '\0';
123     clientes[ncliente].puerto = puerto;
124     clientes[ncliente].requestId = message_rcv.requestId; // Almacenamos el ID de la solicitud, esto es importante, porque
125                                                         // así evitaremos solicitudes repetidas del mismo cliente.
126 }
127 else
128 {
129     // cout << "RequestID: " << message_rcv.requestId << endl;
130     // cout << "RequestID BD: " << clientes[ncliente].requestId << endl;
131     if(message_rcv.requestId == clientes[ncliente].requestId)
132     {
133         cout << "This request has already been done: " << message_rcv.requestId << endl;
134         // message_rcv.operationId = 0;
135         // exit(EXIT_FAILURE);
136     }
137 }
138 }
139
140
141
142
143
144
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
1000

```

Figura 4

```

16 PaqueteDatagrama paquete_rcv(sizeof(struct mensaje));
17
18 // VAR PARA ALMACENAR EL MENSAJE RECIBIDO
19 struct mensaje *mensaje_rcv;
20 // PREPARAMOS UNA VARIABLE PARA ALMACENAR EL ARGUMENTO PASADO (CANTIDAD DE DEPÓSITO)
21 int cantidad_deposito;
22 // VAR PARA ALMACENAR EL RESULTADO DE HACER LA OPERACIÓN, EN ESTE CASO EL SALDO EN LA CUENTA DEL CLIENTE
23 int resultado_nbd;
24
25 while(1)
26 {
27     // GUARDAMOS LA REFERENCIA AL MENSAJE RECIBIDO
28     mensaje_rcv = respuesta.getRequest();
29
30     // IMPRIMIMOS LA INFORMACIÓN RECIBIDA
31     // cout << "\nSOLICITUD RECIBIDA" << endl;
32     // cout << "Tipo de mensaje: " << mensaje_rcv->messageType << endl;
33     // cout << "ID de solicitud: " << mensaje_rcv->requestId << endl;
34     // cout << "ID de operación: " << mensaje_rcv->operationId << endl;
35
36     // COPIAMOS EL(LOS) ARGUMENTOS(CANTIDAD A DEPOSITAR) EN LA VAR CANTIDAD_DEPOSITO
37     memcpy(&cantidad_deposito, mensaje_rcv->arguments, sizeof(int));
38
39     // EJECUTAMOS LA OPERACIÓN SOLICITADA
40     resultado_nbd = 0;
41     if(mensaje_rcv->operationId == deposito)
42         resultado_nbd = fdeposito(cantidad_deposito, mensaje_rcv->messageType);
43     else if(mensaje_rcv->operationId == 0)
44         cout << "AAAAAAAAAAAAAAAAAAAA" << endl;
45
46     // ENVIAMOS LA RESPUESTA
47     respuesta.sendReply((char *) &resultado_nbd);
48 }
49
50 return 0;
51 }
52

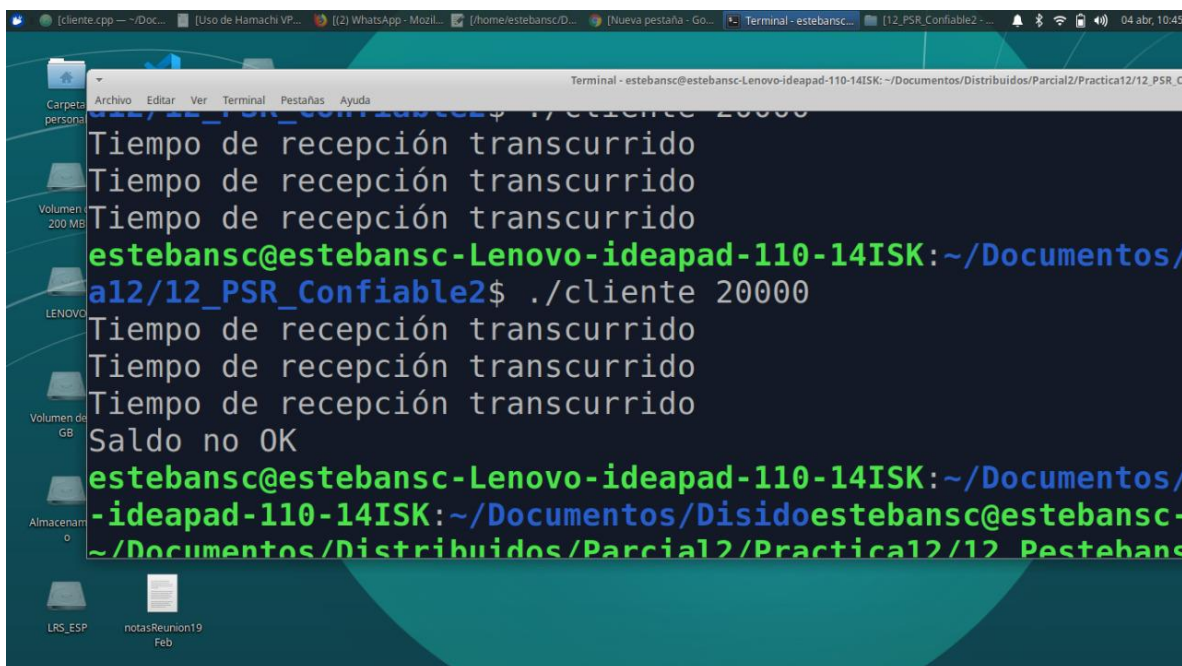
```

Figura 5

En el lado del servidor también se hizo así, solo indicando cuando una petición ya había sido hecha, fijándonos en el *requestId*.

Las pruebas se realizaron en dos diferentes computadoras, conectadas a través de hamachi.

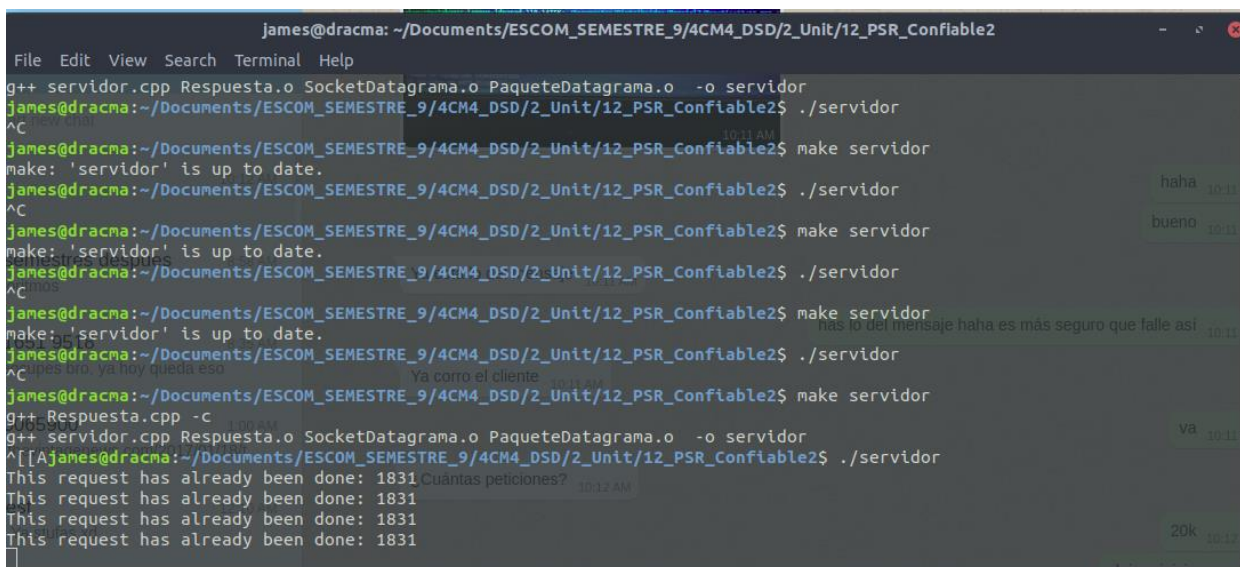
Del lado del cliente se corrió haciendo 20,000 depósitos hasta que ocurrió un fallo, como podemos ver:



```
estebansc@estebansc-Lenovo-ideapad-110-14ISK: ~/Documentos/Distribuidos/Parcial2/Practica12/12_PSR_Confiable2$ ./cliente 20000
Tiempo de recepción transcurrido
Tiempo de recepción transcurrido
Tiempo de recepción transcurrido
estebansc@estebansc-Lenovo-ideapad-110-14ISK: ~/Documentos/Distribuidos/Parcial2/Practica12/12_PSR_Confiable2$ ./cliente 20000
Tiempo de recepción transcurrido
Tiempo de recepción transcurrido
Tiempo de recepción transcurrido
Saldo no OK
estebansc@estebansc-Lenovo-ideapad-110-14ISK: ~/Documentos/Distribuidos/Parcial2/Practica12/12_PSR_Confiable2$
```

Figura 6

Mientras que en el servidor nos limitamos a imprimir la petición repetida cuando ocurre el fallo



```
james@dracma: ~/Documents/ESCOM_SEMESTRE_9/4CM4_DSD/2_Unit/12_PSR_Confiable2
g++ servidor.cpp Respuesta.o SocketDatagrama.o PaqueteDatagrama.o -o servidor
james@dracma: ~/Documents/ESCOM_SEMESTRE_9/4CM4_DSD/2_Unit/12_PSR_Confiable2$ ./servidor
^C
james@dracma: ~/Documents/ESCOM_SEMESTRE_9/4CM4_DSD/2_Unit/12_PSR_Confiable2$ make servidor
make: 'servidor' is up to date.
james@dracma: ~/Documents/ESCOM_SEMESTRE_9/4CM4_DSD/2_Unit/12_PSR_Confiable2$ ./servidor
^C
james@dracma: ~/Documents/ESCOM_SEMESTRE_9/4CM4_DSD/2_Unit/12_PSR_Confiable2$ make servidor
make: 'servidor' is up to date.
james@dracma: ~/Documents/ESCOM_SEMESTRE_9/4CM4_DSD/2_Unit/12_PSR_Confiable2$ ./servidor
^C
james@dracma: ~/Documents/ESCOM_SEMESTRE_9/4CM4_DSD/2_Unit/12_PSR_Confiable2$ make servidor
make: 'servidor' is up to date.
james@dracma: ~/Documents/ESCOM_SEMESTRE_9/4CM4_DSD/2_Unit/12_PSR_Confiable2$ ./servidor
^C
james@dracma: ~/Documents/ESCOM_SEMESTRE_9/4CM4_DSD/2_Unit/12_PSR_Confiable2$ make servidor
g++ servidor.cpp Respuesta.o SocketDatagrama.o PaqueteDatagrama.o -o servidor
james@dracma: ~/Documents/ESCOM_SEMESTRE_9/4CM4_DSD/2_Unit/12_PSR_Confiable2$ ./servidor
This request has already been done: 1831
This request has already been done: 1831
This request has already been done: 1831
This request has already been done: 1831
Cuántas peticiones?
```

Figura 7

Lo que está ocurriendo es que la respuesta a las peticiones a veces no llega al cliente en tiempo, esto se puede dar lugar a dos escenarios diferentes.

Nos centraremos en la solución al escenario de "Solicitudes duplicadas" descrito por Colouris en la página 189 del texto "Distributed Systems"

1. El *timeout* del cliente ha expirado y el servidor ha recibido la solicitud, pero aún sigue procesando la operación.
En este caso el cliente hace un reenvío de la solicitud al servidor, el cual debe reconocer que se trata de la misma solicitud hecha por el mismo cliente, en cuyo caso debe omitir la solicitud pues ya la está procesando y enviará la respuesta cuando haya terminado.
2. El *timeout* del cliente ha expirado y el servidor ha recibido la solicitud y ha respondido pero la respuesta no ha llegado al cliente.
En este caso el servidor debe ser capaz de reconocer que la petición ya ha sido hecha y además que la operación ya ha sido efectuada, en cuyo caso, debe simplemente reenviar la respuesta sin necesidad de hacer de nuevo la operación.

Para resolver el problema implementamos un *struct* cliente, para que el servidor sea capaz de almacenar todos los clientes diferentes que le hayan hecho una petición, usando como identificador dentro de la red a su IP y puerto desde el cual ha enviado la solicitud, y como identificador de solicitud un campo *requestId*:

```
8 // EL SERVIDOR MANTENDRÁ UNA BD DE HASTA 10 CLIENTES
9 #define CAPACIDAD_CLIENTES 10
10
11 // CREAMOS UN STRUCT CLIENTE PARA ALMACENAR LA INFO DE CADA UNO
12 struct cliente
13 {
14     char ip[16];
15     int puerto;
16     unsigned int requestId;
17 };
18
```

Figura 8

Y agregamos un atributo a la clase Respuesta, con capacidad para 10 clientes y un atributo *ncliente*, para identificar a cada uno.

```
19 class Respuesta
20 {
21     public:
22         Respuesta(int pl);
23         ~Respuesta();
24         struct mensaje *getRequest(void);
25         void sendReply(char *respuesta);
26
27     private:
28         SocketDatagrama *socketlocal;
29         struct mensaje message_rcv;
30         char ip[16];
31         int puerto;
32         struct cliente clientes[CAPACIDAD_CLIENTES];
33         int ncliente = 0;
34 };
35
36 #endif
```

Figura 9

El mecanismo de funcionamiento cuando llega una solicitud al servidor es el siguiente.

Se recibe un datagrama, se extrae el mensaje, y se guarda la *ip* y puerto del que envía la solicitud, como se ha estado haciendo normalmente.

```
34 struct mensaje *Respuesta::getRequest(void)
35 {
36     bool cliente_encontrado;
37
38     // RECIBIMOS LA SOLICITUD DEL CLIENTE EN UN DATAGRAMA VACÍO CON EL ESPACIO SUFICIENTE PARA ALMACENAR UN MENSAJE
39     PaqueteDatagrama paquete(sizeof(struct mensaje));
40     socketlocal->recibe(paquete);
41
42     // COPIAMOS LOS DATOS DEL MENSAJE HACIA NUESTRA VARIABLE MIEMBRO "MESSAGE_RECV"
43     memcpy(&message_rcv, paquete.obtieneDatos(), sizeof(struct mensaje));
44
45     // GUARDAMOS LA DIRECCIÓN DEL IP Y EL PUERTO DEL CLIENTE QUE HACE LA SOLICITUD, PARA POSTERIORMENTE IDENTIFICARLO EN NUESTRA PEQUEÑA BD
46     memcpy(ip, paquete.obtieneDireccion(), strlen(paquete.obtieneDireccion()));
47     ip[strlen(paquete.obtieneDireccion())] = '\0';
48     puerto = paquete.obtienePuerto();
49 }
```

Figura 10

Ahora, procedemos a buscar a ese cliente en la pequeña BD, lo identificamos por su *ip* y puerto.

```

51 // HACEMOS LA BÚSQUEDA DEL CLIENTE EN NUESTRA BD, UN CLIENTE TIENE UN IDENTIFICADOR ÚNICO
52 // FORMADO POR SU IP Y SU PUERTO
53 cliente_encontrado = false;
54 for(int i = 0; i < CAPACIDAD_CLIENTES; i++)
55 {
56     if(strcmp(clientes[i].ip, ip) == 0)
57     {
58         if(clientes[i].puerto == puerto)
59         {
60             // cout << "Cliente hallado en la BD" << endl;
61             cliente_encontrado = true;
62             ncliente = i;
63         }
64     }
65
66     if(cliente_encontrado)
67         break;
68 }
69

```

Figura 11

Si el cliente no fue encontrado, lo agregamos a la BD, y guardamos el ID de la solicitud. Si el cliente ya existía en la BD, necesitamos comprobar que el ID de la solicitud recién recibida no es igual a la última almacenada en la BD, si es igual quiere decir que es un reenvío de parte del cliente, en cuyo caso ponemos el ID de operación "*operationId*" a 0,, indicándole al servidor que no debe volver a realizar la operación, es decir, descartar la solicitud repetida.

```

70 // SI EL CLIENTE NO FUE ENCONTRADO LO ANEXAMOS A LA BD CON SUS DATOS CORRESPONDIENTES
71 if(!cliente_encontrado)
72 {
73     // cout << "Cliente no hallado en BD" << endl;
74     // cout << "Agregando nuevo cliente: " << ncliente << endl;
75     memcpy(clientes[ncliente].ip, ip, strlen(ip));
76     clientes[ncliente].ip[strlen(ip)] = '\0';
77     clientes[ncliente].puerto = puerto;
78     clientes[ncliente].requestId = message_rcv.requestId; // Almacenamos el ID de
79                                                         // así evitaremos solici
80 }
81 else
82 {
83     // COMPROBAMOS SI EL ID DE SOLICITUD DEL CLIENTE RECIBIDO ES IGUAL A LA QUE YA TENI
84     // EN CUYO CASO MODIFICAMOS EL ID DE OPERACIÓN A 0, INDICANDO AL SERVIDOR QUE NO R
85     if(message_rcv.requestId == clientes[ncliente].requestId)
86     {
87         cout << "Esta solicitud ya ha sido hecha: " << message_rcv.requestId << endl;
88         message_rcv.operationId = 0;
89     }
90     else // SI SE TRATA DE UNA NUEVA ID DE SOLICITUD, LA ALMACENAMOS EN DICHO CLIENTE
91         clientes[ncliente].requestId = message_rcv.requestId;
92 }
93
94 // ESTA FUE LA MAYOR INCERTIDUMBRE DE LA PRÁCTICA, PARA INDICARLE AL SERVIDOR A QUE CL
95 // PUES ÉSTE MANTIENE UNA PEQUEÑA BD CON LOS SALDOS DE SUS CLIENTES,
96 // SE NECESITA DE ALGÚN MEDIO, ASÍ QUE DECIDIMOS INDICAR EL NUMERO DE CLIENTE EN EL CA
97 // TAMBIÉN SE PUDO HABER AGREGADO UN CAMPO MÁS EN EL STRUCT MENSAJE.
98 message_rcv.messageType = ncliente;

```

Figura 12

En caso de ser diferente, simplemente se guarda esta última ID de solicitud "requestId".

Nos encontramos pues al reto de ¿Cómo comunicarle al servidor a qué cliente debe hacer el depósito, o de que cliente se trata la solicitud? Para resolverlo decidimos almacenar el número de cliente en la variable miembro de la estructura mensaje: "*messageType*"

En este caso necesitamos que nos indique cual sería la mejor manera de hacerlo profesor, y si se puede, por ejemplo, crear un nuevo campo en el struct mensaje para allí almacenar el número de cliente.

3.1 Pruebas

Corriendo el servidor y cliente en dos computadoras diferentes usando hamachi, procedemos a hacer las pruebas que se muestran a continuación:

1. Funcionamiento normal sin fallos

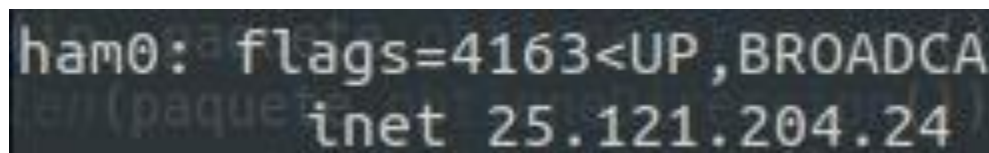
Creamos una red hamachi llamada *distrosys*, en la cual hacemos dos *peers* conectados.



```
james@dracma: ~/Documents/ESCOM_SEMESTRE_9/4CM4_DSD/2_Unit/12_PSR_Confiable2
File Edit View Search Terminal Help
james@dracma:~/Documents/ESCOM_SEMESTRE_9/4CM4_DSD/2_Unit/12_PSR_Confiable2$ sudo hamachi list
[sudo] password for james:
* [distrosys] capacity: 2/5, subscription type: Free, owner: This computer
      243-108-238 estebansc 25.47.119.146 alias: not set 2620:9b::192f:7792
```

Figura 13

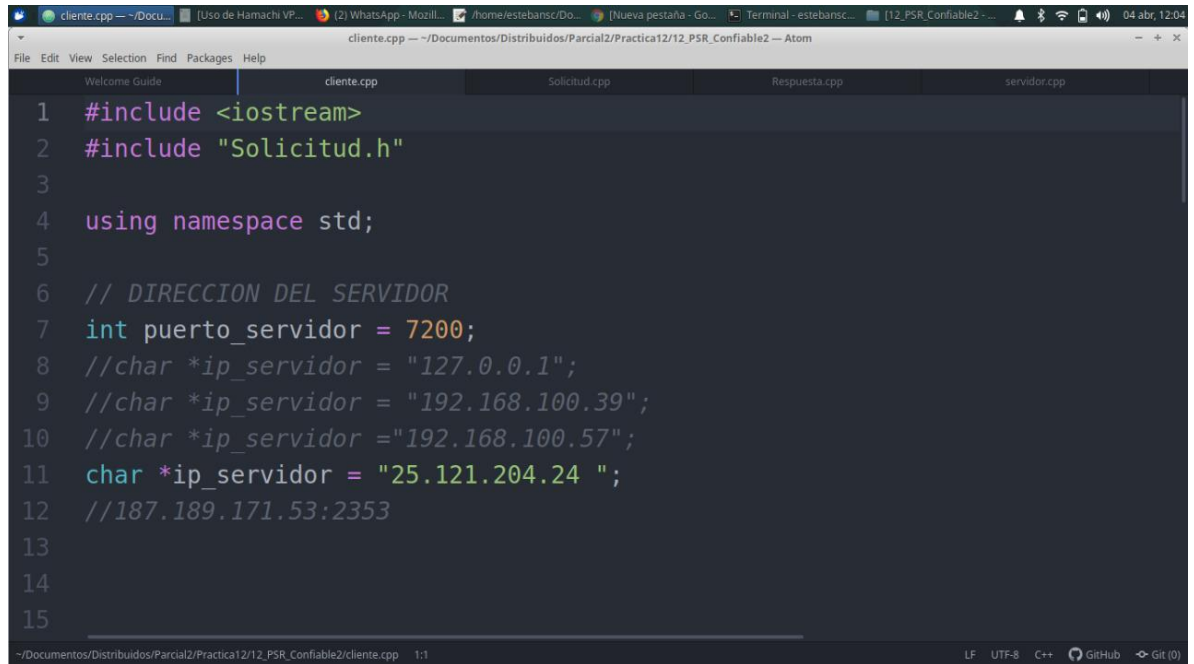
La IP VPN del servidor es la siguiente:



```
ham0: flags=4163<UP, BROADCAST
en (paquete) inet 25.121.204.24
```

Figura 14

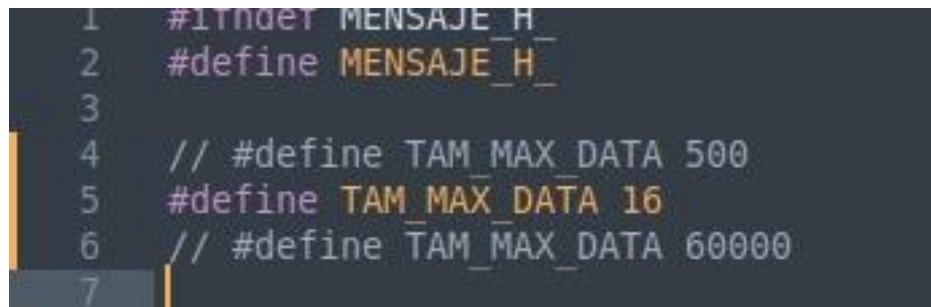
Y en el cliente colocamos esa IP:



```
1 #include <iostream>
2 #include "Solicitud.h"
3
4 using namespace std;
5
6 // DIRECCION DEL SERVIDOR
7 int puerto_servidor = 7200;
8 //char *ip_servidor = "127.0.0.1";
9 //char *ip_servidor = "192.168.100.39";
10 //char *ip_servidor = "192.168.100.57";
11 char *ip_servidor = "25.121.204.24 ";
12 //187.189.171.53:2353
13
14
15
```

Figura 15

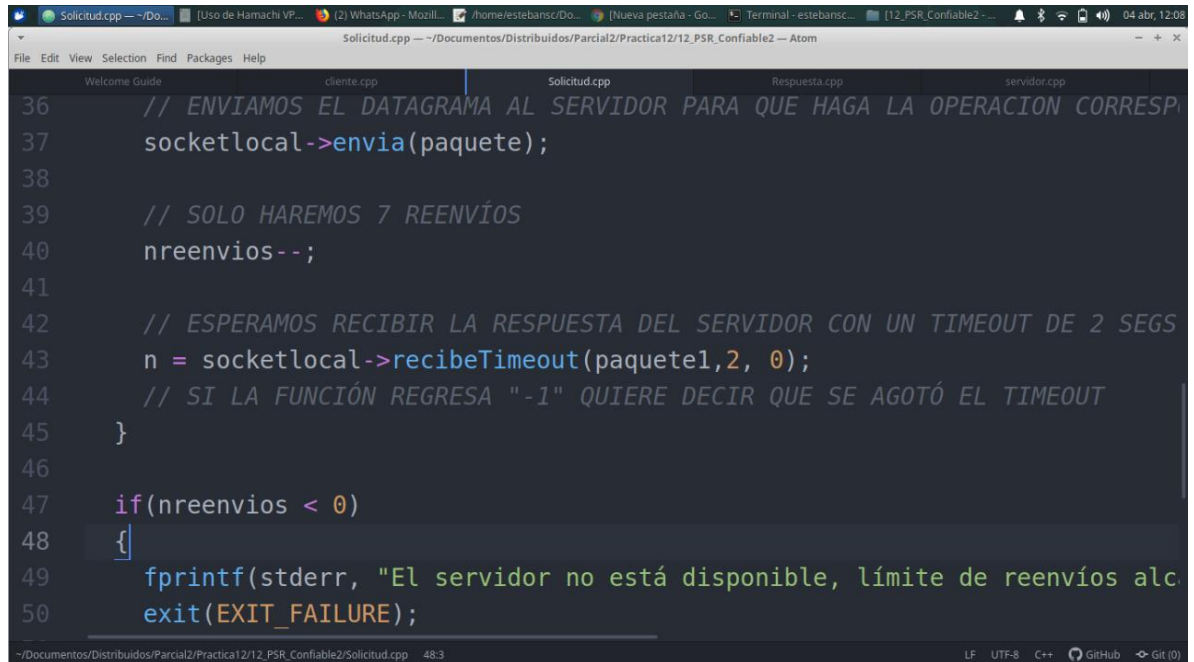
El TAM_MAX_DATA lo tenemos en 16 para optimizar y el timeout del cliente en 2 segundos.



```
1 #ifndef MENSAJE_H
2 #define MENSAJE_H
3
4 // #define TAM_MAX_DATA 500
5 #define TAM_MAX_DATA 16
6 // #define TAM_MAX_DATA 60000
7
```

Figura 16

TimeOut:

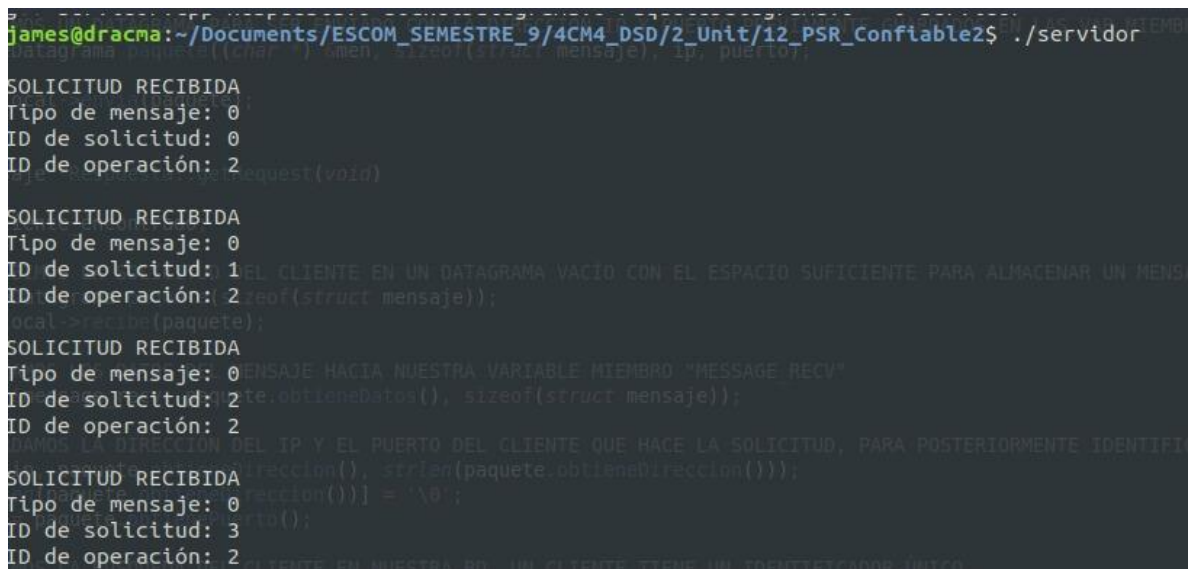


```
36 // ENVIAMOS EL DATAGRAMA AL SERVIDOR PARA QUE HAGA LA OPERACION CORRESP
37 socketlocal->envia(paquete);
38
39 // SOLO HAREMOS 7 REENVÍOS
40 nreenvios--;
41
42 // ESPERAMOS RECIBIR LA RESPUESTA DEL SERVIDOR CON UN TIMEOUT DE 2 SEGS
43 n = socketlocal->recibeTimeout(paquete1, 2, 0);
44 // SI LA FUNCIÓN REGRESA "-1" QUIERE DECIR QUE SE AGOTÓ EL TIMEOUT
45 }
46
47 if(nreenvios < 0)
48 {
49     fprintf(stderr, "El servidor no está disponible, límite de reenvíos alc
50     exit(EXIT_FAILURE);
```

Figura 17

Procedemos a hacer la prueba.

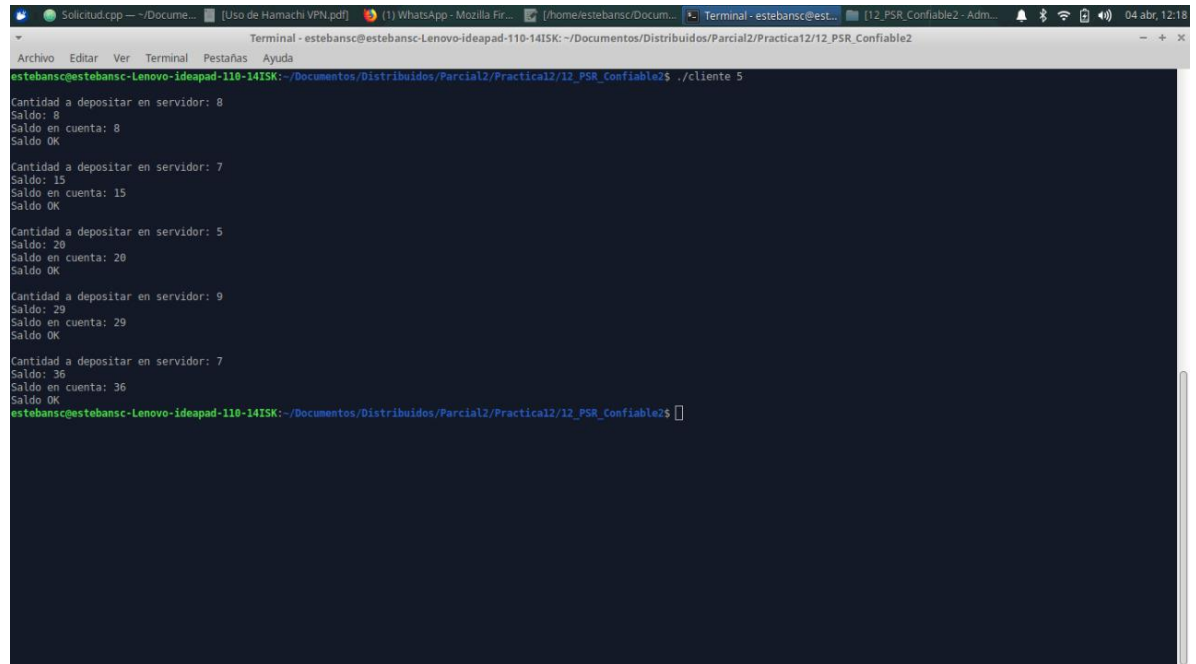
Servidor:



```
james@dracma:~/Documents/ESCOM_SEMESTRE_9/4CM4_DSD/2_Unit/12_PSR_Confiable2$ ./servidor
SOLICITUD RECIBIDA
Tipo de mensaje: 0
ID de solicitud: 0
ID de operación: 2
SOLICITUD RECIBIDA
Tipo de mensaje: 0
ID de solicitud: 1
ID de operación: 2
SOLICITUD RECIBIDA
Tipo de mensaje: 0
ID de solicitud: 2
ID de operación: 2
SOLICITUD RECIBIDA
Tipo de mensaje: 0
ID de solicitud: 3
ID de operación: 2
```

Figura 18

Cliente:



```
estebansc@estebansc-Lenovo-ideapad-110-14ISK:~/Documentos/Distribuidos/Parcial2/Practical2/12_PSR_Confiable2$ ./cliente 5
Cantidad a depositar en servidor: 8
Saldo: 8
Saldo en cuenta: 8
Saldo OK

Cantidad a depositar en servidor: 7
Saldo: 15
Saldo en cuenta: 15
Saldo OK

Cantidad a depositar en servidor: 5
Saldo: 20
Saldo en cuenta: 20
Saldo OK

Cantidad a depositar en servidor: 9
Saldo: 29
Saldo en cuenta: 29
Saldo OK

Cantidad a depositar en servidor: 7
Saldo: 36
Saldo en cuenta: 36
Saldo OK
estebansc@estebansc-Lenovo-ideapad-110-14ISK:~/Documentos/Distribuidos/Parcial2/Practical2/12_PSR_Confiable2$
```

Figura 19

2. Funcionamiento frente a fallos

Funcionamiento con reenvío de mensajes perdidos o que no llegaron a tiempo.

Para lograr esto hicimos que el servidor se durmiera 2 segundos, después de haber recibido un mensaje, haciendo esperar al cliente y forzando a que reenvíe la solicitud como si se tratara de un mensaje perdido.

El servidor está programado para que cuando reciba un código de operación 0 (esto pasa cuando se detecta una solicitud repetida), no realice ninguna operación y prosiga a esperar una nueva petición.


```

24 while(1)
25 {
26     // GUARDAMOS LA REFERENCIA AL MENSAJE RECIBIDO
27     mensaje_recv = respuesta.getRequest();
28
29     // IMPRIMIMOS LA INFORMACIÓN RECIBIDA
30     cout << "\nSOLICITUD RECIBIDA" << endl;
31     cout << "Tipo de mensaje: " << mensaje_recv->messageType << endl;
32     cout << "ID de solicitud: " << mensaje_recv->requestId << endl;
33     cout << "ID de operación: " << mensaje_recv->operationId << endl;
34
35     // COPIAMOS EL(LOS) ARGUMENTOS(CANTIDAD A DEPOSITAR) EN LA VAR CANTIDAD_DEPOSITO
36     memcpy(&cantidad_deposito, mensaje_recv->arguments, sizeof(int));
37
38     // EJECUTAMOS LA OPERACIÓN SOLICITADA
39     resultado_nbd = 0;
40     if(mensaje_recv->operationId == deposito)
41         resultado_nbd = fdeposito(cantidad_deposito, mensaje_recv->messageType);
42     else if(mensaje_recv->operationId == 0)
43     {
44         // SI EL ID DE OPERACIÓN ES CERO, EL SERVIDOR NO EFECTUARÁ NINGUNA OPERACIÓN, SE DESCARTA LA SOLICITUD REPETIDA
45         continue;
46     }
47
48     // ENVIAMOS LA RESPUESTA
49     respuesta.sendReply((char *) &resultado_nbd);
50 }

```

Figura 20

Entonces, corremos el cliente haciendo 2 depósitos:

```

Terminal - estebansc@estebansc-Lenovo-ideapad-110-14ISK: ~/Documentos/Distribuidos/Parcial2/Practica12/12_PSR_Confiable2
estebansc@estebansc-Lenovo-ideapad-110-14ISK:~/Documentos/Distribuidos/Parcial2/Practica12/12_PSR_Confiable2$ ./cliente 2
Cantidad a depositar en servidor: 1
Saldo: 1
Tiempo de recepción transcurrido
Tiempo de recepción transcurrido
Saldo en cuenta: 1
Saldo OK

Cantidad a depositar en servidor: 3
Saldo: 4
Tiempo de recepción transcurrido
Tiempo de recepción transcurrido
Tiempo de recepción transcurrido
Tiempo de recepción transcurrido
Tiempo de recepción transcurrido
Saldo en cuenta: 4
Saldo OK
estebansc@estebansc-Lenovo-ideapad-110-14ISK:~/Documentos/Distribuidos/Parcial2/Practica12/12_PSR_Confiable2$

```

Figura 21

Servidor:

```
james@dracma:~/Documents/ESCOM_SEMESTRE_9/4CM4_DSD/2_Unit/12_PSR_Confiable2$ ./servidor
Mensaje recv = respuesta.getRequest();
SOLICITUD RECIBIDA
Tipo de mensaje: 0
ID de solicitud: 0
ID de operación: 2
Esta solicitud ya ha sido hecha
ID de operación: 2
SOLICITUD RECIBIDA
Tipo de mensaje: 0
ID de solicitud: 0
ID de operación: 0
Esta solicitud ya ha sido hecha
ID de operación: 0
Mensaje recv.operationId = deposito
SOLICITUD RECIBIDA
Tipo de mensaje: 0
ID de solicitud: 0
ID de operación: 0
OPERACIÓN ES CERO, EL SERVIDOR NO EFECTUARÁ NINGUNA OPERACIÓN, SE DESCARTA LA SOLICITUD
continúe:
SOLICITUD RECIBIDA
Tipo de mensaje: 0
ID de solicitud: 1
ID de operación: 2
Esta solicitud va ha sido hecha
```

Figura 22

Podemos observar, que a pesar de que el cliente haya realizado múltiples reenvíos de solicitudes, el servidor es capaz de identificarlas y no volver a hacer las operaciones, de esta manera no se descuadra el saldo del cliente.

4. Conclusiones

En conclusión, creemos que los objetivos marcados en esta práctica se cumplieron satisfactoriamente al poder desarrollar de manera efectiva un protocolo de solicitud-respuesta confiable, fue un proceso un tanto tedioso debido al distanciamiento por la cuarentena y el uso de nuevas herramientas, en este caso el manejo de Hamachi para la comunicación y pruebas a distancia entre los integrantes del equipo. Pero podemos afirmar que luego de muchas pruebas (tests) con nuestro protocolo pudimos comprender mejor el uso de este en los sistemas distribuidos, así como el de poder solucionar los errores presentados de una forma más clara y eficiente.

También gracias al desarrollo de esta práctica comprendimos mejor el funcionamiento de varios procesos, no solo los esperados en la práctica, sino también de externos, sobre todo gracias a las imposibilidades que tuvimos para el desarrollo de la misma, en este caso, el trabajar la práctica a distancia cada quién en su casa.