



# PROGRAMACIÓN EN RED (SOCKETS)

(CC) Moreno, A. M. & Bravo, S. (Redes I, 2016)

## Contenido

2

- Presentación
  - ▣ Objetivos y entorno de desarrollo
  - ▣ Requisitos
- Ejemplos
  - ▣ Servidor TCP y UDP
  - ▣ Cliente TCP
  - ▣ Cliente UDP
- Consideraciones

## Objetivos y entorno de desarrollo

3

- El objetivo de esta práctica es implementar un aplicación en red como
  - ▣ Usuario del nivel de transporte y
  - ▣ Según la arquitectura o modelo de programación cliente-servidor
- Entorno de desarrollo
  - ▣ Estación de trabajo con S.O. Solaris (olivo.fis.usal.es) y Linux
  - ▣ Sockets de Berkeley
  - ▣ Lenguaje de programación C

## Requisitos (I)

4

- Se implementará un mini servicio web
- **Órdenes del cliente al servidor**

```
GET /index.html HTTP/1.1<CR><LF>
Host: www.usal.es<CR><LF>
Connection: keep-alive<CR><LF>
<CR><LF> (Línea en blanco enviada también por el navegador.)
```
- **Respuestas del servidor**
  - ▣ **Correcta**

```
HTTP/1.1 200 OK<CR><LF>
Server: Servidor de Nombre_alumno<CR><LF>
Connection: keep-alive<CR><LF>
<CR><LF>
<html><body><h1>Universidad de
Salamanca</h1></body></html>
```

## Requisitos (II)

5

### □ Errores

#### □ No existe el objeto

- HTTP/1.1 404 Not Found<CR><LF>
- Server: Servidor de Nombre\_alumno<CR><LF>
- Connection: keep-alive<CR><LF>
- <CR><LF>
- <html><body><h1>404 Not found</h1></body></html>

#### □ Orden errónea (algo distinto a GET ...)

- HTTP/1.1 501 Not Implemented<CR><LF>
- Server: Servidor de Nombre\_alumno<CR><LF>
- Connection: close<CR><LF>
- <CR><LF>

## Requisitos (III)

6

### □ Ejemplo de dialogo (lo que ve el cliente):

□ Cliente> GET /index.html k

<html><body><h1>Web de la Universidad de Salamanca</h1></body></html>

□ Cliente> GET /docencia.html k

<html><body><h2>Docencia de la Universidad de Salamanca</h2></body></html>

□ Cliente> GET /ejemplo.html k

<html><body><h1>404 Not Found</h1></body></html>

□ Cliente> DAME /index.htm

<html><body><h1>501 Not Implemented</h1></body></html>

(Se cierra la comunicación)

## Requisitos (III)

7

### Lo que realmente viaja por la red (los mensajes del protocolo)

#### Cliente> GET /index.html k

##### Petición del cliente al servidor:

- GET /index.html  
HTTP/1.1<CR><LF>
- Host: url\_servidor<CR><LF>
- Connection: keep-alive<CR><LF>
- <CR><LF>

##### Respuesta del servidor:

- HTTP/1.1 200 OK<CR><LF>
- Server: Servidor de  
Nombre\_alumno<CR><LF>
- Connection: keep-alive<CR><LF>
- <CR><LF>
- <html><body><h1>Universidad de  
Salamanca</h1></body></html>

#### Cliente> GET /docencia.html k

##### Petición del cliente al servidor:

- GET /docencia.html  
HTTP/1.1<CR><LF>
- Host: url\_servidor<CR><LF>
- Connection: keep-alive<CR><LF>
- <CR><LF>

##### Respuesta del servidor:

- HTTP/1.1 200 OK<CR><LF>
- Server: Servidor de  
Nombre\_alumno<CR><LF>
- Connection: keep-alive<CR><LF>
- <CR><LF>
- <html><body><h2>Docencia de la  
USAL</h2></body></html>

## Requisitos (III)

8

### Lo que realmente viaja por la red (los mensajes del protocolo)

#### Cliente> GET /ejemplo.html k

##### Petición del cliente al servidor:

- GET /ejemplo.html  
HTTP/1.1<CR><LF>
- Host: url\_servidor<CR><LF>
- Connection: keep-alive<CR><LF>
- <CR><LF>

##### Respuesta del servidor:

- HTTP/1.1 404 Not Found<CR><LF>
- Server: Servidor de  
Nombre\_alumno<CR><LF>
- Connection: keep-alive<CR><LF>
- <CR><LF>
- <html><body><h1>404 Not Found</h1></body></html>

#### Cliente> DAME /index.htm

##### Petición del cliente al servidor:

- DAME /index.htm  
HTTP/1.1<CR><LF>
- Host: url\_servidor<CR><LF>
- <CR><LF>

##### Respuesta del servidor:

- HTTP/1.1 501 Not Implemented<CR><LF>
- Server: Servidor de  
Nombre\_alumno<CR><LF>
- Connection: close<CR><LF>
- <CR><LF>
- <html><body><h1>501 Not Implemented</h1></body></html>

#### (Se cierra la comunicación)

## Requisitos (IV)

- **Programa Servidor**
  - Aceptará peticiones de sus clientes tanto en TCP como en UDP
  - Registrará todas las peticiones en un fichero de "log" llamado peticiones.log en el que anotará:
    - Comunicación realizada: nombre del host, dirección IP, protocolo de transporte, nº de puerto efímero del cliente y la fecha y hora a la que se ha producido.
    - Una línea por cada objeto solicitado indicando si se ha atendido correctamente o en caso contrario especificar la causa del error.
  - Se ejecutará como un "daemon".
- **Programa Cliente**
  - Se conectará con el servidor bien con TCP o UDP.
  - Leerá por parámetros el nombre del servidor y el protocolo de transporte TCP o UDP de la siguiente forma:
    - cliente nombre\_o\_IP\_del\_servidor TCP
  - Realizará peticiones al servidor como se ha indicado anteriormente.
  - Realizará las acciones oportunas para su correcta finalización.

## Requisitos (II): pruebas

- Durante la fase de pruebas el cliente podrá ejecutarse como se muestra en el ejemplo de diálogo anterior, pero en la versión para entregar el cliente
  - Leerá de un fichero las órdenes que ha de ejecutar. El nombre del fichero lo recibirá como parámetro
  - Escribirá las respuestas obtenidas del servidor y los mensajes de error y/o depuración en un fichero con nombre el número de puerto efímero del cliente y extensión .txt

## Requisitos (III): versión entregable

11

- Para verificar que esta práctica funciona correctamente y permite operar con varios clientes, se utilizará el *script* `lanzaServidor.sh` que ha de adjuntarse obligatoriamente en el fichero de entrega de esta práctica
- El contenido de `lanzaServidor.sh` es el siguiente:

```
# lanzaServidor.sh
# Lanza el servidor que es un daemon y varios clientes
# las ordenes están en un fichero que se pasa como tercer
  parámetro
servidor
cliente olivo TCP ordenes.txt &
cliente olivo TCP ordenes1.txt &
cliente olivo TCP ordenes2.txt &
cliente olivo UDP ordenes.txt &
cliente olivo UDP ordenes1.txt &
cliente olivo UDP ordenes2.txt &
```

## Requisitos (IV): documentación

12

- Entregar un informe en formato PDF que contenga:
  - ▣ Detalles relevantes del desarrollo de la práctica
  - ▣ Justificación del protocolo de transporte que elegiría para desarrollar esta aplicación

## Servidor TCP y UDP

Ejemplos de servidor en TCP y UDP  
(servidor.c)

### Servidor TCP (I): *servidor.c*

14

- Crear el socket local (ls)

htons (host to network short) y htonl (host to network long) realizan una conversión del dato de tipo entero (corto o largo) para garantizar su envío por la red. Esta conversión es necesaria para **hacer portable el programa entre diferentes arquitecturas de procesador**

- Número de puerto
- Obtener el descriptor del socket TCP
- SOCK\_STREAM

```

69  myaddr_in.sin_port = htons(7070);
70  /* Find the IP address of the "example" server
71  * in order to connect to the desired port number.
72  */
73  myaddr_in.sin_port = htons(7070);
74
75  /* Create the listen socket. */
76  ls = socket (AF_INET, SOCK_STREAM, 0);
77  if (ls == -1) {
78      perror(argv[0]);
79      fprintf(stderr, "is: unable to create socket\n", argv[0]);
80      exit(1);
81  }

```

## Servidor TCP (II): *servidor.c*

15

- Asociar (bind) el socket local de acuerdo con la información previamente indicada en la estructura `sockaddr_in`
- Reservar (listen) una cola para guardar las peticiones pendientes de aceptación

```

83  /* Bind the listen address to the socket. */
84  if (bind(is, (const struct sockaddr *) &myaddr_in, sizeof(struct sockaddr_in)) == -1) {
85      perror(argv[0]);
86      fprintf(stderr, "%s: unable to bind address\n", argv[0]);
87      exit(1);
88  }
89  /* Initiate the listen on the socket so remote users
90   * can connect. The listen backlog is set to 5, which
91   * is the largest currently supported.
92   */
93  if (listen(is, 5) == -1) {
94      perror(argv[0]);
95      fprintf(stderr, "%s: unable to listen on socket\n", argv[0]);
96      exit(1);
97  }

```

## Servidor (III): *servidor.c*

16

- Convertir el servidor en un proceso demonio
  - Se desvincula el proceso del terminal abierto (setpgrp)
  - Se crea el proceso que hará las funciones de servidor (fork)
  - Se ignora la señal SIGCHLD (sigaction) para evitar la acumulación de procesos zombi al morir el proceso padre
  - Se crea un bucle (for) infinito para que el proceso esté siempre ejecutándose (daemon)

```

99  /* Now, all the initialization of the server is
100   * complete, and any user errors will have already
101   * been detected. Now we can fork the daemon and
102   * return to the user. We need to do a setpgrp
103   * so that the daemon will no longer be associated
104   * with the user's control terminal. This is done
105   * before the fork, so that the child will not be
106   * a process group leader. Otherwise, if the child
107   * were to open a terminal, it would become associated
108   * with that terminal as its control terminal. It is
109   * always best for the parent to do the setpgrp.
110   */
111  setpgrp();
112  switch (fork()) {
113      case -1: /* Unable to fork, for some reason. */
114          perror(argv[0]);
115          fprintf(stderr, "%s: unable to fork daemon\n", argv[0]);
116          exit(1);
117      case 0: /* The child process (daemon) comes here. */
118          /* Close stdin and stdout so that they will not
119           * be kept open. Stdout is assumed to have been
120           * redirected to some logging file, or /dev/null.
121           * From now on, the daemon will not report any
122           * error messages. This daemon will loop forever,
123           * waiting for connections and forking a child
124           * server to handle each one.
125           */
126          close(stdin);
127          close(stdout);
128          /* Set SIGCHLD to SIG_IGN, in order to prevent
129           * the accumulation of zombies as each child
130           * terminates. This means the daemon does not
131           * have to make wait calls to clean them up.
132           */
133          sigaction(SIGCHLD, &sa, NULL);
134          for(;;) {

```



## Servidor TCP (IV): *servidor.c*

17

### □ Aceptar peticiones

- La función `accept` devuelve un nuevo socket (`s`) a través del cual se desarrollará el diálogo con el cliente (multiplexación de conexiones en el mismo número de puerto)
- La dirección del cliente se almacena en una nueva estructura `sockaddr_in` y queda asociada al nuevo socket
- Para cada cliente que llega se crea un proceso independiente que lo atiende
  - El servidor queda liberado para aceptar nuevos clientes

```

137 0
138 1
139 2
140 3
141 4
142 5
143 6
144 7
145 8
146 9
147 10
148 11
149 12
150 13
151 14
152 15
153 16
154 17
155 18
156 19
157 20
158 21
159 22
160 23
161 24
162 25
163 26
164 27
165 28
166 29
167 30
168 31
169 32
170 33
171 34
172 35
173 36
174 37
175 38
176 39
177 40
178 41
179 42
180 43
181 44
182 45
183 46
184 47
185 48
186 49
187 50
188 51
189 52
190 53
191 54
192 55
193 56
194 57
195 58
196 59
197 60
198 61
199 62
200 63
201 64
202 65
203 66
204 67
205 68
206 69
207 70
208 71
209 72
210 73
211 74
212 75
213 76
214 77
215 78
216 79
217 80
218 81
219 82
220 83
221 84
222 85
223 86
224 87
225 88
226 89
227 90
228 91
229 92
230 93
231 94
232 95
233 96
234 97
235 98
236 99
237 100
238 101
239 102
240 103
241 104
242 105
243 106
244 107
245 108
246 109
247 110
248 111
249 112
250 113
251 114
252 115
253 116
254 117
255 118
256 119
257 120
258 121
259 122
260 123
261 124
262 125
263 126
264 127
265 128
266 129
267 130
268 131
269 132
270 133
271 134
272 135
273 136
274 137
275 138
276 139
277 140
278 141
279 142
280 143
281 144
282 145
283 146
284 147
285 148
286 149
287 150
288 151
289 152
290 153
291 154
292 155
293 156
294 157
295 158
296 159
297 160
298 161
299 162
300 163
301 164
302 165
303 166
304 167
305 168
306 169
307 170
308 171
309 172
310 173
311 174
312 175
313 176
314 177
315 178
316 179
317 180
318 181
319 182
320 183
321 184
322 185
323 186
324 187
325 188
326 189
327 190
328 191
329 192
330 193
331 194
332 195
333 196
334 197
335 198
336 199
337 200
338 201
339 202
340 203
341 204
342 205
343 206
344 207
345 208
346 209
347 210
348 211
349 212
350 213
351 214
352 215
353 216
354 217
355 218
356 219
357 220
358 221
359 222
360 223
361 224
362 225
363 226
364 227
365 228
366 229
367 230
368 231
369 232
370 233
371 234
372 235
373 236
374 237
375 238
376 239
377 240
378 241
379 242
380 243
381 244
382 245
383 246
384 247
385 248
386 249
387 250
388 251
389 252
390 253
391 254
392 255
393 256
394 257
395 258
396 259
397 260
398 261
399 262
400 263
401 264
402 265
403 266
404 267
405 268
406 269
407 270
408 271
409 272
410 273
411 274
412 275
413 276
414 277
415 278
416 279
417 280
418 281
419 282
420 283
421 284
422 285
423 286
424 287
425 288
426 289
427 290
428 291
429 292
430 293
431 294
432 295
433 296
434 297
435 298
436 299
437 300
438 301
439 302
440 303
441 304
442 305
443 306
444 307
445 308
446 309
447 310
448 311
449 312
450 313
451 314
452 315
453 316
454 317
455 318
456 319
457 320
458 321
459 322
460 323
461 324
462 325
463 326
464 327
465 328
466 329
467 330
468 331
469 332
470 333
471 334
472 335
473 336
474 337
475 338
476 339
477 340
478 341
479 342
480 343
481 344
482 345
483 346
484 347
485 348
486 349
487 350
488 351
489 352
490 353
491 354
492 355
493 356
494 357
495 358
496 359
497 360
498 361
499 362
500 363
501 364
502 365
503 366
504 367
505 368
506 369
507 370
508 371
509 372
510 373
511 374
512 375
513 376
514 377
515 378
516 379
517 380
518 381
519 382
520 383
521 384
522 385
523 386
524 387
525 388
526 389
527 390
528 391
529 392
530 393
531 394
532 395
533 396
534 397
535 398
536 399
537 400
538 401
539 402
540 403
541 404
542 405
543 406
544 407
545 408
546 409
547 410
548 411
549 412
550 413
551 414
552 415
553 416
554 417
555 418
556 419
557 420
558 421
559 422
560 423
561 424
562 425
563 426
564 427
565 428
566 429
567 430
568 431
569 432
570 433
571 434
572 435
573 436
574 437
575 438
576 439
577 440
578 441
579 442
580 443
581 444
582 445
583 446
584 447
585 448
586 449
587 450
588 451
589 452
590 453
591 454
592 455
593 456
594 457
595 458
596 459
597 460
598 461
599 462
600 463
601 464
602 465
603 466
604 467
605 468
606 469
607 470
608 471
609 472
610 473
611 474
612 475
613 476
614 477
615 478
616 479
617 480
618 481
619 482
620 483
621 484
622 485
623 486
624 487
625 488
626 489
627 490
628 491
629 492
630 493
631 494
632 495
633 496
634 497
635 498
636 499
637 500
638 501
639 502
640 503
641 504
642 505
643 506
644 507
645 508
646 509
647 510
648 511
649 512
650 513
651 514
652 515
653 516
654 517
655 518
656 519
657 520
658 521
659 522
660 523
661 524
662 525
663 526
664 527
665 528
666 529
667 530
668 531
669 532
670 533
671 534
672 535
673 536
674 537
675 538
676 539
677 540
678 541
679 542
680 543
681 544
682 545
683 546
684 547
685 548
686 549
687 550
688 551
689 552
690 553
691 554
692 555
693 556
694 557
695 558
696 559
697 560
698 561
699 562
700 563
701 564
702 565
703 566
704 567
705 568
706 569
707 570
708 571
709 572
710 573
711 574
712 575
713 576
714 577
715 578
716 579
717 580
718 581
719 582
720 583
721 584
722 585
723 586
724 587
725 588
726 589
727 590
728 591
729 592
730 593
731 594
732 595
733 596
734 597
735 598
736 599
737 600
738 601
739 602
740 603
741 604
742 605
743 606
744 607
745 608
746 609
747 610
748 611
749 612
750 613
751 614
752 615
753 616
754 617
755 618
756 619
757 620
758 621
759 622
760 623
761 624
762 625
763 626
764 627
765 628
766 629
767 630
768 631
769 632
770 633
771 634
772 635
773 636
774 637
775 638
776 639
777 640
778 641
779 642
780 643
781 644
782 645
783 646
784 647
785 648
786 649
787 650
788 651
789 652
790 653
791 654
792 655
793 656
794 657
795 658
796 659
797 660
798 661
799 662
800 663
801 664
802 665
803 666
804 667
805 668
806 669
807 670
808 671
809 672
810 673
811 674
812 675
813 676
814 677
815 678
816 679
817 680
818 681
819 682
820 683
821 684
822 685
823 686
824 687
825 688
826 689
827 690
828 691
829 692
830 693
831 694
832 695
833 696
834 697
835 698
836 699
837 700
838 701
839 702
840 703
841 704
842 705
843 706
844 707
845 708
846 709
847 710
848 711
849 712
850 713
851 714
852 715
853 716
854 717
855 718
856 719
857 720
858 721
859 722
860 723
861 724
862 725
863 726
864 727
865 728
866 729
867 730
868 731
869 732
870 733
871 734
872 735
873 736
874 737
875 738
876 739
877 740
878 741
879 742
880 743
881 744
882 745
883 746
884 747
885 748
886 749
887 750
888 751
889 752
890 753
891 754
892 755
893 756
894 757
895 758
896 759
897 760
898 761
899 762
900 763
901 764
902 765
903 766
904 767
905 768
906 769
907 770
908 771
909 772
910 773
911 774
912 775
913 776
914 777
915 778
916 779
917 780
918 781
919 782
920 783
921 784
922 785
923 786
924 787
925 788
926 789
927 790
928 791
929 792
930 793
931 794
932 795
933 796
934 797
935 798
936 799
937 800
938 801
939 802
940 803
941 804
942 805
943 806
944 807
945 808
946 809
947 810
948 811
949 812
950 813
951 814
952 815
953 816
954 817
955 818
956 819
957 820
958 821
959 822
960 823
961 824
962 825
963 826
964 827
965 828
966 829
967 830
968 831
969 832
970 833
971 834
972 835
973 836
974 837
975 838
976 839
977 840
978 841
979 842
980 843
981 844
982 845
983 846
984 847
985 848
986 849
987 850
988 851
989 852
990 853
991 854
992 855
993 856
994 857
995 858
996 859
997 860
998 861
999 862
1000 863

```

## Servidor TCP (V): *servidor.c*

18

### □ La función `serverTCP (I)`

- Traduce la dirección IP del cliente a su nombre en Internet (`getnameinfo`)
  - En caso de que no sea posible anota su dirección IP en formato decimal punto (`inet_ntop`)

```

255 void serverTCP(int s, struct sockaddr_in clientaddr_in)
256 {
257     int request = 0; /* keeps count of number of requests */
258     char buf[TAM_BUFFER]; /* This example uses TAM_BUFFER byte messages. */
259     char hostname[MAXHOST]; /* remote host's name string */
260
261     int len, len1, status;
262     struct hostent *hp; /* pointer to host info for remote host */
263     long timeout; /* contains time returned by time() */
264
265     struct linger linger; /* allow a lingering, graceful close; */
266                          /* used when setting SO_LINGER */
267
268     /* Look up the host information for the remote host
269      * that we have connected with. Its internet address
270      * was returned by the accept call, in the main
271      * daemon loop above.
272      */
273
274     status = getnameinfo((struct sockaddr *)&clientaddr_in, sizeof(clientaddr_in),
275                          hostname, MAXHOST, NULL, 0, 0);
276
277     if (status) {
278         /* The information is unavailable for the remote
279          * host. Just format its internet address to be
280          * printed out in the logging information. The
281          * address will be shown in "internet dot format".
282          */
283
284         /* inet_ntop para interoperatividad con IPv6 */
285         if (inet_ntop(AF_INET, &clientaddr_in.sin_addr, hostname, MAXHOST) == NULL)
286             perror("inet_ntop\n");
287     }
288 }

```

## Servidor TCP (VI): *servidor.c*

19

### □ La función serverTCP (II)

- Se muestra (en pantalla) la dirección IP del cliente (hostname), el número de puerto del cliente (ahora convertido a variable del host local, ntohs) y la hora de llegada
- Configura el socket para un cierre ordenado (setsockopt)

```

330 time (&timevar);
331 /* The port number must be converted first to host byte
332  * order before printing. On most hosts, this is not
333  * necessary, but the ntohs() call is included here so
334  * that this program could easily be ported to a host
335  * that does require it.
336  */
337 printf("Startup from %s port %u at %s",
338        hostname, ntohs(clientaddr_in.sin_port), (ohar *) ctime(&timevar));
339
340 /* Set the socket for a lingering, graceful close.
341  * This will cause a final close of this socket to wait until all of the
342  * data sent on it has been received by the remote host.
343  */
344 linger.l_onoff = 1;
345 linger.l_linger = 1;
346 if (setsockopt(s, SOL_SOCKET, SO_LINGER, &linger,
347              sizeof(linger)) == -1) {
348     errout(hostname);
349 }
350

```

## Servidor TCP (VII): *servidor.c*

20

### □ La función serverTCP (III)

- Comienza el diálogo recibiendo datos del cliente (recv) a través del socket (s)
  - Los datos son almacenados en una cadena de caracteres (buf)
  - La función devuelve el número de bytes recibidos
- La sentencia sleep(1) representa las tareas que tuviera que hacer el servidor
- Envía datos (send) a través del socket (s)
  - La variable buf representa los datos enviados
  - La función devuelve el número de bytes enviados

```

361 while (len = recv(s, buf, TAM_BUFFER, 0)) {
362     if (len == -1) errout(hostname); /* error from recv */
363     /* The reason this while loop exists is that there
364      * is a remote possibility of the above recv returning
365      * less than TAM_BUFFER bytes. This is because a recv returns
366      * as soon as there is some data, and will not wait for
367      * all of the requested data to arrive. Since TAM_BUFFER bytes
368      * is relatively small compared to the allowed TCP
369      * packet sizes, a partial receive is unlikely. If
370      * this example had used 32K bytes requests instead,
371      * a partial receive would be far more likely.
372      * This loop will keep receiving until all TAM_BUFFER bytes
373      * have been received, thus guaranteeing that the
374      * next recv at the top of the loop will start at
375      * the beginning of the next request.
376      */
377     while (len < TAM_BUFFER) {
378         len1 = recv(s, &buf[len], TAM_BUFFER-len, 0);
379         if (len1 == -1) errout(hostname);
380         len += len1;
381     }
382     /* Increment the request count. */
383     request++;
384     /* This sleep simulates the processing of the
385      * request that a real server might do.
386      */
387     sleep(1);
388     /* Send a response back to the client. */
389     if (send(s, buf, TAM_BUFFER, 0) != TAM_BUFFER) errout(hostname);
390 }

```

## Servidor TCP (y VIII): servidor.c

21

### □ La función serverTCP (IV)

- Una vez terminado el diálogo el socket (s) se cierra (close)
- Nuevamente se muestra en pantalla la dirección IP del cliente (hostname), el número de puerto del cliente (ahora convertido a variable del host local, ntohs) y la hora de llegada

```

392  /* The loop has terminated, because there are no
393  * more requests to be serviced. As mentioned above,
394  * this close will block until all of the sent replies
395  * have been received by the remote host. The reason
396  * for lingering on the close is so that the server will
397  * have a better idea of when the remote has picked up
398  * all of the data. This will allow the start and finish
399  * times printed in the log file to reflect more accurately
400  * the length of time this connection was used.
401  */
402  close(s);
403
404  /* Log a finishing message. */
405  time (timevar);
406  /* The port number must be converted first to host byte
407  * order before printing. On most hosts, this is not
408  * necessary, but the ntohs() call is included here so
409  * that this program could easily be ported to a host
410  * that does require it.
411  */
412  printf("Completed %s port %d, %d requests, at %s\n",
413        hostname, ntohs(clientaddr_in.sin_port), reqcnt, (char *) ctime(timevar));
414 }
415
416 /* This routine aborts the child process attending the client.
417 */
418 void abort(char *hostname)
419 {
420     printf("Connection with %s aborted on error\n", hostname);
421     exit(1);
422 }
423
424

```

## Cliente TCP (I): clienttcp.c

22

- Inicializar la estructura sockaddr\_in con los datos del servidor al que desea conectarse
  - Familia de direcciones
  - Dirección IP
    - El cliente recoge como argumento del main el nombre del servidor al que desea conectarse
    - Obtener la IP asociada al nombre (getaddrinfo)
  - Número de puerto por el que escucha el servidor (htons)
- Crear el socket TCP local (s)
  - SOCK\_STREAM

```

66  /* Set up the peer address to which we will connect. */
67  struct sockaddr_in serveraddr_in;
68  /* Get the host information for the hostname that the
69  * user passed in.
70  */
71  memset (&serveraddr_in, 0, sizeof (serveraddr_in));
72  serveraddr_in.sin_family = AF_INET;
73  /* This function as of recommended para la compatibilidad con IPv6 gethostbyname queda obsoleta */
74  struct hostent *hp = gethostbyname (argv[1]);
75  if (hp == NULL) {
76      /* There was not found. Return a
77       * special value signifying the
78       * error.
79       */
80      fprintf(stderr, "No se pudo resolver la IP de %s\n",
81            argv[1], argv[1]);
82      exit(1);
83  }
84  /* Copy address of host */
85  serveraddr_in.sin_addr = *((struct sockaddr_in *) hp->ai_addr);
86  /* Set the port number */
87  if (hp->ai_family == AF_INET) {
88      /* If hp == NULL,
89       * fprintf(stderr, "No se pudo resolver la IP de %s\n",
90       * argv[1], argv[1]);
91       * exit(1);
92       */
93      serveraddr_in.sin_port = htons(PORT);
94  }
95  /* Create the socket. */
96  s = socket (AF_INET, SOCK_STREAM, 0);
97  if (s == -1) {
98      fprintf(stderr, "No se pudo crear el socket\n", argv[1]);
99      exit(1);
100 }

```

## Cliente TCP (II): clienttcp.c

23

- Conectar con la dirección del socket remoto (connect)
  - ▣ Si todo va bien la conexión queda establecida
- Si se necesita puede obtenerse la información del socket creado localmente en la máquina del cliente (getsockname)
  - ▣ Rellena una estructura sockaddr\_in con la información del socket local

```

187  /* Try to connect to the remote server at the address
188   * which was just built into peeraddr.
189   */
190  if (connect(s, (const struct sockaddr *)&peeraddr_in, sizeof(struct sockaddr_in))
191      perror(argv[0]);
192      fprintf(stderr, "%s: unable to connect to remote\n", argv[0]);
193      exit(1);
194  }
195
196  /* Since the connect call assigns a file address
197   * to the local end of this connection, let's use
198   * getsockname to see what is assigned. Note that
199   * address needs to be passed in as a pointer,
200   * because getsockname returns the actual length
201   * of the address.
202   */
203  addrlen = sizeof(struct sockaddr_in);
204  if (getsockname(s, (struct sockaddr *)&myaddr_in, &addrlen) == -1) {
205      perror(argv[0]);
206      fprintf(stderr, "%s: unable to read socket address\n", argv[0]);
207      exit(1);
208  }
209
210  /* Print out a startup message for the user. */
211  time(&timevar);
212  /* The port number must be converted first to host byte
213   * order before printing. On most hosts, this is not
214   * necessary, but the ntohs() call is included here so
215   * that this program could easily be ported to a host
216   * that does require it.
217   */
218  printf("Connected to %s on port %u at %s",
219         argv[1], ntohs(myaddr_in.sin_port), (char *) ctime(&timevar));
220

```

## Cliente TCP (y III): clienttcp.c

24

- Ahora entramos en un diálogo con el servidor enviando datos (send) y recibiendo (recv)
- Se puede indicar al servidor la terminación de la fase de envío de datos (shutdown)
- Una vez terminado el diálogo se recomienda cerrar ordenadamente el socket local (close)

```

146  for (i=1; i<=5; i++) {
147      *buf = i;
148      if (send(s, buf, 10, 0) != 10) {
149          fprintf(stderr, "%s: Connection aborted on error ",
150                  argv[0]);
151          fprintf(stderr, "on send number %d\n", i);
152          exit(1);
153      }
154  }
155
156  /* Now, shutdown the connection for further sends.
157   * This will cause the server to receive an end-of-file
158   * condition after it has received all the requests that
159   * have just been sent, indicating that we will not be
160   * sending any further requests.
161   */
162  if (shutdown(s, 1) == -1) {
163      perror(argv[0]);
164      fprintf(stderr, "%s: unable to shutdown socket\n", argv[0]);
165      exit(1);
166  }
167
168  /* Now, start receiving all of the replies from the server.
169   * This loop will terminate when the recv returns zero,
170   * which is an end-of-file condition. This will happen
171   * after the server has sent all of its replies, and closed
172   * its end of the connection.
173   */
174  while (i = recv(s, buf, 10, 0)) {
175      if (i == -1) {
176          perror(argv[0]);
177          fprintf(stderr, "%s: error reading result\n", argv[0]);
178          exit(1);
179      }
180  }

```

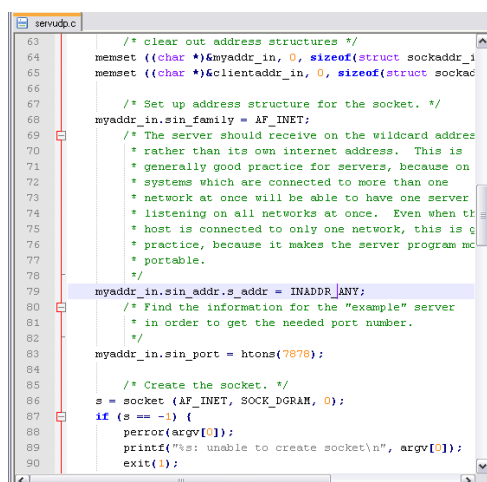
## UDP

Ejemplos del servidor y cliente UDP  
(servidor.c y clientudp.c)

### Servidor UDP(I): *servidor.c*

26

- Crear el socket local (s)
  - ▣ Inicializar la estructura `sockaddr_in` con la información del socket
    - Familia de direcciones
    - Dirección IP por la que escuchar
    - Número de puerto por el que escuchar
  - ▣ Obtener el descriptor del socket UDP (socket)
    - `SOCK_DGRAM`



```

63  /* clear out address structures */
64  memset(&myaddr_in, 0, sizeof(struct sockaddr_in));
65  memset(&clientaddr_in, 0, sizeof(struct sockaddr_in));
66
67  /* Set up address structure for the socket. */
68  myaddr_in.sin_family = AF_INET;
69  /* The server should receive on the wildcard address
70   * rather than its own internet address. This is
71   * generally good practice for servers, because on
72   * systems which are connected to more than one
73   * network at once will be able to have one server
74   * listening on all networks at once. Even when the
75   * host is connected to only one network, this is a
76   * practice, because it makes the server program more
77   * portable.
78   */
79  myaddr_in.sin_addr.s_addr = INADDR_ANY;
80  /* Find the information for the "example" server
81   * in order to get the needed port number.
82   */
83  myaddr_in.sin_port = htons(7876);
84
85  /* Create the socket. */
86  s = socket(AF_INET, SOCK_DGRAM, 0);
87  if (s == -1) {
88      perror(argv[0]);
89      printf("%s: unable to create socket\n", argv[0]);
90      exit(1);
  
```

## Servidor UDP(II): *servidor.c*

27

- La inicialización en UDP termina con el bind, que asocia el socket local (s) con la información indicada en la estructura sockaddr\_in

```

92  /* Bind the server's address to the socket. */
93  if (bind(s, (struct sockaddr *) &sockaddr_in, sizeof(struct sockaddr_in)) != 0) {
94      perror(argv[0]);
95      printf("Unable to bind address\n", argv[0]);
96      exit(1);
97  }
98

```

## Servidor (III): *servidor.c*

28

- Convertir el servidor en un proceso demonio
  - Se desvincula el proceso del terminal abierto (setpgrp)
  - Se crea el proceso que hará las funciones de servidor (fork)
  - Se ignora la señal SIGCHLD (sigaction) para evitar la acumulación de procesos zombi al morir el proceso padre
  - Se crea un bucle (for) infinito para que el proceso esté siempre ejecutándose (daemon)

```

99  /* Now, all the initialization of the server is
100  * complete, and any user errors will have already
101  * been detected. Now we can fork the daemon and
102  * return to the user. We need to do a setpgrp
103  * so that the daemon will no longer be associated
104  * with the user's control terminal. This is done
105  * before the fork, so that the child will not be
106  * a process group leader. Otherwise, if the child
107  * were to open a terminal, it would become associated
108  * with that terminal as its control terminal. It is
109  * always best for the parent to do the setpgrp.
110  */
111  setpgrp();
112
113  switch (fork()) {
114      case -1: /* Unable to fork, for some reason. */
115          perror(argv[0]);
116          printf("Unable to fork daemon\n", argv[0]);
117          exit(1);
118
119      case 0: /* The child process (daemon) comes here. */
120          /* Close stdin and stdout so that they will not
121           * be kept open. Stdout is assumed to have been
122           * redirected to some logging file, or /dev/null.
123           * From now on, the daemon will not report any
124           * error messages. This means the daemon does not
125           * have to make wait calls to clean them up.
126           */
127          close(stdin);
128          close(stdout);
129
130          /* Set SIGCHLD to SIG_IGN, in order to prevent
131           * the accumulation of zombies as each child
132           * terminates. This means the daemon does not
133           * have to make wait calls to clean them up.
134           */
135          sigaction(SIGCHLD, &sa, NULL);
136
137          for(;;) {

```

## Servidor UDP(III): *servidor.c*

29

- El servidor UDP responde al cliente con la dirección IP asociada a un nombre de dominio que previamente ha recibido del cliente
  - La función `recvfrom` además de recibir los datos del cliente (`buffer`), rellena otra estructura `sockaddr_in` (`clientaddr_in`) con los datos del socket del cliente para poder contestarle
  - Destacar que el último argumento es un puntero (`addr1en`)

```

134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157

```

```

/* Note that addr1en is passed as a pointer
 * so that the recvfrom call can return the
 * size of the returned address.
 */
addr1en = sizeof(struct sockaddr_in);
/* This call will block until a new
 * request arrives. Then, it will
 * return the address of the client,
 * and a buffer containing its request.
 * BUFFER_SIZE - 1 bytes are read so that
 * room is left at the end of the buffer
 * for a null character.
 */
cc = recvfrom(s, buffer, BUFFER_SIZE - 1, 0,
              (struct sockaddr *)&clientaddr_in, &addr1en);
if (cc == -1) exit(1);
/* Make sure the message received is
 * null terminated.
 */
buffer[cc] = '\0';
/* Treat the message as a string containing
 * a hostname. Search for the name in
 * /etc/hosts.
 */

```

## Servidor UDP(y IV): *servudp.c*

30

- La función `getaddrinfo` hace el mejor esfuerzo para traducir el nombre en su dirección IP correspondiente (buscando primero en el fichero `/etc/hosts` y en caso de no obtener resultado haciendo una petición al servidor de DNS)
- Por último, se envía la dirección IP solicitada (`reqaddr`, una estructura de tipo `in_addr`)

```

64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99

```

```

errcode = getaddrinfo(buffer, NULL, &hints, &res);
if (errcode != 0)
{
    /* Name was not found. Return a
     * special value signifying the
     * error.
     */
    reqaddr.s_addr = ADDRNOTFOUND;
}
else {
    /* Copy address of host into the
     * return buffer.
     */
    reqaddr = ((struct sockaddr_in *) res->ai_addr)->in_addr;
}
freeaddrinfo(res);
/*
hp = gethostbyname(buffer);
if (hp == NULL) {
    reqaddr.s_addr = ADDRNOTFOUND;
} else {
    reqaddr.s_addr = ((struct in_addr *) (hp->h_addr))->s_addr;
}
*/
/* Send the response back to the
 * requesting client. The address
 * is sent in network byte order. Note that
 * all errors are ignored. The client
 * will retry if it does not receive
 * the response.
 */
nc = sendto(s, reqaddr, sizeof(struct in_addr),
            0, (struct sockaddr *)&clientaddr_in, addr1en);
if (nc == -1) {
    perror("serverUDP");
    printf("In: sendto error\n", "serverUDP");
    return;
}

```

## Cliente UDP (I): clientudp.c

31

- Inicializar `servaddr_in` (estructura `sockaddr_in`) con los datos del servidor al que desea conectarse

- ▣ Familia de direcciones

- ▣ Dirección IP

- El cliente recoge como argumento del `main` el nombre del servidor al que desea conectarse

- Obtiene la IP asociada al nombre (`getaddrinfo`)

- ▣ Número de puerto por el que escucha el servidor (`htons`)

- Crear el socket UDP local (`s`)

- ▣ `SOCK_DGRAM`

```

119 memset(&servaddr_in, 0, sizeof(struct sockaddr_in));
120 memset(&servaddr_in, 0, sizeof(struct sockaddr_in));
121 address = malloc(sizeof(struct sockaddr_in));
122
123 /* Set up the server address. */
124 servaddr_in.sin_family = AF_INET;
125 /* Set the host information for the server's hostname that the
126    * user passed in.
127    */
128 memset(&hints, 0, sizeof(hints));
129 hints.ai_family = AF_INET;
130
131 /* Note: función no se recomienda para la compatibilidad con IPv6 pathosystems queda obsoleta */
132 errorcode = getaddrinfo(argv[1], NULL, &hints, &res);
133 if (errorcode != 0)
134 {
135     /* Error was not found. Return a
136        * special value signifying the
137        * error.
138        */
139     fprintf(stderr, "No se ha podido resolver la IP de %s\n",
140             argv[1], argv[1]);
141     exit(1);
142 }
143 else {
144     /* Copy address of host */
145     servaddr_in.sin_addr = (struct sockaddr_in *) gse->ai_addr->sin_addr;
146     freeaddrinfo(res);
147     servaddr_in.sin_port = htons(PORT);
148
149     /* Create the socket. */
150     s = socket(AF_INET, SOCK_DGRAM, 0);
151     if (s == -1) {
152         perror(argv[1]);
153         fprintf(stderr, "No se puede crear socket\n", argv[1]);
154         wait();
155     }
156 }

```

## Cliente UDP (II): clientudp.c

32

- En este caso es obligatorio el uso de la función `bind` para asociar el socket local (`s`) con la información del socket contenida en la estructura `sockaddr_in`

- ▣ La inicialización de la estructura siempre requiere

- la familia de direcciones (`AF_INET`),

- un número de puerto para el cliente (un 0 significa que se escoja un puerto aleatorio libre)

- dirección IP del cliente (`INADDR_ANY`)

- ▣ El `bind` rellena la estructura con el número de puerto efímero escogido y dirección IP del cliente

```

119 /* Bind socket to some local address so that the
120    * server can send the reply back. A port number
121    * of zero will be used so that the system will
122    * assign any available port number. An address
123    * of INADDR_ANY will be used so we do not have to
124    * look up the internet address of the local host.
125    */
126 myaddr_in.sin_family = AF_INET;
127 myaddr_in.sin_port = 0;
128 myaddr_in.sin_addr.s_addr = INADDR_ANY;
129 if (bind(s, (const struct sockaddr *) &myaddr_in, sizeof(struct
130         perror(argv[0]);
131         fprintf(stderr, "%s: unable to bind socket\n", argv[0]);
132         exit(1);
133     }
134 }

```



## Cliente UDP (y III): clientudp.c

33

### Funciones para envío y recepción son

- sendto y recvfrom

### El cliente

#### Envía con sendto

- El segundo argumento son los datos que se desean enviar
- El quinto argumento (servaddr\_in) contiene los datos del socket remoto

#### Recibe con recvfrom recogiendo la dirección del socket remoto

- Hay que habilitar mecanismos de *timeout* puesto que *recvfrom* es bloqueante y la respuesta puede no llegar nunca

```
while (n_retry > 0) {
    /* Send the request to the nameserver. */
    if (sendto(s, argv[2], strlen(argv[2]), 0, (struct sockaddr *)&servaddr_in,
              sizeof(struct sockaddr_in)) == -1) {
        perror(argv[0]);
        fprintf(stderr, "%s: unable to send request\n", argv[0]);
        exit(1);
    }

    /* Set up a timeout so I don't hang in case the packet
     * gets lost. After all, UDP does not guarantee
     * delivery.
     */
    alarm(TIMOUT);
    /* Wait for the reply to come in. */

    if (recvfrom(s, &reqaddr, sizeof(struct in_addr), 0,
                (struct sockaddr *)&servaddr_in, &addrlen) == -1) {
        if (errno == EINTR) {
            /* Alarm went off and aborted the receive.
             * Send to retry the request if we have
             * not already exceeded the retry limit.
             */
            printf("Attempt %d (retries %d).\n", n_retry, RETRIES);
            n_retry--;
        }
        else {
            printf("Unable to get response from\n");
            exit(1);
        }
    }
    else {
        alarm(0);
        /* Print out response. */
        if (reqaddr.sin_addr == ADDRNOTFOUND)
            printf("Address not found\n");
        else
            printf("Address found\n");
    }
}
```

## Servidor – Multiplexación de entrada/salida: servidor.c

34

### Función select

- Crea el conjunto de sockets o descriptores (macros `FD_ZERO` y `FD_SET`) y se selecciona (función `select`) el socket que ha cambiado de estado (habitualmente por la llegada de datos)
- La detección del socket que se ha activado se realiza con la macro `FD_ISSET`
  - Un bloque para procesar la recepción de datos a través del socket UDP (ds)
    - ServerUDP
  - Otro bloque distinto para procesar la recepción de datos a través del socket TCP (ss)
    - accept crea un nuevo socket (s)
    - serverTCP

```
if ( (numfds = select(getdtablesize(), &readmask, (fd_set *)0, (fd_set *)0, &timeout)) < 0) {
    if (errno == EINTR) {
        FPM++;
        perror("select failed\n");
    }
}
else {
    /* Compruebo si el socket seleccionado es el socket TCP */
    if (FD_ISSET(&ds, &readmask)) {
        s = accept(&ds, (struct sockaddr *)&clientaddr_in, &addrlen);
        if (s == -1) exit(1);
        switch (fork()) {
            case -1: /* Can't fork, just exit. */
                exit(1);
            case 0: /* Child process comes here. */
                close(&ds); /* Close the listen socket inherited from the daemon. */
                serverTCP(s, clientaddr_in);
                exit(0);
            default: /* Daemon process comes here. */
                close(&ds);
        }
    }
    /* De TCP */
    /* Compruebo si el socket seleccionado es el socket UDP */
    if (FD_ISSET(&ss, &readmask)) {
        ss = recvfrom(&ss, buffer, BUFFER_SIZE - 1, 0,
                     (struct sockaddr *)&clientaddr_in, &addrlen);
        if (ss == -1) {
            perror(argv[0]);
            printf("ss: recvfrom error\n", argv[0]);
            exit(1);
        }
        /* Make sure the message received is
         * null terminated.
         */
        buffer[ss] = '\0';
        serverUDP(&ss, buffer, clientaddr_in);
    }
}
```

## Consideraciones

35

- Las funciones `send`, `recv`, `sendto` y `recvfrom` permiten enviar y recibir datos de cualquier tipo (`void *`)
  - ▣ Se pueden enviar cadenas de caracteres, estructuras, enteros, etc.
  - ▣ Sin embargo hay que tener presente que, dado que las arquitecturas de las máquinas cliente y servidor pueden ser distintas, **para garantizar la comunicación y portabilidad del código, se recomienda usar siempre cadenas de caracteres** (o estructuras con miembros de tipo cadena)
    - En caso de usar elementos de tipo numérico utilizar las funciones `htons` (o `htonl`) en el envío y `ntohs` (ó `ntohl`) en la recepción