#### Guía de ejercicios: Programación con sockets

La finalidad de estos ejercicios es familiarizarse con los conceptos básicos de programación con sockets y la resolución de nombres, conceptos básicos para poder afrontar la resolución del TPE

1. Realizar distintos filtros en addrinfo.c (modificando la variable hint) para que sólo obtenga direcciones IPv4 (o sólo IPv6), sólo protocolos orientados a conexión, etc.
2. Intentar ejecutar tcpEchoServer para que escuche en el puerto 7 (el puerto estándar para Echo). ¿Qué mensaje recibe? (Nota: si no recibe error es porque está logueado como root, lo cual no es una buena costumbre)
3. Probar la ejecución de tcpEchoServer utilizando strace o similar en un puerto que esté libre.
   1. Verificar con nmap que el servidor esté activo.
   2. Usar netcat como cliente echo, probar con distintas IPs, tanto v4 como v6. ¿Funciona con ambas o solo con una?
   3. En otra consola intentar ejecutar el servidor en el mismo puerto. ¿Qué sucede?
   4. Conectarse desde otra consola usando nc. Analizar lo informado por strace
   5. Sin cerrar la sesión del punto anterior, usar nc desde otra consola para conectarse al servicio. ¿Qué sucede? Cerrar la sesión del punto c, ¿qué sucede?
4. tcpEchoClient.c y tcpEchoServer.c
   1. ¿Se podría hacer que el servicio escuche en más de un puerto?
   2. Suponiendo que la red local es 192.168.1.0/24 ¿Se podría modificar para que el servidor sólo escuche pedidos de la red local? Si la respuesta es no, ¿podría sólo atender los pedidos provenientes de la red local?
   3. ¿Qué sucede si un cliente no cierra la conexión?
   4. ¿Qué sucede si el servidor en la respuesta envía octetos de menos?
   5. ¿Qué sucede si el servidor en la respuesta octetos de más?
   6. Modificar el protocolo (programa cliente y servidor) para que el servidor, antes de leer lo que el cliente envía, le envíe un mensaje de bienvenida. Una vez que el cliente leyó el mensaje de bienvenida envía su mensaje y espera por la respuesta. ¿Puede el servidor decidir en forma arbitraria el mensaje o debe haber un acuerdo previo sobre las características del mismo?
   7. Modificar TCPEchoClient4.c para que use IP y puerto específico de origen (usar bind()). Ver qué sucede si elige una IP inválida o un puerto en uso.
   8. ¿Qué sucedería si un servidor TCP en vez de usar un socket pasivo para atender conexiones y un socket activo para usar esa nueva conexión, pudiera usar un único socket? Esto es, que pudiera crear un *socket server*, llamar a bind() y listen(), esperar por una comunicación, enviar y recibir a través de ese socket y cuando termine cierre y repita el proceso?
   9. Modificar en tcpServerUtil.c la siguiente línea

          for (struct addrinfo \*addr = servAddr; addr != NULL && servSock == -1; addr = addr->ai\_next) {

para que quede así

         for (struct addrinfo \*addr = servAddr; addr != NULL ; addr = addr->ai\_next) {

y hacer los ajustes necesarios para que sólo retorne -1 si NUNCA servSock llegó a tomar un valor distinto de -1.

Y con estos cambios repetir el ejercicio 3. ¿Qué cambió en el comportamiento del servidor? ¿Qué habría que cambiar para que responda (no solo escuche) peticiones tanto de IPv4 como ipV6?

1. En base a los códigos de ejemplo, implementar cliente y servidor para un protocolo orientado a conexión “Echo modificado”, que tenga las siguientes diferencias con el Echo original
   1. Si una línea (cada línea está finalizada por ‘\n’) supera los 64 caracteres, le envíe al cliente los primeros 64 caracteres y un ‘\n’, ignorando el resto
   2. Una vez que el servidor acepta una conexión con un cliente, aplicará en forma opcional un timeout a las lecturas. La cantidad de segundos no estará fijada por el protocolo, cada servidor puede decidir cuántos segundos serán. Si pasado ese timeout no se reciben datos por parte del cliente, el servidor cerrará la conexión. Ver <https://stackoverflow.com/questions/2876024/linux-is-there-a-read-or-recv-from-socket-with-timeout>
2. udpEchoClient.c y udpEchoServer.c
   1. ¿Qué cambios habría que hacer si quisiéramos aceptar mensajes arbitrariamente largos (que no entren en un solo datagrama)? Asumir que no habrá pérdida y los datagramas se reciben en forma ordenada.
   2. ¿Sería correcto usar este cliente en producción? Pista: qué sucede si el mensaje enviado o la respuesta se pierde.
   3. ¿Cómo se podría usar estos códigos para determinar empíricamente el tamaño máximo de un datagrama? Determinar si el máximo es el mismo para IPv4 e IPv6.
   4. Realizar cambios en el código del servidor para que envíe menos bytes y también para que envíe más bytes de los recibidos, ¿qué sucede en cada caso?
   5. Pensar qué cambios habría que hacer si se quieren enviar y recibir mensajes arbitrariamente largos y que el protocolo sea confiable.
   6. Realizar los cambios en udpClientEcho.c para que no tome la primer entrada que retorna getaddrinfo sino que tome la primera que funcione, como se hizo con el cliente TCP
3. Se diseña un protocolo de aplicación para dar soporte a un horóscopo diario. El mismo se implementa sobre UDP y tiene un esquema sencillo: se recibe un datagrama con el signo, y si el mismo es correcto se le envía en un único datagrama el horóscopo del día, y luego se espera un ACK de ese mismo cliente para verificar que haya llegado correctamente. Para ello se diseña el siguiente algoritmo que implementa el servidor de horóscopos.

Nota: por simplificación se han omitido todos los controles de errores al enviar o recibir.

socket = …

while (true ) {

struct sockaddr\_storage clntAddr;

// En buffer se lee el signo

// clntAddr: datos del cliente

ok = recvFrom(socket, buffer, MAX\_LENGTH, 0,

&clntAddr, sizeof(clntAddr));

// llenamos buffer con el horóscopo para ese signo

// o un mensaje de error si el signo no existe

…

// Enviamos horóscopo del día o mensaje de error

sendTo(socket, buffer, bytesBuffer, 0, &clntAddr, sizeof(clntAddr));

// Leemos el ACK

ok = recvFrom(socket, buffer, MAX\_LENGTH, 0, &clntAddr, sizeof(clntAddr));

}

1. Detallar los errores **de diseño** cometidos en el algoritmo (o sea el programa, no el protocolo).
2. Explicar los cambios que habría que hacer para corregir los errores en el programa, NO en el protocolo.

8. En tcpEchoAddrInfo.c se realiza una implementación no bloqueante para el envío de la respuesta, haciendo que lo leído se almacene en un buffer asociado al socket y que sea enviado recién cuando el socket tenga espacio para la escritura, y al hacer el write se lo hace no bloqueante y se tiene en cuenta cuántos bytes se pudieron enviar. Ahora bien, ¿qué sucede si el cliente sigue enviando información pero no consume lo recibido? ¿Tiene sentido que el servidor siga leyendo y almacenando en el buffer propio lo enviado por ese cliente? Realizar los cambios necesarios para limitar el tamaño del buffer.