Sockets - Recap - Networking

Collapse context

Sin entrar en detalle, veremos un par de herramientas para practicar sockets TCP y el debuggeo del mismo.

En este *recap* veras:

- `netcat`
- `netstat` / `ss`
- `tiburoncin`

Prestá particular atención a `tiburoncin` ya que el Sercom lo usará como parte de los tests automáticos.

Verás algo de sockets aunque mucho del material lo veras en las clases y códigos de ejemplo.

Los sockets son fundamentales en el mundo actual: todo esta conectado y TCP/IP es la red que lo conecta.

Incluso si en el resto de la carrera o en tu vida profesional no usas sockets directamente, estaras usando librerias que los usan.

Saber que son, como usarlos y como debuggerlos es importantísimo.

El recap finaliza con algunos detalles sobre protocolos. En el trabajo práctico de sockets se te dará el protocolo y solo tendrás que implementarlo pero entenderlo y saber el por qué de las cosas te será muy útil cuando diseñes tu propio protocolo.

¡Tu respuesta es exitosa! Tu calificación es 100.0%. [Tarea #66db407fa455aa3987341f2e] ×

Pregunta 1:

`netcat` o `nc` te permite establecer una comunicación TCP emulando ser un cliente o un servidor.

Esto es, con ciertos flags `netcat` estará a la *espera* de una conexión mientras que con otros flags `netcat` será quien *inicie* la conexión.

Cuales de las siguientes observaciones son correctas? Hay mas de una, **marcarlas todas!**

Tene en cuenta que hay **varias** versiones de `netcat` en los distintos linuxes. Algunas requieren el flag `-p` cuando se lo combina con el flag `-l`; otras en cambio prohiben la combinacion de los flags `-p` y `-l`.

Te dejo las manpages de 2 versiones de netcat. Fijate cual anda en tu máquina!

Referencias:
- manpage netcat (a)
- manpage netcat (b)

☑ `nc google.com 80` se conecta a un servidor de Google

☐ `nc -k foo.com bar.com baz.com` pone el netcat en modo keep para conectarse a multiples servers a la vez.

☑ `nc -l -p http-alt` pone el netcat en *espera*, *escuchando* en el servicio http (puerto 8080). Nota: puede que el flag `-p` no lo necesites segun la version de tu netcat.

☑ `nc 127.0.0.1 8080` hace que el netcat se conecte a un servidor en localhost que esta escuchando en el puerto 8080.

Pregunta 2:

En una consola ejecutá `nc -l -p 8080` y en *otra* consola ejecutá `nc 127.0.0.1 8080`.

Si ingresas por entrada estandar una frase, que sucede?

Nota: puede que el flag `-p` no lo necesites segun la versión de tu netcat.

● Si ingresas la frase en *cualquiera* de los 2 netcats esta frase aparece en el otro netcat ya que la comunicacion entre los 2 netcats es bidireccional (no importa quien se conectó ni quien estaba escuchando).

○ Si ingresas la frase en el netcat `nc 127.0.0.1 8080` esta frase aparece en el otro netcat pero si ingresas la frase en `nc -l -p 8080` esta frase *no* aparece en el otro netcat por que la comunicacion entre los 2 netcats solo funciona en un solo sentido (de quien se conecta hacia quien estaba escuchando).

Pregunta 3:

Suponete que queres enviar un archivo por internet desde tu computadora local a un servidor con hostname `alice.com`.

Que deberías ejecutar?

Nota: puede que el flag `-p` no lo necesites segun la versión de tu netcat.

○ En la máquina remota ejecutas `nc localhost 8080 > somefile` y en tu máquina local ejecutas `nc -l -p 8080 < somefile`.

○ En la máquina remota ejecutas `nc -l -p 8080 > somefile` y en tu máquina local ejecutas `nc localhost 8080 < somefile`.

● En la máquina remota ejecutas `nc -l -p 8080 > somefile` y en tu máquina local ejecutas `nc alice.com 8080 < somefile`.

Pregunta 4:

Tanto `ss` como `netstat` te muestran el estado de las conexiones TCP y UDP de toda tu máquina. Vas a ver muchas cosas!

Son herramientas simples pero te permitirán ver si tus programas estan o no levantando los sockets en los puertos correctos.

Lo primero que tenes que revisar cuando algo no anda!

Deja corriendo en una consola `nc -l -p 9000` y en otra consola ejecutá `ss -tuplan` (o bien `netstat -tauopen`).

Cuál es el estado de la conexión del netcat sobre el puerto 9000? (el nombre exacto dependerá del idioma de tu máquina, aca pondré sus nombres in ingles).

Nota: puede que el flag `-p` no lo necesites segun la versión de tu netcat.

Referencias:
- manpage netstat
- manpage ss

● *LISTEN.*

○ *TIME WAIT.*

○ *CONNECTED.*

Pregunta 5:

`tiburoncin` es un pequeño *man in the middle*. Es un programa (que tendrás que compilar) que intercepta los mensajes entre un cliente y servidor e imprime su contenido.

Es muy útil para debugguear y ver **exactamente** que bytes se estan enviando por los sockets y el Sercom usa a *tiburoncin* justamente para capturar y verificar que tus programas respeten el protocolo especificado.

Supone que ejecutas en una consola `nc -l -p 9091`, en otra consola ejecutas `tiburoncin -o -A 127.0.0.1:9095 -B 127.0.0.1:9091` y en una tercer consola ejecutas `nc 127.0.0.1 9095`.

Ahora en esta última consola escribis `"taller"` seguido de un salto de línea y luego en la primer consola escribis `"41424344"`.

Qué es lo que muestra `tiburoncin`?

Nota: puede que el flag `-p` no lo necesites segun la versión de tu netcat.

Referencias:
- tiburoncin

● Se ve una salida similar a la de hexdump con los bytes enviados desde el cliente y desde el servidor.

○ No se imprime nada. *No* hay que escribir en las consolas de los netcats sino en la consola de `tiburoncin`.

○ Se ve una salida similar a la de hexdump con la particularidad que el mensaje `"41424344"` es interpretado como hexadecimal e imprime `"ABCD"`.

Pregunta 6:

Ademas de imprimir por salida estandar, `tiburoncin` puede guardar en dos archivos los bytes enviados hacia un lado y hacia el otro.

Estos archivos tendras los bytes en hexadecimal.

Supone que ejecutas en una consola `nc -l -p 9091`, en otra consola ejecutas `tiburoncin -o -A 127.0.0.1:9095 -B 127.0.0.1:9091 -o` y en una tercer consola ejecutas `echo "foobar" | nc 127.0.0.1 9095`.

Con el flag `-o`, `tiburoncin` te habrá creado 2 archivos con los bytes enviados.

Como es posible recuperar el contenido original?

Referencias:
- manpage xxd
- manpage hexdump
- tiburoncin

○ Ejecutando `hexdump -C ElArchivoQueEscribioTiburoncin`

● Ejecutando `xxd -p -c 16 -r ElArchivoQueEscribioTiburoncin`.

Pregunta 7:

Suponete que queres enviar un mensaje de 16 bytes usando sockets.

Funciona este código?

```
bool enviar_mensaje(int fd, char* buf) {
    int ret = send(fd, buf, 16, 0);
    if (ret == -1)
       return false; // hubo un error

    return true; // el mensaje fue enviado
}
```

Cuales de las siguientes observaciones son correctas? Hay mas de una, **marcarlas todas!**

☑ Falta el flag *MSG_NOSIGNAL*. Sin él el sistema operativo podría matar el programa con un *SIGPIPE*.

☐ *send* retorna los bytes enviados. El código deberia checkear `ret == 16` y fallar sino (si *send* retorna menos bytes de 16 es que hubo un error totalmente inesperado)

☑ *send* puede no fallar y aun asi enviar menos de 16 bytes (es normal).

☑ *send* retorna -1 si se cerro el socket. Si es esperado o no dependera del protocolo (puede significar un error o no).

☑ Aunque podría funcionar, es simpre preferible pasar el size por parámetro (faltaría el parámetro en `enviar_mensaje`)

☑ *send* sufre de *short write*. Si se quieren enviar exactamente 16 bytes hay que usar un loop.

Pregunta 8:

Suponete que queres recibir un mensaje usando sockets. El mensaje tiene 2 partes: 1 byte con la longitud del mensaje y el mensaje en si.

Funciona este código?

```
bool recibir_mensaje(int fd, char* buf) {
    uint8_t sz; int ret;

    ret = recv(fd, &sz, 1, 0);
    if (ret == 0)
       return false; // hubo un error

    ret = recv(fd, buf, sz, 0);
    if (ret == 0)
       return false; // hubo un error

    return true; // el mensaje recibido
}
```

Cuales de las siguientes observaciones son correctas? Hay mas de una, **marcarlas todas!**

☐ *recv* retorna -1 si se cerro el socket.

☑ *recv* puede no fallar y aun asi puede recibir menos de `sz` bytes (es normal).

☐ *recv* retorna los bytes recibidos. El código deberia checkear `ret == sz` y fallar sino (si *recv* recibe menos bytes es que hubo un error)

☑ Aunque podría funcionar, el código podría terminar en un buffer overflow. Como se podría garantizar que el buffer tiene suficiente espacio si *recibir_mensaje* no recibe el size?

☑ *recv* retorna 0 si se cerro el socket. Si se recibe justo **luego** de leer el size pero **antes** del mensaje, seguro que es un error por que se estaria recibiendo un mensaje partido

☐ Si se quieren recibir exactamente 1 byte para recibir el size del mensaje hay que usar un loop.

☑ *recv* sufre de *short read*. Si se quieren recibir exactamente `sz` bytes hay que usar un loop.

☐ Falta el flag *MSG_NOSIGNAL*. Sin él el sistema operativo podría matar el programa con un *SIGPIPE*.

☑ *recv* retorna 0 si se cerro el socket. Si se recibe justo **antes** de leer el size podría ser algo esperado o podría no serlo y dependera del protocolo si es un error o no.

Pregunta 9:

Suponete que queres enviar un mensaje usando Python.

Sufre *socket.send* en Python de un short write?

```
skt.send(b"hello Python")
```

Y en Ruby?

```
skt.send "hello Ruby", 0
```

Cuales de las siguientes observaciones son correctas? Hay mas de una, **marcarlas todas!**

Referencias:
- Python socket
- Ruby socket

☑ Ruby es un lenguaje de alto nivel y *skt.send* **no** sufre de *short writes*

☐ *skt.send* en Ruby sufre de *short writes* igual que en C++ (hay que usar loops)

☐ Python es un lenguaje de alto nivel y *skt.send* **no** sufre de *short writes*

☑ *skt.send* en Python sufre de *short writes* igual que en C++ (hay que usar loops)

Pregunta 10:

Se quiere implementar un juego multijugador y entre las muchas cosas que un cliente puede hacer esta la de mover una pieza (digamos un tanquecito) de un lugar del mapa/escenario/tablero a otro.

Claramente el cliente debe comunicarle esta acción al servidor para que este la valide, por ejemplo que un jugador no pueda mover una pieza que no es suya o que no existe.

Suponete que los desarrolladores estan pensando en enviar desde el cliente un único mensaje para realizar esta acción y han pensado/diseñado los siguientes posibles mensajes.

```
msj 1:  <acción> <pieza id> <origen x> <orig

msj 2:  <acción> <destino x> <destino y>

msj 3:  <pieza id> <destino x> <destino y>
```

Cuales de las siguientes observaciones son correctas? Hay mas de una, **marcarlas todas!**

Esta pregunta te podra parecer abstracta pero trata de pensarla. Qué harías vos si recibis uno de esos mensajes? Tendrías la info necesaria para saber qué hacer, sobre qué objeto y toda la info para realizar la acción?

☑ Es necesario enviar `<acción>` para que el servidor sepa que acción hay que ejecutar (en este caso estamos hablando de mover). Si no se envia, el servidor no sabría como interpretar el resto de los bytes que constituyen el mensaje.

☑ Si se usa el mensaje 3 el servidor no tiene forma de saber que acción se quiere hacer con la pieza.

☑ Si se usa el mensaje 2 el servidor no tiene forma de saber cual es la pieza que se quiere mover.

☑ El mensaje 1 tiene informacion de más ya que la posición de origen (x,y) puede ser deducida por el servidor (el servidor sabe cual de todas las piezas hay que mover ya que el mensaje tiene el id de la pieza)

Pregunta 11:

Suponete que el juego le permite a un jugador seleccionar múltiples piezas (tanquecitos, soldados, etc) y darles una única acción, por ejemplo, moverse hacia una posición determinada del mapa.

Claramente el cliente debe comunicarle esta acción al servidor.

Suponete que los desarrolladores estan pensando en enviar desde el cliente un único mensaje para realizar esta acción y han pensado/diseñado los siguientes posibles mensajes.

```
msj 1:  <acción> <destino x> <destino y> <cr

msj 2:  <acción> <cnt> [<pieza 1 id>, <pieza

msj 3:  <acción> <cnt> [<pieza 1 id> <destir
```

Cuales de las siguientes observaciones son correctas? Hay mas de una, **marcarlas todas!**

☑ Si se usa el mensaje 2 el servidor no tiene forma de saber a dónde se quiere mover a las piezas.

☑ Si se usa el mensaje 1 el servidor puede saber que acción realizar (digamos mover), a dónde, cuantas piezas fueron seleccionadas y sus ids.

☑ Es necesario enviar `<cnt>` para que el servidor sepa cuánto más hay que leer del socket. De otro modo el servidor no sabe cuántos bytes tiene que leer (no puede saber a priori cuantas piezas vendrán seleccionadas).

☑ El mensaje 3 tiene informacion de más ya que la posición de destino (x,y) esta repetida N veces, una para cada pieza seleccionada.

Enviar tarea

Información

Autor (res)	Martin Di Paola
Fecha de entrega	10/09/2024 18:00:00
Estado	Succeeded
Calificación	100%
Promedio ponderado	1.0
Número de intentos	27
Tiempo límite de envío	Sin límite de envío

Enviado como

Santiago Jorda - 102924

Grupo: Default classroom

Para evaluación

Tu mejor envío es

06/09/2024 14:48:45 - 100.0%

Bitácora de envíos

06/09/2024 14:48:45 - 100.0%