Cliente - Servidor Multithreading

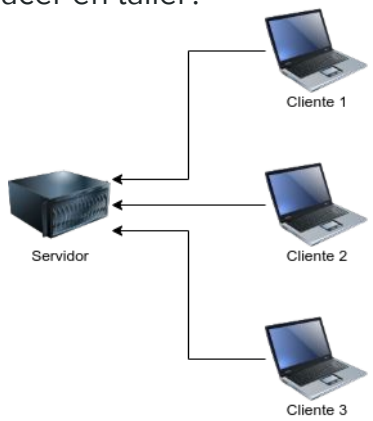
Taller de Programación I - FIUBA

¿De qué va esta parte de la clase?

- No hay teoría
- Es una ayuda, un libro de cocina para implementar el servidor multicliente

Ya no Spoiler: ¿Qué vamos a hacer en taller?

- Más de un Cliente
- Un Server



¿Qué cambios hacen falta? Cliente

- El cliente se conecta usando un socket (con un connect)
- Empieza a hablar, sin importar cuántos clientes haya
- El cliente no necesita cambios

En la clase de Multithreading dimos un Spoiler, sobre que íbamos a manejar más de un cliente con el mismo server, y dijimos que no nos importaba por el momento. Pues ahora sí!

Con lo que vimos hasta ahora, cuántos sockets hay en cada proceso del diagrama?
Hay uno en cada cliente
Y 1+3 = 4 en el servidor (un pasivo, tres activos)

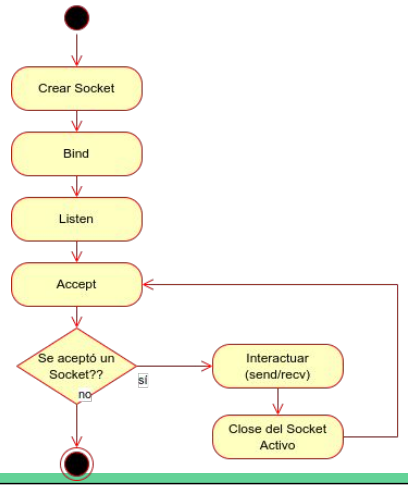
¿Qué cambios hacen falta? Servidor

- El servidor creaba un socket pasivo
- Aceptaba un cliente
- Hablaba con ese cliente
- Y ahora? Debería aceptar un segundo cliente? Tiene que esperar hasta haber terminado la conversación con el primero?

<leer diapo>
Bueno, no. Hoy estamos viendo cómo hacer varias cosas al mismo tiempo, y por ende es esperable que queramos manejar más de un cliente simultáneamente

Secuencial

- Bind/Listen: Se hacen una sola vez
- Accept: BLOQUEANTE, crea los sockets activos
- Una vez que no hay más clientes, terminamos (cómo hacemos para que accept termine sin un cliente entrante?)
- Si se aceptó un cliente, lo atendemos



<leer el diagrama de la diapo>

¿Qué parte habría que hacer “en paralelo”?

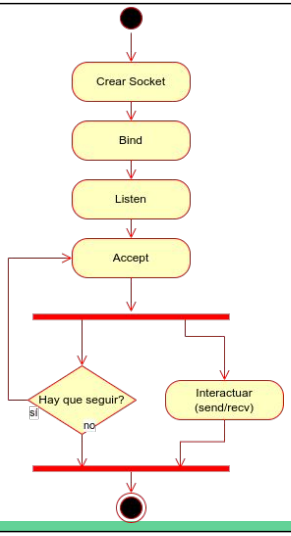
La parte de interactuar sería deseable hacerla aparte, así mientras atendemos un cliente podemos seguir aceptando otros!

Otro punto a tener en cuenta es que accept “sale” cuando hay un socket que se aceptó (devolviendo el fd), o bien cuando hay un error. Sino no termina! Ojo!

Atender en paralelo

Ahora cuando aceptamos, mandamos a atender a ese cliente, y ni bien lanzamos el thread, volvemos a aceptar. Entonces en alguna parte del código aparecerá un bucle como este:

```
1|  while (hayQueSeguir) {
2|    Socket aceptado = escuchador.accept();
3|    lanzarHiloManejador(aceptado);
4|  }
5|  esperarHilosManejadores();
```



<leer diapo>

¿Qué se desprende de esta primera versión del código?

1. Para esperar a todos los manejadores, habría que guardarlos en algún lado cuando los lanzamos. La clase que tiene ese código es la dueña (owner) de los manejadores. Este problema es trivial, simplemente agregamos un contenedor a la clase que tenga este código
2. El socket aceptado deja de pertenecer al scope del while rápidamente, para pasar a ser owneado por el manejador (move semantics). Esto ya está resuelto porque ya hicimos el Socket “movible”
3. Necesitamos que el flag hayQueSeguir cambie de alguna manera
4. Si los hilos manejadores van terminando, se van acumulando hasta que hayQueSeguir cambie (horrible, sería un leak?)

Limpiar manejadores finalizados

```
 1| void limpiarManejadoresFinalizados() {
 2|   manejadores.erase(std::remove_if(manejadores.begin(),
 3|                                    manejadores.end(),
 4|                                    esperarSiHaFinalizado),
 5|                     manejadores.end());
 6| }
 7|
 8| void atenderClientes() {
 9|   while (hayQueSeguir) {
10|     Socket aceptado = escuchador.accept();
11|     lanzarHiloManejador(aceptado);
12|     limpiarManejadoresFinalizados();
13|   }
14|   esperarHilosManejadores();
15| }
16|
```

<leer diapo>

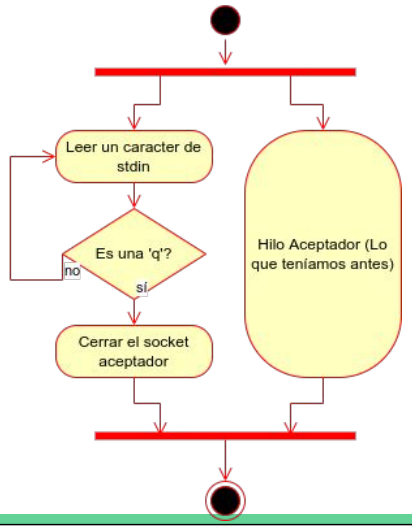
La función “esperarSiHaFinalizado” recibirá un hilo y va a preguntar si el hilo terminó. En tal caso va a hacer un join sobre ese hilo y retornará true, y por ende el objeto será removido del contenedor

Por supuesto, la función “esperarHilosManejadores” va a hacer lo mismo pero sin preguntar si los hilos terminaron

Aclaración: std::thread no tiene un método para preguntar si finalizó (tiene un método joinable, pero pregunta otras cosas, no usen eso). Tenemos que hacer nuestra propia clase, y agregarle un método con esa semántica

Cerrar el socket aceptador

- Dijimos que el accept iba a salir solamente al aceptar o al detectar un error
- Podemos forzar ese error cerrando el socket
- ¿Cuándo? Cuando el usuario lo pida. En este ejemplo y en el TP, la señal será una ‘q’ ingresada por stdin



<leer diapo>

¿Quién es el dueño del socket aceptador?

Una solución común a esto es que haya una clase Servidor, que tiene un Socket aceptador.

Entonces ese Servidor puede ser instanciado en el main() del servidor (suena bien?)

El Servidor puede tener un método para iniciarApagado(), que ponga el flag hayQueSeguir en false, y cierre el socket. Obvio, este diseño supone que el servidor es el dueño de ese flag también. “cierre el socket” en este caso == shutdown & close

Notar, sobre ese flag, que está siendo accedido tanto por el hilo aceptador como por el hilo mian, entonces debe estar protegido. Para tipos nativos, existe std::atomic. Recomendamos usar simplemente std::atomic para este booleano.

Recap: ¿Cuántos hilos hay?

El servidor que planteamos tiene estos hilos:

1. El hilo main() lanza el hilo aceptador y espera la ‘q’
2. El hilo aceptador espera conexiones, y lanza un hilo nuevo cada vez que hay una (también limpia los threads que lanzó)
3. Hay un hilo por cada cliente aceptado, que recibe requisitos y los responde

<leer diapo>
Y qué están haciendo la mayoría del tiempo??

¿Y qué están haciendo?

El servidor que planteamos tiene estos hilos:

1. El hilo main() lanza el hilo aceptador y espera la ‘q’
   Lee de entrada estándar. BLOQUEANTE
2. El hilo aceptador espera conexiones, y lanza un hilo nuevo cada vez que hay una (también limpia los threads que lanzó)
3. Hay un hilo por cada cliente aceptado, que recibe requisitos y los responde

El hilo main pasa su vida leyendo de entrada estándar

¿Y qué están haciendo?

El servidor que planteamos tiene estos hilos:

1. El hilo main() lanza el hilo aceptador y espera la ‘q’
   Lee de entrada estándar. BLOQUEANTE
2. El hilo aceptador espera conexiones, y lanza un hilo nuevo cada vez que hay una (también limpia los threads que lanzó)
   Acepta conexiones. BLOQUEANTE
3. Hay un hilo por cada cliente aceptado, que recibe requisitos y los responde

El hilo aceptador se pasa la vida en el accept, que es bloqueante!

¿Y qué están haciendo?

El servidor que planteamos tiene estos hilos:

1. El hilo main() lanza el hilo aceptador y espera la ‘q’
   Lee de entrada estándar. BLOQUEANTE
2. El hilo aceptador espera conexiones, y lanza un hilo nuevo cada vez que hay una (también limpia los threads que lanzó)
   Acepta conexiones. BLOQUEANTE
3. Hay un hilo por cada cliente aceptado, que recibe requisitos y los responde
   Los hilos manejadores de clientes se la pasan en el recv. BLOQUEANTE

El hilo manejador se pasa la vida en el read, que es bloqueante!

Esto no es casualidad. El diseño más sencillo para este tipo de aplicaciones es lanzar un hilo por cada cosa bloqueante que necesitamos, y por eso les proponemos este diseño

Dato: los sockets pueden ser no bloqueantes también, y existen otras syscalls como select o epoll. En esta materia **no está permitido** usar esas cosas, porque son mucho más complicadas.

Vamos a ver alguna excepción a esto de “un thread por cada cosa bloqueante” cuando veamos game loops, pero en general usen ese lineamiento que funciona bastante bien

Manejadores de Clientes y el Modelo

- Vamos a tener los Manejadores de Clientes, cada uno en su hilo
- Y aparte, ciertos objetos compartidos entre esos hilos. Recordar programar los respectivos Monitores (las Critical Sections son los métodos, no olvidar)
- Cada uno de estos manejadores van a tener una referencia al Monitor del modelo, y de esa manera tenemos todo desacoplado:
  - El ManejadorDeCliente hace la comunicación y le pega al Monitor
  - El Modelo tiene la lógica de negocio sin preocuparse por la concurrencia
  - El Monitor tiene el manejo de concurrencia (critical sections)