Descripción de las estructuras utilizadas

Client:

Se leen los comandos de la entrada estándar y se encarga su procesamiento a la clase correspondiente.

Processor:

Clase base de la cual heredan los 2 procesadores del programa:

- ***NewProcessor:*** Por medio de la clase ApplicantRequest envía al Server una solicitud para crear un nuevo certificado. Recibe notificación del server indicando si el certificado se creo correctamente o, en caso de error, qué tipo de error ocurrió. Verifica que los datos encriptados por el server sean correctos y notifica el resultado de su ejecución por salida estándar.
- ***RevokeProcessor:*** Solicita al server revocar un certificado. Recibe notificación del server indicando si el certificado se revocó correctamente o, en caso de error, qué tipo de error ocurrió.

ApplicantRequest:

Recibe los dos archivos necesarios para solicitar un nuevo certificado y los envía al server.

CertificateInfoParser:

Recibe el nombre de un archivo que contiene un certificado válido y los guarda en los string pasados por parámetro.

InvalidRequest:

Excepción que es lanzada cuando el server notifica un error en la creación o revocación de un certificado.

Time:

Se crea automáticamente con la fecha actual. Se le puede sumar 30 días Se puede pasar a string con el formato MMM DD HH:mm:SS YYYY.

Server:

Crea el socket. Procesa tanto el archivo con los certificados existentes como el archivo que contiene las claves del server. Lanza un thread encargado de aceptar y procesar los clientes. Deja correr este thread hasta que recibe una 'q' por entrada estándar. Actualiza el archivo con los certificados existentes Termina su ejecución.

Acceptor:

Mientras el server no mande la señal de 'stop' acepta a un cliente, libera un thread (ClientProcessor) que lo procese, "joinea" los clientes que ya terminaron su ejecución y vuelve a iterar.

ClientProcessor:

Clase que hereda de Thread y, a su vez, es clase base de la cual heredan los 2 procesadores del programa:

- ***New:*** Si el certificado existe envia el error al cliente y lanza un error de tipo ExistingCertificate. En caso contrario almacena el certificado en memoria, calcula su encriptación y la envía al cliente. Espera la respuesta del cliente, si la encriptación no es correcta borra el certificado. Luego termina su ejecución y es "joineado" por el Acceptor.
- ***Revoke:*** Si el certificado no existe envia el error al cliente y lanza un error de tipo InexistingCertificate. Verifica si la encriptación del cliente es correcta y notifica. Si lo es borra el certificado de memoria sino simplemente termina su ejecución

Index:

- Lee el archivo con los certificados y el próximo número de serie y los almacena.
- Borra y guarda certificados en memoria.
- Vuelve a crear el archivo con los certificados existentes (actualizado).

Common:

Certificate:

- Lee un archivo con un certificado válido y almacena sus datos.
- Manda y recibe los datos del certificado.
- Crea un certificado a partir de los datos recibidos.

Protocol:

Encapsula al socket. Conoce la forma de comunicar las cadenas y los números que se envían en este proyecto.

Hash:

Recibe un string y lo hashea a un número de 4 bytes.

Rsa:

Recibe un número de 4 bytes y lo encripta en un nuevo número de 4 bytes.

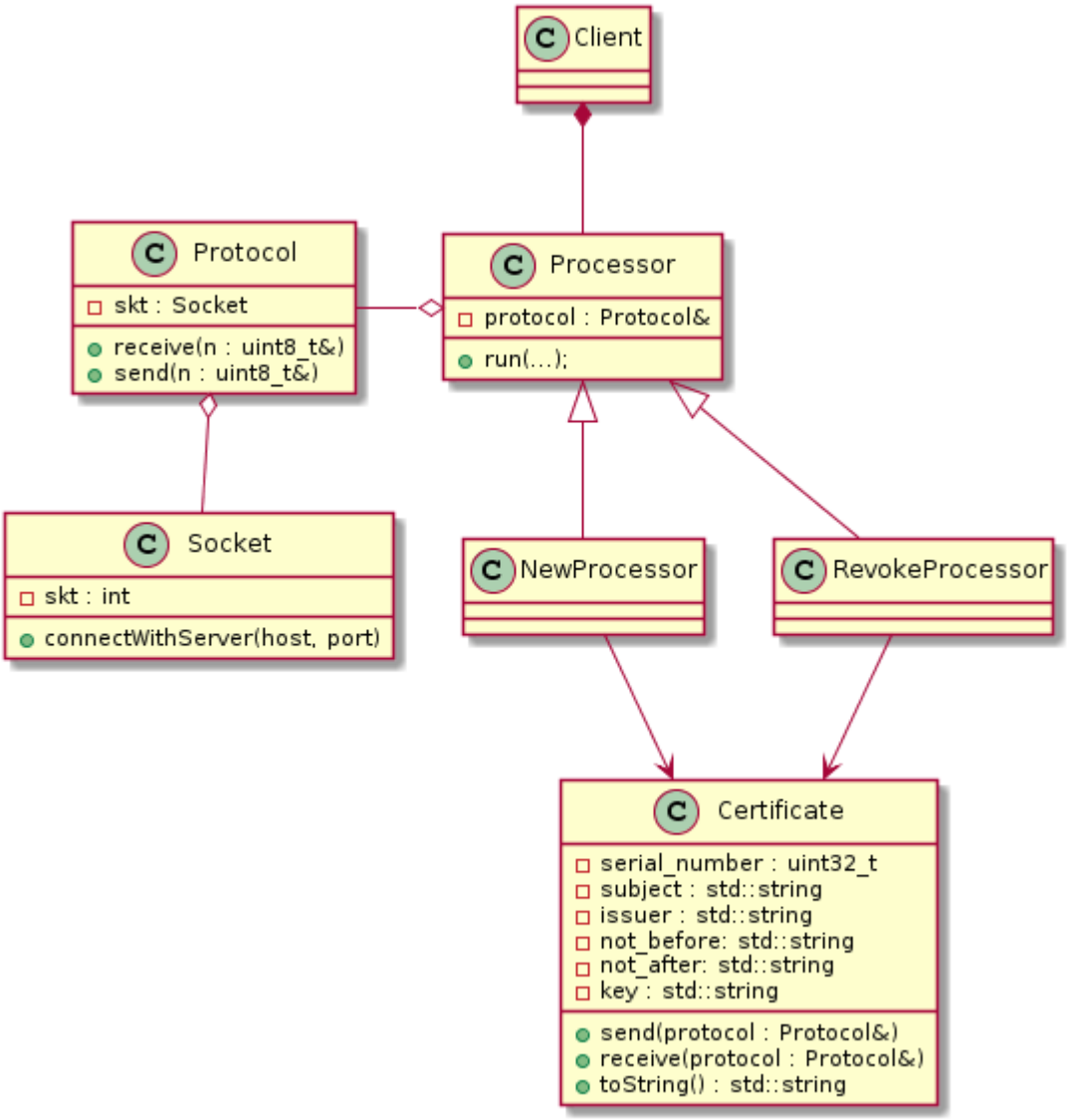
Key:

- Recibe el nombre de un archivo de tipo `<exp_publico> <exp_privado> <modulo>` y almacena los valores.
- Manda y recibe sus datos a través de un protocolo pasado por parámetro.

Esquema del diseño

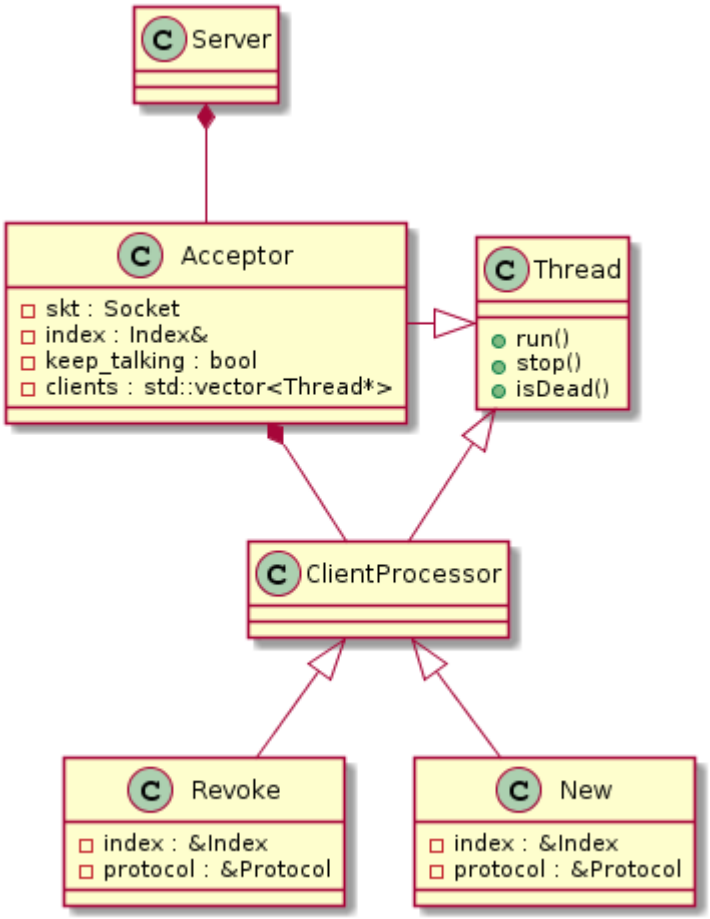
Diagramas de clases

Cliente:



Server:

Principales clases:



En función de revoke y new:

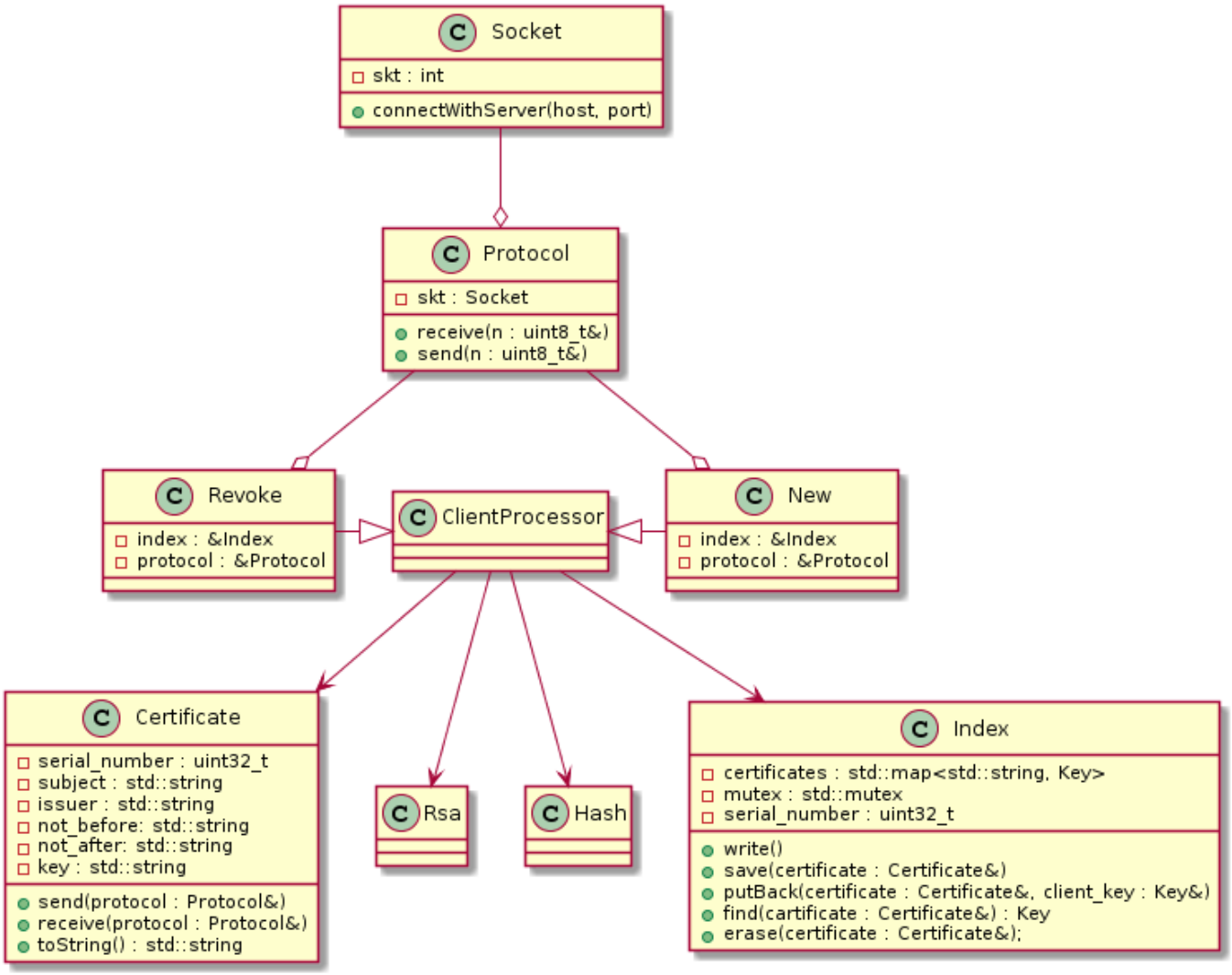
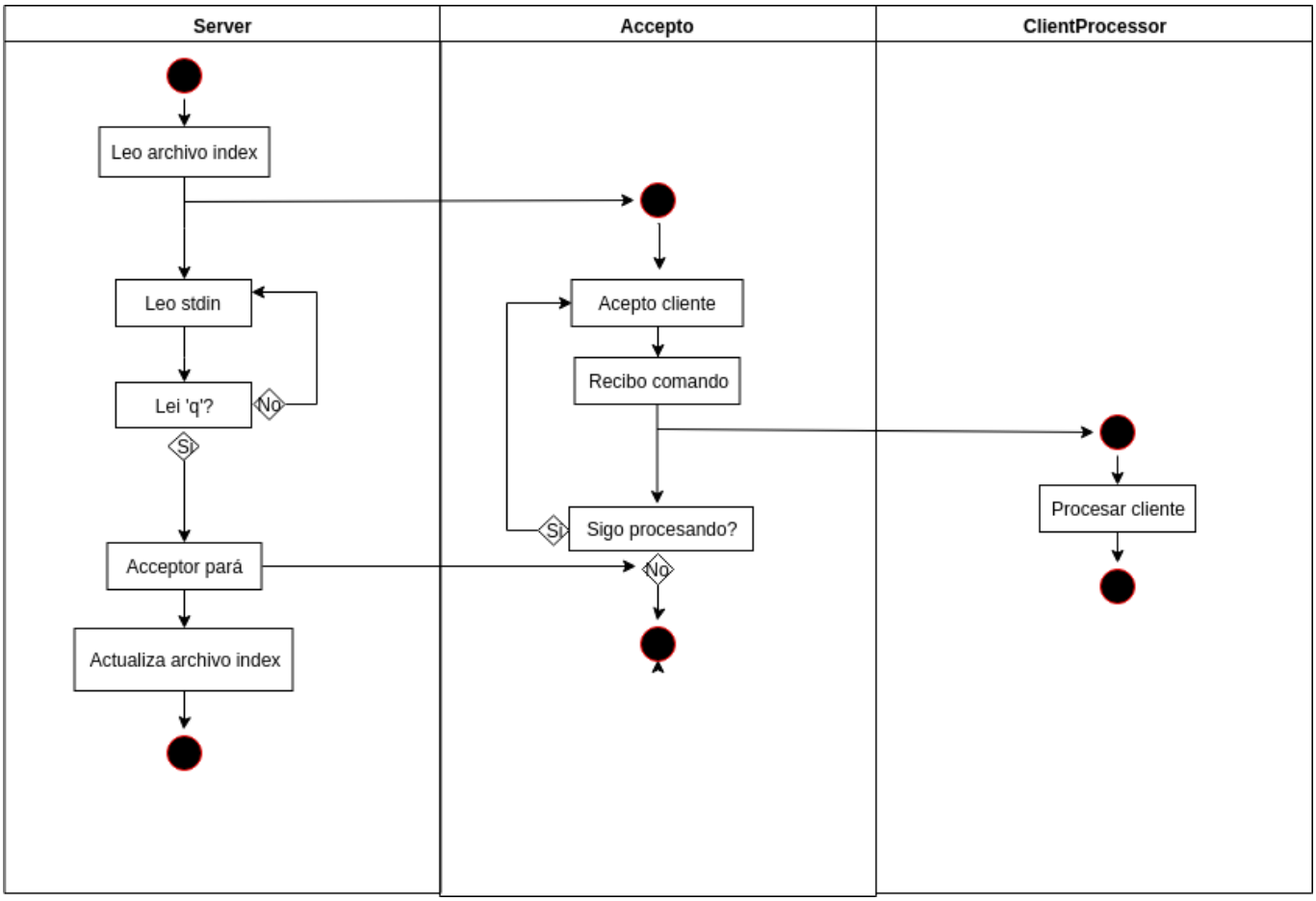


Diagrama de actividad

Procesar clientes:



Correcciones

Obligatorias para aprobar

1- Falta un informe con diagramas de clase y secuencia explicando el trabajo realizado. Acá está 😃

3- Hay copia de objetos a lo largo de todo el código. ¿Era necesario copiarlos? Revisá los strings que pases por parámetro, los sockets, etc. Fueron retirados todos los parámetros que eran pasados por copia (si es que no se me escapó ninguno).

4- El programa no te funciona por un conjunto de situaciones (me costó bastante darme cuenta la verdad):

- ***Estás copiando los sockets en el server cuando appeas tu socket en tu protocolo. Cuando el constructor e MySocket termina, esa copia se destruye, dejandote inválido tus sockets recientemente creados (porque el se cierra el file descriptor). Por estas cosas se insiste 1 mil veces evitar copias a menos que realmente se quiera hacerlo. Poné el atributo delete en el constructor de copia y asignación por copia de todas tus clases que no quieras se copien. De esta forma el compilador te va ayudar muchisimo.*** La clase Protocolo (ex MySocket) recibe el socket como referencia pero lo almacena moviéndolo a su atributo. La misma lógica se utiliza al pasar

el protocolo como parámetro (el movimiento de un protocolo es básicamente el movimiento del socket que tiene como atributo)

- ***El otro error está en que copypasteaste todo el código de socket sin considerar algún cambio. En el tp1 el protocolo constaba de enviar un stream de datos hasta que se cierre el canal del socket. En ese caso, en las funciones send y recv del socket no se consideraba como error que llegue un status igual a cero. En este protocolo, los mensajes son de ancho fijo y/o con un header que indica el tamaño (para los strings). No es esperable que te cierren el socket, por lo que deberías lanzar una excepción notificando que el canal de cerró inesperadamente. Lo que te está sucediendo es que como el canal está cerrado, recibís siempre cero en ambos lados. Cuando ejecutas el comando new en el cliente, el recibir cero implica un error, y en el comando revoke cierra "ordenadamente" ante el "éxito" de la revocación.*** La clase Socket fue refactorizar furiosamente antes de encontrar el bug (antes de recibir esta correcciones) porque no estaba muy prolija. Además se agregaron las excepciones correspondientes.

12- Tanto en la función main como toda función principal que ejecutes en un hilo deberían tener un bloque try...catch que capture una referencia a std::exception y otro catch más que atrape las elipsis. Revisá el apunte de clase. Se agregan `try catch` para atrapar excepciones propias del tp (hashes que no coinciden, certificados que ya existen, certificados que no existen) Se hace un `try catch` genérico que no deje pasar ninguna excepción (error en un archivo, en un sockets y demás).

13- Es peligroso como estás eliminando los clientes terminados. Si eliminaste el nodo donde apunta it del vector, ¿A donde va a apuntar it++ ? Podés tener corrupción de memoria así. El método erase te devuelve un iterator que apunta al nodo posterior del eliminado. Tenés que pisar tu variable iterator para que no tengas corrupción de memoria. Notar que si la función te devuelve el nodo siguiente al eliminado en esa iteración no deberías incrementar el iterator (porque sinó te habría un cliente que no estarías eliminando de estar finalizado). antes:

```
for (; it != this->clients.end(); ++it) {
    Thread* client = *it;
   if (client->isDead()) {
       client->join();
       delete client;
       this->clients.erase(it);
   }
}
```

ahora:

```
while (it != this->clients.end()) {
   Thread* client = *it;
   if (client->isDead()) {
      client->join();
       delete client;
       this->clients.erase(it);
   } else {
       ++it;
```

```
    }
}
```

16- Tenés race conditions. Están mal definidas tus funciones atómicas. ¿Las funciones atómicas son agregar, chequear, borrar , o deberían ser agregarSiNoExiste, borrarSiExiste ?. Revisar apuntes de clase y evaluar si existen otras critical sections. Pensar que pasaría (teniendo tu código así como está), si dos hilos quieren crear un certificado para el mismo subject. ¿Qué pasaría si un hilo quiere crear un certificado y al mismo tiempo otro cliente quiere revocar un certificado para ese subject? Se modifican los métodos guardar y borrar para emitir una excepción si ya existe el certificado (guardar) o no existe (borrar). Ya no se chequea por fuera del index si existe o no el certificado.

Suman puntos

2. Es un nombre poco claro el "MySocket". Si implementa el protocolo capaz sería más explicativo que se llame "Protocol". Cambiado.

5. Los mensajes de error planteados en el enunciado deberían ser una excepción (La razón de que deban printearse en el estándar de salida es porque el sercom no chequea el estándar de error de los clientes. Es un feature que hay que corregir por parte nuestra y por el que tuve que pelearme este cuatrimestre al confeccionar el tp). Recomiendo hacer una clase que herede de std::exception que printee estos mensajes de error en el estándar de salida, y otra clase genérica (como la vista en clase) que printee en el estándar de error todos los mensajes de error propios. Podrías hacer la clase genérica y luego otra que herede de esta clase, llamada por ejemplo TpException (tuve todo un brainstorm con el nombre). En cliente.cpp se "catchea" los errores propios del tp y simplemente imprime su mensaje.

8. Evitá usar la función atoi . Es una función que no chequea errores y es insegura. En C te diría que uses stol, en C++ recomiendo usar el operator>> Fueron reemplazados por el uso de la clase `istringstream` y su operador >>. En mi IDE ya no aparecen más resultados a la búsqueda `atoi`.

9. Evitá que los constructores tengan lógica extra que no sea la de inicializar atributos. En `CertificateInfoParcer` se agrega el método `run()` al cual se le delega la responsabilidad que antes tenía el constructor.

11. Revisar la biblioteca iomanip para ver como manipular la salida de un número o un time_t . Cambios visibles en clase Time (archivo cliente_time.cpp)

15. No tiene ningún sentido allocar el protocolo en el heap. Como se explicó en el punto 4: El protocolo se pasa como referencia a los procesadores del server que lo mueven para almacenarlo como atributo. Por ello, ya no es necesario que el protocolo esté allocado para no morir con el scope.

Porque si:

(cambios que recuerdo haber hecho entre entrega anterior y esta)

- Protocol ahora tiene dos métodos nada mas, pero sobreescritos. Antes:

```

```

```
sendNumber(uint_8)
send(string)
receiveNumber(uint_8)
receive(string)
```

Ahora:

```
send(uint_8)
send(string)
receive(uint_8)
receive(string)
```

- Se agrega herencia tanto en los procesadores del cliente como en los del server.