Trabajo Práctico final 2024/2C

Objetivo

El objetivo del presente trabajo es comprender el desarrollo de protocolos de aplicación para TCP/IP, orientados y no orientados a conexión.
El mismo deberá ser realizado en grupos de hasta 3 alumnos. Los grupos deben anotarse creando una issue en https://github.com/marcelogarberoglio/PDC_2024_02/issues, indicando legajo y apellido de los miembros del grupo

Especificaciones

En el servidor deberá configurarse una carpeta donde se alojen archivos de texto, y el protocolo deberá tener los comandos necesarios para recuperar el texto de un archivo, sólo para usuarios autorizados.
Los protocolos a desarrollar deberán ser **de texto**, no binarios.

El protocolo debe cumplir como mínimo con los siguientes requisitos:

1) El servidor debe aceptar una conexión usando TCP en la cual
    a) Recibe líneas de texto US-ASCII finalizadas con \r\n
    b) La primera línea de texto debe ser el nombre del usuario, un espacio y la clave del mismo. Las credenciales estarán almacenadas en un archivo de texto plano, en donde además se indicará si es un usuario administrador o no. Si la autenticación falla se cierra la conexión
    c) Cada línea recibida debería tener una longitud máxima de 100 caracteres (sin contar los \r\n al final de la línea)
    d) Si una línea supera la longitud, los caracteres excedentes se ignoran
    e) Si la línea contiene algún carácter no US-ASCII se ignora desde ese carácter hasta el final de la línea
    f) Registrar en un archivo de log todas las actividades del servidor, como conexiones, comandos recibidos y errores.
    g) Cada línea será uno de los siguientes comandos
        i) **`AUTH username password\r\n`** que permita autenticar a los clientes antes de realizar cualquier operación. Sólo un usuario administrador debería enviar este comando
        ii) **`LIST FILES\r\n:`** el servidor responderá con una lista de nombres de archivos habilitados para su recuperación. Cada grupo deberá definir el formato de la respuesta.
        iii) **`ECHO `*`texto`*`\r\n`**: el servidor le enviará como respuesta al cliente el texto recibido, finalizado con \r\n
        iv) **`GET `*`fileName`*`\r\n`**: si el archivo existe, el servidor responde primero con una línea indicando OK, un espacio y la cantidad de bytes

del archivo (por ejemplo "OK 132456\r\n") y luego el contenido del archivo. Si el archivo no existe debe indicar un mensaje de error.

Por defecto el protocolo es case-insensitive (*) para los comandos, los siguientes mensajes son equivalentes

- GET errores.txt
- get errores.txt
- Get errores.txt

(*) en un archivo de propiedades se puede cambiar este valor, y que por defecto sea case sensitive.

El protocolo no contempla que se puedan subir archivos, los mismos deberán alojarse en la carpeta correspondiente utilizando algún otro servicio

2) A su vez el servidor puede recibir los siguientes datagramas (vía UDP)
    a) **`SET case ON`**: a partir de ese momento, el servidor será case-sensitive para los comandos de ese usuario. Este valor debe perdurar aún si se reinicia el servicio
    b) **`SET case OFF`**: a partir de ese momento, el servidor no será case-sensitive para los comandos de ese usuario. Este valor debe perdurar aún si se reinicia el servicio
    c) **`STATS`**: le devuelve al cliente un datagrama con las siguientes estadísticas en modo texto, cada una separada por \r\n,
        i) cantidad de conexiones realizadas desde que inició la ejecución
        ii) cantidad de líneas incorrectas recibidas
        iii) cantidad de líneas correctas recibidas
        iv) cantidad de datagramas incorrectos recibidos (comandos inválidos o inexistentes)
        v) cantidad de archivos descargados
        vi) cantidad de archivos subidos (sólo para los que entreguen en segunda fecha)

3) En ambos casos el puerto por defecto a usar por el servidor será el 9999, pero podrá setearse en el archivo de configuración en qué puerto escuchar

Los grupos que entreguen en segunda fecha deberán además implementar los siguientes ítems

4) El servidor deberá cerrar la conexión de los clientes que hayan estado ociosos por más de N segundos, donde el valor de N se obtiene del archivo de propiedades
5) Soportar el comando **`GET base64 `*`fileName`*`\r\n`**: si el archivo existe, el servidor responde primero con una línea indicando OK, un espacio y la cantidad de bytes **del archivo** (por ejemplo "OK 132456\r\n") y luego el contenido del archivo codificado en base 64. Si el archivo no existe debe indicar un mensaje de error.

A tener en cuenta:

- Se desea que el servidor utilice la menor cantidad de memoria posible. Y no debe tener límites prefijados de conexiones simultáneas.
- Antes de desarrollar el TPE, aconsejamos ver los ejercicios de la guía "Programación con sockets", especialmente los que plantean algunas deficiencias o limitaciones de los códigos vistos en clase.

Uso de Git

Es obligatorio el uso de un repositorio Git para la resolución de este final. Siguiendo los pasos indicados en la clase extra de Git deberán crear un repositorio en GitHub donde todos los integrantes del grupo colaboren con las modificaciones del código provisto. **No se aceptarán entregas que utilicen un repositorio *git* con un único *commit* que consista en la totalidad del código a entregar**.

Los distintos *commits* deben permitir ver la evolución del trabajo, tanto grupal como **individual,** por lo que el nombre de cada usuario deberá tener relación con el nombre de cada miembro del grupo.

Muy importante: los repositorios creados deben ser privados.

Material a entregar

Un archivo compactado conteniendo como mínimo los siguientes archivos:

- Archivos fuentes y de encabezado de la aplicación servidor.
- Archivo de texto README con la explicación de cómo generar los ejecutables y de cómo ejecutarlos.
- Uno o más programas cliente de prueba. Dichos programas deben contemplar tanto los casos exitosos como casos de error (enviar comandos incorrectos, solicitar archivos no existentes, pruebas de stress, etc.)
- `makefile` (se deben usar gcc y los flags `-Wall` y `-fsanitize=address`).
- Carpeta .git

Para facilitar el testeo, se deben crear al menos tres configuraciones diferentes del servidor usando Docker Compose

Por ejemplo:

1. **Servidor con case-sensitive activado y sólo un usuario admin.**
2. **Servidor con case-insensitive y usuarios ya creados.**
3. **etc**

Ejemplo parcial de archivo compose.yaml

```
services:
  server1:
    image: _____
    container_name: server1
    ports:
      - "9999:9999"
    volumes:
      - ./server1/files:/app/files # archivos a compartir
```

```
      - ./server1/config:/app/config # archivo de configuración

  server2:
    image: _____
    container_name: server2
    ports:
      - "10000:9999"
    volumes:
      - ./server2/files:/app/files # archivos a compartir
      - ./server2/config:/app/config # archivo de configuración

  server3:
```

La fecha límite para la entrega para los que rindan en primera fecha es el 10/12 a las 23:59 hs, y el 16/12 a las 23:59 para los que rindan en segunda fecha

Dudas sobre el TPE

Si bien el enunciado contempla la funcionalidad completa a desarrollar es normal que surjan dudas acerca de cómo interpretar ciertos casos. O que una consigna genere más de una posible solución, por lo que es importante que analicen bien el enunciado, y ante cualquier duda pregunten. Sólo se contestarán dudas sobre el enunciado. Las mismas deben volcarse en Github: https://github.com/marcelogarberoglio/PDC_2024_02/issues

En caso de realizar alguna aclaración o consideración sobre el enunciado, la misma deberá ser tenida en cuenta por todos los grupos, no solo para el grupo que haya hecho la pregunta.