Índice

1. Introducción

Un **honeypot** es una aplicación que simula ser un servidor de otra aplicación mayor, para que cuando un usuario malicioso se conecte, ataque este servidor falso, permitiendo tomar registro de las técnicas de ataque que se utilizan en la red.
En este trabajo práctico prototiparemos un honeypot de un servidor **FTP**.

1.1. Motivaciones

La motivaciones principales y la búsqueda del **Trabajo Práctico 3** son:

- El diseño y la construcción de sistemas con acceso distribuido.
- Encapsulamiento de Threads y Sockets en Clases.
- La protección de los recursos compartidos entre clientes clientes en una arquitectura **cliente-servidor**.
- El uso criterioso y medido de excepciones bajo un esquema de diseño **RAII**.

1.2. Protocolo FTP

Protocolo FTP El protocolo FTP es un protocolo de texto formado por mensajes delimitados por un salto de línea.
Los comandos son palabras en su mayoría de 4 letras, seguidos de un espacio que los separa de sus argumentos. El servidor responde con un código numérico y un texto descriptivo.

2. Cuestiones de implementación

2.1. Descripción de las Clases implementadas

A continuación, mencionaremos y explicaremos brevemente las clases involucradas en la implementación del presente trabajo.

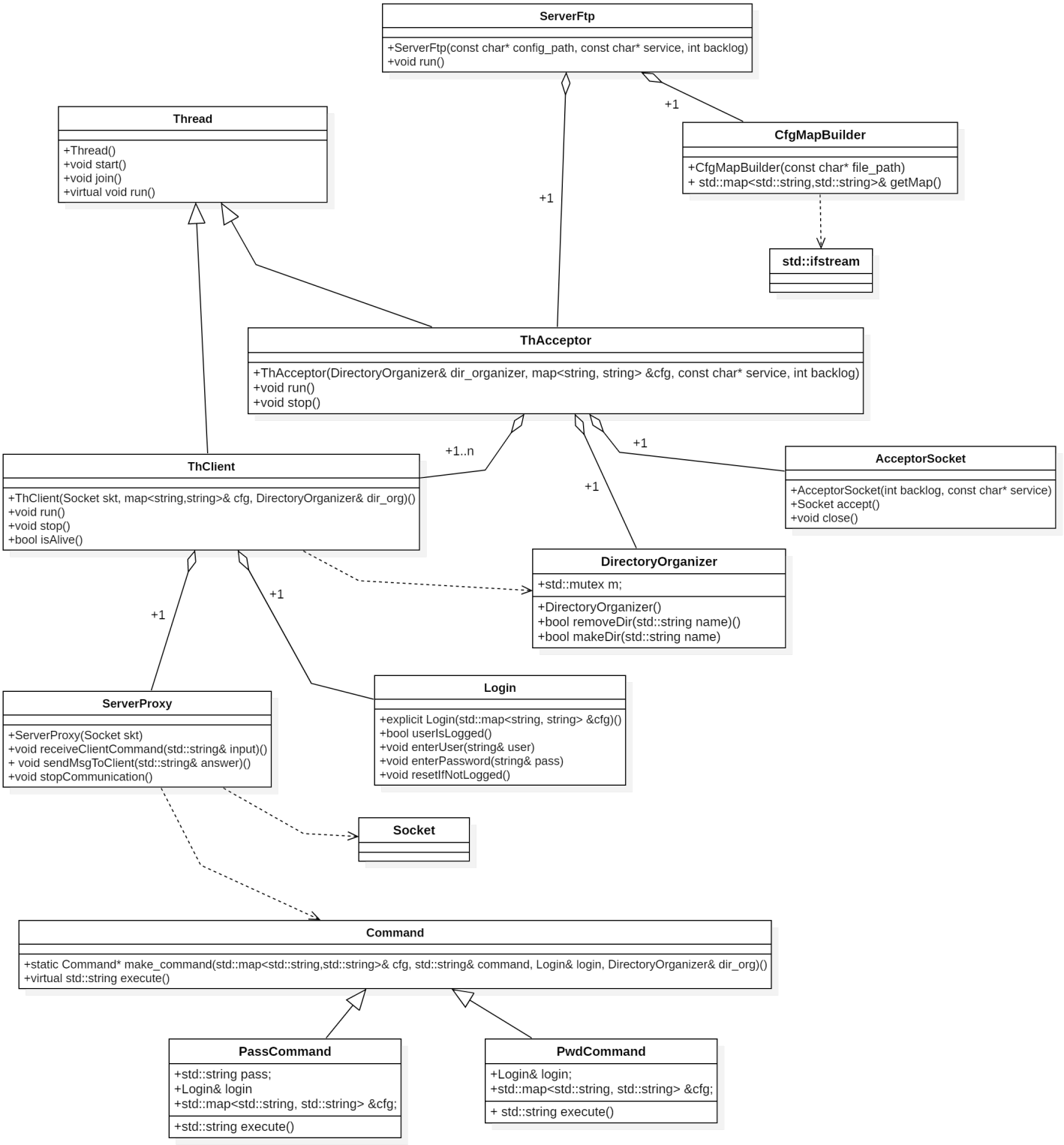
- **Thread** Es una clase abstracta que contiene en su interfaz a los métodos propios de un *thread* (tales como *run* y *join*).
- **Socket** Modela el comportamiento propio de un *socket* el cual puede, únicamente, enviar y recibir mensajes mediante el protocolo TCP. Puede construirse proporcionando **host** y un **puerto**, o un file descriptor válido para establecer la comunicación.
- **AcceptorSocket** Tiene la finalidad de realizar la acción *accept* de un socket para establecer la conexión entre el servidor y un cliente en lista de espera.
- **ServerFtp** Es la interfaz del servidor. Sus acciones principales son: inicializar los objetos propios del servidor, lanzar el hilo aceptador y frenarlo (liberando adecuadamente los recursos) cuando se deba finalizar la ejecución del servidor.
- **ServerProxy** Se encarga de enviar y recibir mensajes desde el servidor utilizando el protocolo **FTP**.
- **ThAcceptor** Es un *thread* que cumple la finalidad de aceptar nuevos clientes. Este puede finalizarse mediante el método *stop*, que finaliza los *threads* pertenecientes a las instancias de **ThClient**, liberando adecuadamente sus recursos.
- **DirectoryOrganizer** Representa, como su nombre lo indica, a un organizador de directorios. Tiene la tarea de crear directorios nuevos, borrarlos y devolver una lista de los directorios presentes en su interior ordenados alfabéticamente.

- **CfgMapBuilder** Se encarga de crear un *map* que tiene como claves "modificadores"de respuesta del servidor ante una determinada petición del cliente, y como valor una respuesta a ese modificador.
- **ThClient** Es un *thread* que se encarga de procesar los pedidos realizados por el **cliente i** y responderle. Utiliza el *map* creado por el **CfgMapBuilder** para obtener una respuesta ante un pedido, y envía la misma haciendo uso de una instancia de **ServerProxy**.
- **Login** "Vive"dentro de cada instancia de **ThClient** y se encarga de chequear si el usuario está loggeado en el servidor para aceptar o rechazar la acción que quiere realizar.
- **Command** Es la clase madre de todos los comandos presentes en el servidor. Se hace uso del patrón **Factory Method** para ejecutar comandos a partir del uso de esta clase.
  Clases "hijas"que surgen a partir de la misma:
  - UserCommand
  - PassCommand
  - SystCommand
  - QuitCommand
  - ListCommand
  - PwdCommand
  - UnknownCommand
  - MkdCommand
  - RmdCommand
  - HelpCommand
- **ClientFtp** Es la interfaz del cliente. Se encarga de leer comandos por entrada estandar, enviarlo y recibir una respuesta al mismo a través del **ClientProxy**, y por último imprimir la respuesta en pantalla.
- **ClientProxy** Se encarga de enviar y recibir mensajes hacia y desde el servidor. Además, implementa un método para enviar un comando al servidor y recibir respuesta del mismo.

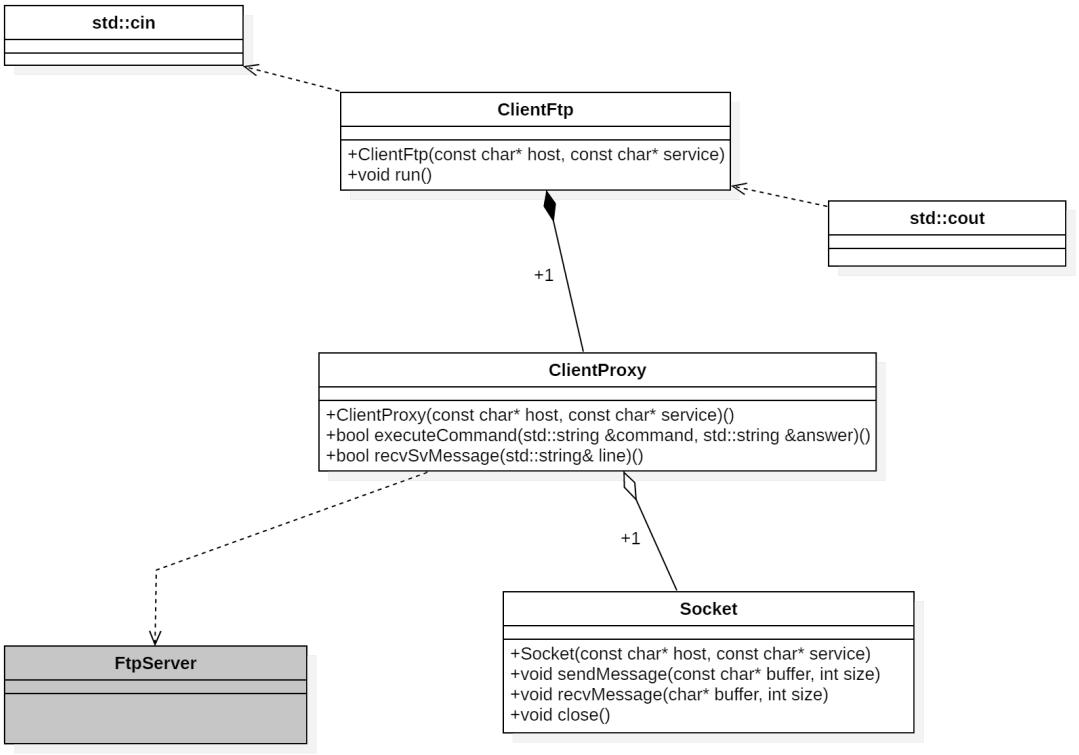
3. Diseño del código

3.1. Diagrama de Clases del subsistema servidor

- **Aclaración 1**: No se incluyen todas las clases que heredan de la clase Command ya que no aportan hacia el fin para el cual se propone en el siguiente esquema.
- **Aclaración 2**: Se describen los métodos de la Clase **Socket** en el diagrama de Clases del subsistema cliente para evitar la sobrecarga del esquema.


ServerFtp
+ServerFtp(const char* config_path, const char* service, int backlog)
+void run()
Thread
+Thread()
+void start()
+void join()
+virtual void run()
CfgMapBuilder
+CfgMapBuilder(const char* file_path)
+ std::map<std::string,std::string>& getMap()
std::ifstream
ThAcceptor
+ThAcceptor(DirectoryOrganizer& dir_organizer, map<string, string> &cfg, const char* service, int backlog)
+void run()
+void stop()
ThClient
+ThClient(Socket skt, map<string,string>& cfg, DirectoryOrganizer& dir_org)()
+void run()
+void stop()
+bool isAlive()
AcceptorSocket
+AcceptorSocket(int backlog, const char* service)
+Socket accept()
+void close()
DirectoryOrganizer
+std::mutex m;
+DirectoryOrganizer()
+bool removeDir(std::string name)()
+bool makeDir(std::string name)
ServerProxy
+ServerProxy(Socket skt)
+void receiveClientCommand(std::string& input)()
+ void sendMsgToClient(std::string& answer)()
+void stopCommunication()
Login
+explicit Login(std::map<string, string> &cfg)()
+bool userIsLogged()
+void enterUser(string& user)
+void enterPassword(string& pass)
+void resetIfNotLogged()
Socket
Command
+static Command* make_command(std::map<std::string,std::string>& cfg, std::string& command, Login& login, DirectoryOrganizer& dir_org)()
+virtual std::string execute()
PassCommand
+std::string pass;
+Login& login
+std::map<std::string, std::string> &cfg;
+std::string execute()
PwdCommand
+Login& login;
+std::map<std::string, std::string> &cfg;
+ std::string execute()
+1
+1
+1
+1..n
+1
+1
+1

3.2. Diagrama de Clases del subsistema cliente



4. Conclusión

4.1. Librerías más importantes utilizadas

1. **set** Utilizada principalmente en la Clase **Directory Organizer** para poder devolver los directorios ordenados alfabéticamente.
2. **string** Utilizada para almacenar comandos y realizar concatenaciones y extracciones de subcadenas para realizar la lógica del procesamiento de comandos.
3. **vector** Utilizada para almacenar los clientes conectados en la Clase **ThAcceptor**.
4. **algorithm** Utilizada para el parseo de comandos.
5. **mutex** Utilizada en la clase **Directory Organizer** para evitar *race conditions*.

4.2. Referencias

- `https://es.cppreference.com`
- `cplusplus.com`