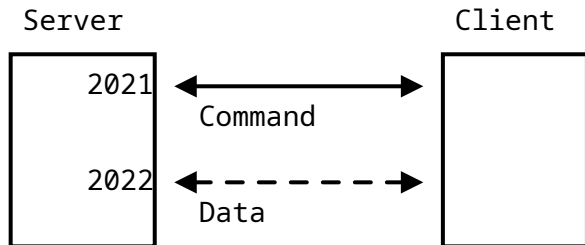


PROTOCOLO FTP

Matías Roqueta

Ingeniería en Telecomunicaciones, Instituto Balseiro

Diseñar arquitectura Cliente ↔ Servidor



- Puerto 2021: Command channel, permanece abierto mientras la sesión esté activa
- Puerto 2022: Data channel, abierto únicamente en respuesta a comandos que lo requieran

FUNCIONES SOCKET

Creación de socket para Command y Data channels

```
int createSocket(int port) {  
    int sockfd = socket(AF_INET, SOCK_STREAM, 0);  
    int reuseAddr = 1;  
    setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &reuseAddr, sizeof(reuseAddr))  
    sockaddr_in serverAddress{};  
    serverAddress.sin_family = AF_INET;  
    serverAddress.sin_port = htons(port);  
    serverAddress.sin_addr.s_addr = INADDR_ANY;  
    bind(sockfd, (struct sockaddr*)&serverAddress, sizeof(serverAddress))  
    listen(sockfd, 1)  
    return sockfd;  
}
```

SO_REUSEADDR=1 permite cerrar y volver a abrir el Data channel según sea requerido

FUNCIONES SOCKET

Command Channel:

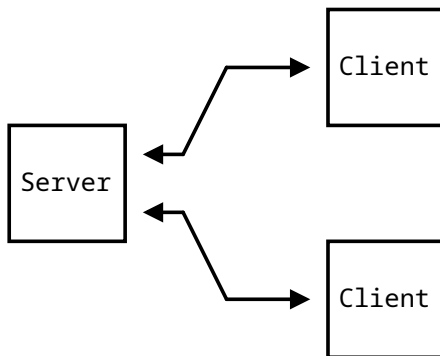
```
int acceptClientConnection(int controlSocket) {  
    sockaddr_in clientAddress{};  
    socklen_t clientAddressLength = sizeof(clientAddress);  
    int clientSocket = accept(controlSocket,  
                              (struct sockaddr*)&clientAddress, &clientAddressLength);  
    return clientSocket;  
}
```

Data Channel:

```
int establishDataConnection(int controlClientSocket, int dataSocket,  
                           std::string &dataAddress, int dataPort){  
    std::string dataMessage = "PORT "+dataAddress+": "+dataPort+"\r\n";  
    send(controlClientSocket, dataMessage.c_str(), dataMessage.size(), 0);  
    int dataClientSocket = acceptClientConnection(dataSocket);  
    return dataClientSocket;  
}
```

MÚLTIPLES CLIENTES

El servidor debe ser capaz de atender a múltiples clientes.



- La conexión Cliente \leftrightarrow Servidor debe ser no-bloqueante.
- Esto se logra con un *fork* al recibir conexión entrante.

MÚLTIPLES CLIENTES - MAIN

```
int controlSocket = createSocket(2021);
std::vector<pollfd> pollFds(1);
pollFds[0].fd = controlSocket; pollFds[0].events = POLLIN;
while (true) {
    int pollResult = poll(pollFds.data(), pollFds.size(), -1);
    if (pollFds[0].revents & POLLIN) {
        pid_t pid = fork();
        if (pid == 0) {
            int controlClientSocket = acceptClientConnection(controlSocket);
            closeSocket(controlSocket);
            handleClientConnection(controlClientSocket);
            return 0;
        }
    }
}
closeSocket(controlSocket);
```

LOOP DE RESPUESTAS

```
void handleClientConnection(int controlClientSocket) {  
    sendWelcomeMessage(controlClientSocket);  
    bool authenticated = false;  
    char commandBuffer[1024];  
    while (true) {  
        memset(commandBuffer, 0, sizeof(commandBuffer));  
        ssize_t bytesRead = recv(controlClientSocket, commandBuffer,  
                                sizeof(commandBuffer), 0);  
        std::vector<std::string> commandParts = splitCommand(commandBuffer);  
        if (!authenticated) {  
            // Rechazo de comandos excepto USER PASS o QUIT + respuesta  
        } else {  
            // Aceptación de otros comandos + respuesta  
        }  
    }  
    closeSocket(controlClientSocket);  
}
```

COMANDOS IMPLEMENTADOS

DEMOSTRACIÓN...

Comandos que usan solamente
Command channel:

- USER <user>
- PASS <pass>
- CWD <dir>
- CDUP
- MKD <dir>
- RMD <dir>
- DELE <file>
- QUIT

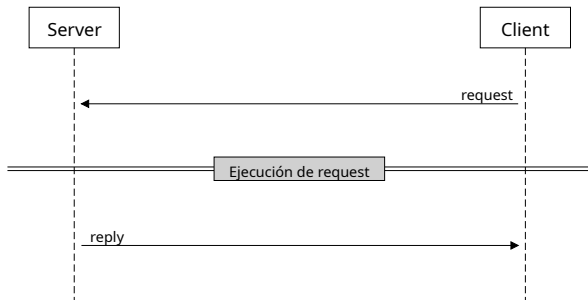
Comandos que usan ambos
Command y Data channel:

- LIST
- RETR <file>
- STOR <file>

La distinción principal en la implementación de los diferentes comandos depende de si estos usan o no el Data channel.

EJECUCIÓN GENERAL DE COMANDOS

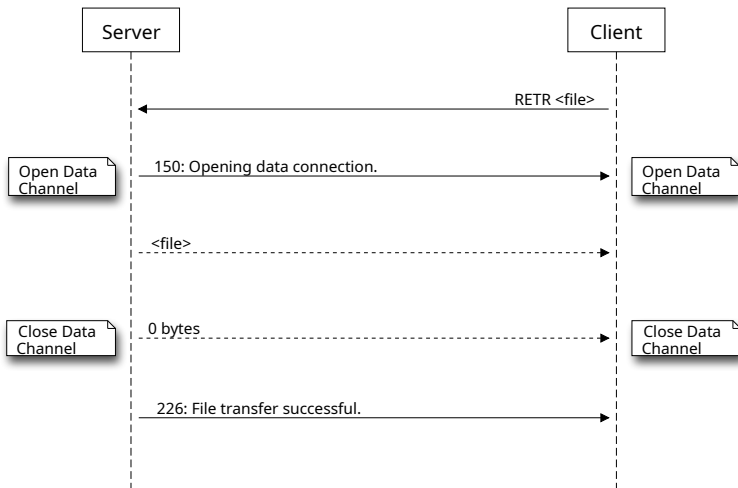
- Cliente inicia request enviando un código FTP por el Command channel



- Servidor eventualmente responde enviando un código de respuesta FTP por el command channel
- Ejecución de request puede incluir intercambio de más mensajes y/o uso del Data channel.

COMANDO RETR <FILE>

DEMOSTRACIÓN...



RETR - SERVER SIDE

```
void handleRETRCommand(int controlClientSocket, const std::string& args) {
    sendResponse(controlClientSocket, "150 Opening data connection.\r\n");
    int dataSocket = createSocket(dataPort);
    int dataClientSocket = establishDataConnection(controlClientSocket,
                                                    dataSocket, dataAddress, dataPort);

    if (sendFile(dataClientSocket, filename)) {
        std::string response = "226 File transfer successful.";
    } else {
        std::string response = "451 File transfer failed.";
    }

    closeSocket(dataClientSocket);
    closeSocket(dataSocket);
    sendResponse(controlClientSocket, response);
    return;
}
```

RETR - SERVER SIDE

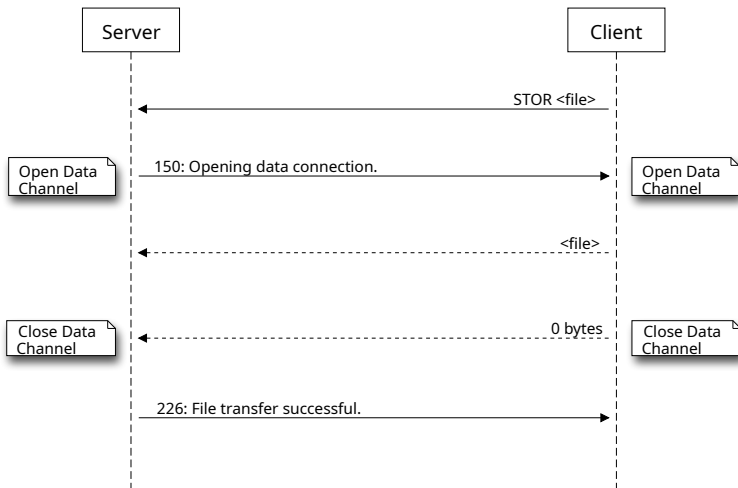
```
bool sendFile(int socket, const std::string& filename) {  
    std::ifstream file(filename, std::ios::binary);  
    if (!file) {  
        std::cerr << "Failed to open file: " << filename << std::endl;  
        return false;  
    }  
    std::vector<char> buffer(std::istreambuf_iterator<char>(file), {});  
    file.close();  
    ssize_t bytesSent = send(socket, buffer.data(), buffer.size(), 0);  
    if (bytesSent == -1) {  
        std::cerr << "Failed to send file contents." << std::endl;  
        return false;  
    }  
    return true;  
}
```

RETR - CLIENT SIDE

```
void recvFile(int controlSocket, const std::string& filename) {  
    std::string response;  
    receiveResponse(controlSocket, response);  
    if (response.substr(0, 3) != "150")  
        return;  
    int dataSocket = establishDataConnection(controlSocket);  
    const int bufferSize = 1024;  
    std::vector<char> buffer(bufferSize);  
    ssize_t bytesRead;  
    while (bytesRead = recv(dataSocket, buffer.data(), bufferSize, 0) > 0) {  
        file.write(buffer.data(), bytesRead);  
    }  
    file.close();  
    receiveResponse(controlSocket, response);  
    closeSocket(dataSocket);  
    return;  
}
```

COMANDO STOR <FILE>

DEMOSTRACIÓN...



STOR - SERVER SIDE

```
void handleSTORCommand(int controlClientSocket, const std::string& args) {  
    sendResponse(controlClientSocket, "150 Opening data connection.\r\n");  
    int dataSocket = createSocket(dataPort);  
    int dataClientSocket = establishDataConnection(controlClientSocket,  
                                                    dataSocket, dataAddress, dataPort);  
  
    if (recvFile(dataClientSocket, filename)) {  
        response = "226 File transfer successful.";  
    } else {  
        response = "451 File transfer failed.";  
    }  
  
    closeSocket(dataClientSocket);  
    closeSocket(dataSocket);  
    sendResponse(controlClientSocket, response);  
    return;  
}
```

STOR - SERVER SIDE

```
bool recvFile(int dataSocket, const std::string& filename) {  
    const int bufferSize = 1024;  
    std::vector<char> buffer(bufferSize);  
    std::ofstream file(filename, std::ios::binary);  
    ssize_t bytesRead;  
    while (bytesRead = recv(dataSocket, buffer.data(), bufferSize, 0) > 0) {  
        file.write(buffer.data(), bytesRead);  
        std::cout << "Read " << bytesRead << " bytes" << std::endl;  
    }  
    if (bytesRead < 0) {  
        file.close();  
        return false;  
    }  
    file.close();  
    return true;  
}
```


STOR - CLIENT SIDE

```
void sendFile(int controlSocket, const std::string& filename) {  
    std::string response;  
    receiveResponse(controlSocket, response);  
    if (response.substr(0, 3) != "150") {  
        return;  
    }  
    int dataSocket = establishDataConnection(controlSocket);  
    std::ifstream file(filename, std::ios::binary);  
    std::vector<char> buffer(std::istreambuf_iterator<char>(file), {});  
    file.close();  
    ssize_t bytesSent = send(dataSocket, buffer.data(), buffer.size(), 0);  
    closeSocket(dataSocket);  
    receiveResponse(controlSocket, response);  
    return;  
}
```

DEMOSTRACIÓN...