Instituto Superior de Engenharia de Lisboa

Redes de Computadores

Relatório da Primeira Fase do Trabalho Prático

Docente

Prof. Diego Passos

Alunos

53255 Magali dos Santos Leodato
Gabriel Affonso Sbutzki

Índice

Intrdução

Nesta fase do projeto, instalamos e configuramos um servidor web em um PC local, garantindo seu funcionamento adequado. Em seguida, utilizamos um navegador e um cliente web personalizado para estabelecer conexões com o servidor e verificar sua resposta às requisições. Para monitorar a comunicação, empregamos o Wireshark, analisando os pacotes de dados trocados e os cabeçalhos HTTP transmitidos. Este relatório detalha cada etapa do processo, incluindo a configuração do servidor, os testes executados, o desenvolvimento do cliente e a análise aprofundada do tráfego de rede.

Instalação e Configuração do Servidor Web:

Passos Realizados:

1. **Download do XAMPP**: Obtivemos o pacote de servidor web gratuito no Apache Friends.
2. **Instalação do XAMPP**: Seguimos os passos de instalação e iniciamos o servidor Apache.
3. **Teste de Acesso Local**: Abrimos um navegador e acessamos o servidor através do endereço `http://127.0.0.1/`.
4. **Identificação do Endereço IP Local**: Utilizamos `ipconfig` (Windows) ou `ifconfig` (Linux/Mac) para obter o IP do PC.
5. **Teste de Acesso na Rede**: Acessamos o servidor a partir de outro dispositivo usando `http://<IP_DO_PC>/`.
6. **Verificação das Configurações de Firewall**: Desativamos o firewall, se necessário, para permitir requisições HTTP.

Software Utilizado:

Para a realização deste trabalho, foram utilizados os seguintes softwares:

- **Google Chrome**: Navegador utilizado para acessar o servidor web.
- **Wireshark**: Ferramenta de análise de tráfego de rede, empregada para capturar e examinar os pacotes transmitidos entre cliente e servidor.
- **XAMPP**: Pacote de software que inclui o servidor Apache, utilizado para hospedar a aplicação web localmente.
- **Kotlin**: Linguagem de programação utilizada no desenvolvimento da aplicação cliente.

Modelo Cliente-Servidor

O modelo adotado para o desenvolvimento da aplicação foi o **Cliente-Servidor**, pois o objetivo era configurar um servidor web e estabelecer comunicação com ele por meio da aplicação desenvolvida. Nesse modelo, o cliente envia requisições ao servidor, que as processa e responde de acordo com a solicitação recebida.

Protocolo HTTP

A comunicação entre cliente e servidor foi estabelecida por meio de endereços IP e do protocolo **TCP**, utilizando um **Socket** na porta **80**. Assim que o servidor aceita a solicitação de conexão do cliente, inicia-se a troca de mensagens, na qual:

1. O cliente envia um pedido ao servidor.
2. O servidor processa a requisição e retorna uma resposta.
3. Após a conclusão da comunicação, a conexão entre ambos é encerrada.

Como o **protocolo HTTP é stateless**, ou seja, não mantém informações sobre interações anteriores, para cada novo objeto solicitado na página, o processo se repete. Isso significa que uma nova conexão deve ser estabelecida e um novo **Socket** criado na porta **80**, pois o protocolo não armazena dados de sessões anteriores.

Wireshark

A utilização do **Wireshark** foi essencial para capturar e analisar as mensagens trocadas entre cliente e servidor durante os testes. O software permitiu visualizar os pacotes transmitidos, identificar padrões de comunicação e entender melhor o funcionamento do protocolo HTTP dentro do modelo Cliente-Servidor.

Código em Kotlin

```kotlin
import java.io.BufferedReader
import java.io.InputStreamReader
import java.io.OutputStreamWriter
import java.io.PrintWriter
import java.net.Socket

fun main() {
    val serverAddress = "127.0.0.1"
    val serverPort = 80

    try {
        val socket = Socket(serverAddress, serverPort)

        // Enviar requisição HTTP
        val outputStream = PrintWriter(OutputStreamWriter(socket.getOutputStream()), autoFlush: true)
        outputStream.println("GET / HTTP/1.1")
        outputStream.println("Host: $serverAddress")
        outputStream.println("Connection: close")
        outputStream.println() // Linha em branco obrigatória para finalizar o cabeçalho HTTP

        // Ler a resposta do servidor
        val inputStream = BufferedReader(InputStreamReader(socket.getInputStream()))
        println("Resposta do servidor:")
        var responseLine: String?
        while (inputStream.readLine().also { responseLine = it } != null) {
            println(responseLine)
```

```
        // Ler a resposta do servidor
        val inputStream = BufferedReader(InputStreamReader(socket.getInputStream()))
        println("Resposta do servidor:")
        var responseLine: String?
        while (inputStream.readLine().also { responseLine = it } != null) {
            println(responseLine)
        }

        // Fechar recursos
        inputStream.close()
        outputStream.close()
        socket.close()
    } catch (e: Exception) {
        println("Erro ao conectar ao servidor: ${e.message}")
    }
}
```

Este código Kotlin implementa um cliente HTTP básico que se conecta a um servidor por meio de um **socket TCP**, envia uma requisição HTTP do tipo **GET** e imprime a resposta recebida do servidor linha por linha. Ele serve como uma introdução prática ao funcionamento do protocolo HTTP e à manipulação de redes com sockets.

A execução começa com a definição do endereço IP e da porta do servidor ao qual será feita a conexão. O endereço utilizado é `127.0.0.1`, conhecido como **localhost**, que representa a própria máquina do usuário. A porta utilizada é a **porta 80**, padrão para conexões HTTP.

A seguir, o código tenta estabelecer uma conexão com o servidor por meio de um objeto da classe `Socket`. Essa conexão funciona como um canal bidirecional de comunicação, permitindo que dados sejam enviados e recebidos.

Uma vez estabelecida a conexão, o programa prepara a **requisição HTTP** que será enviada. Para isso, ele utiliza uma cadeia de objetos que transformam a saída do socket em uma forma prática de escrita de texto: o `OutputStreamWriter` converte caracteres em bytes e o `PrintWriter` permite o uso de comandos simples como `println()` para enviar linhas de texto.

A requisição enviada segue o formato padrão do protocolo HTTP 1.1. A primeira linha da requisição contém o método `GET`, o caminho `/` (indicando a raiz do site) e a versão do protocolo. Em seguida, o cabeçalho `Host` especifica o endereço do servidor, e o cabeçalho `Connection: close` informa que a conexão deve ser encerrada após o envio da resposta. Após os cabeçalhos, uma **linha em branco** é enviada para indicar o fim da requisição — esta linha é obrigatória segundo as regras do protocolo HTTP.

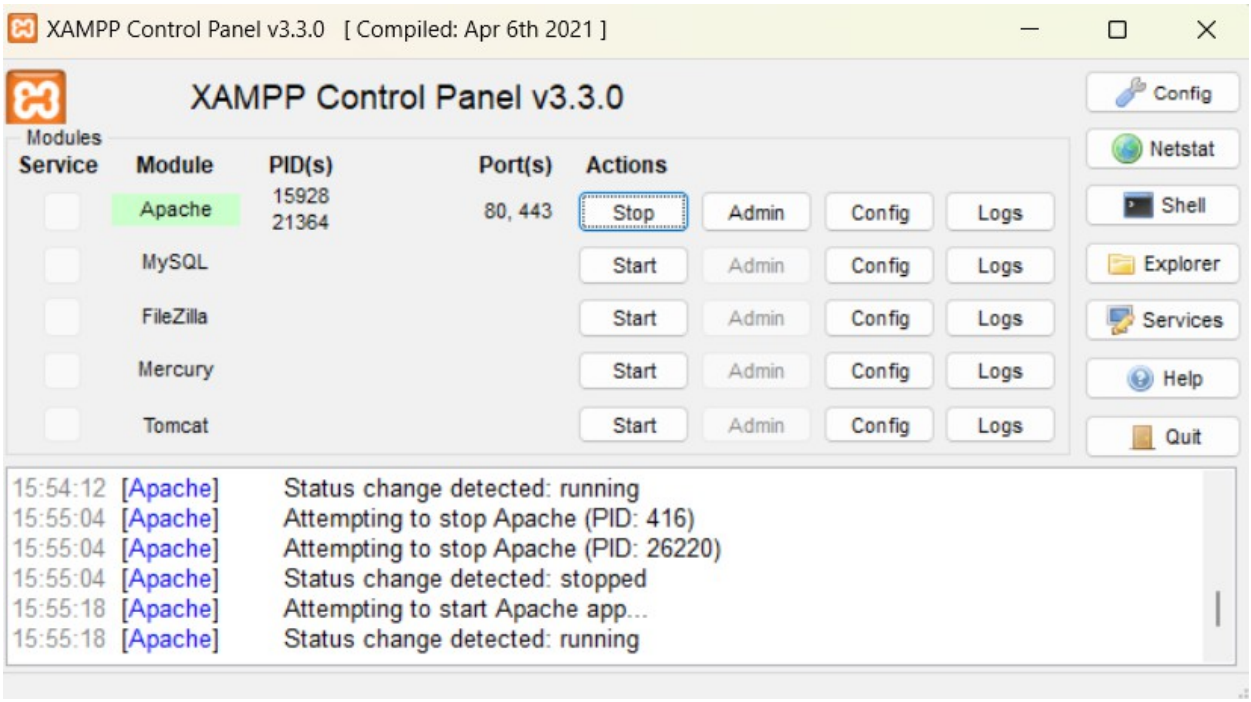
Depois de enviada a requisição, o programa passa a **ler a resposta** do servidor. Ele faz isso por meio de um `BufferedReader`, que lê os dados recebidos através do socket, linha por linha. A cada linha lida, o conteúdo é impresso no console. A resposta pode conter tanto os **cabeçalhos HTTP** quanto o **corpo da resposta** (geralmente em HTML, no caso de servidores web).

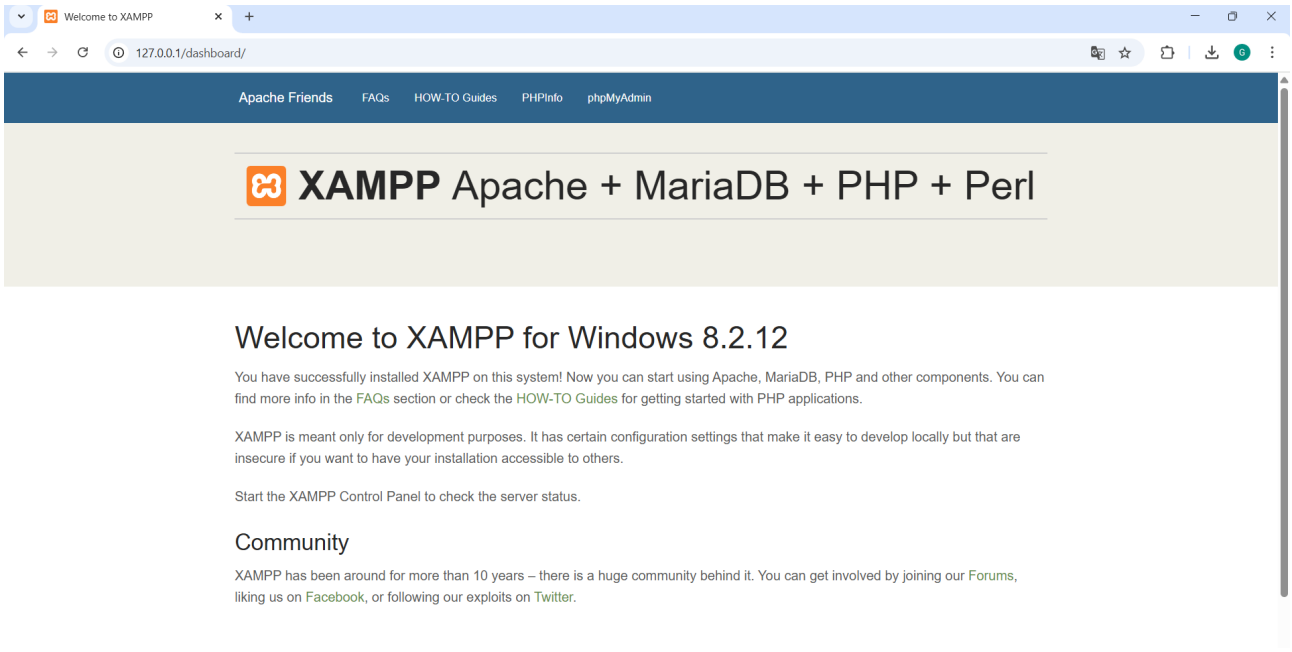
Ao final da leitura, o programa fecha os recursos utilizados: o leitor (`BufferedReader`), o escritor (`PrintWriter`) e o próprio socket, encerrando a comunicação de forma segura.

Todo o processo está envolvido em um bloco `try-catch`, o que permite capturar e tratar eventuais exceções. Caso ocorra algum erro durante a conexão ou a leitura de dados, uma mensagem de erro será exibida com a descrição do problema.

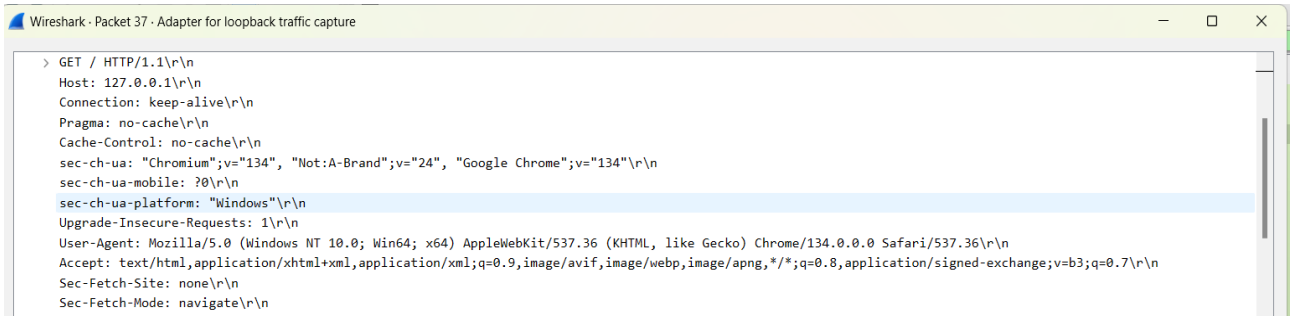
Em resumo, este código é um exemplo prático de como realizar uma requisição HTTP manualmente, sem depender de bibliotecas especializadas. Ele mostra como se comunicam cliente e servidor na web, ajudando a compreender melhor os fundamentos da internet e da troca de dados em redes.

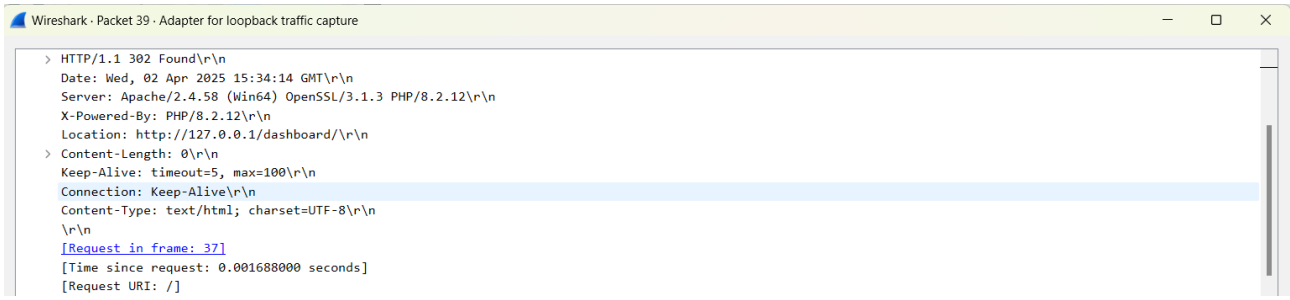
Resultados



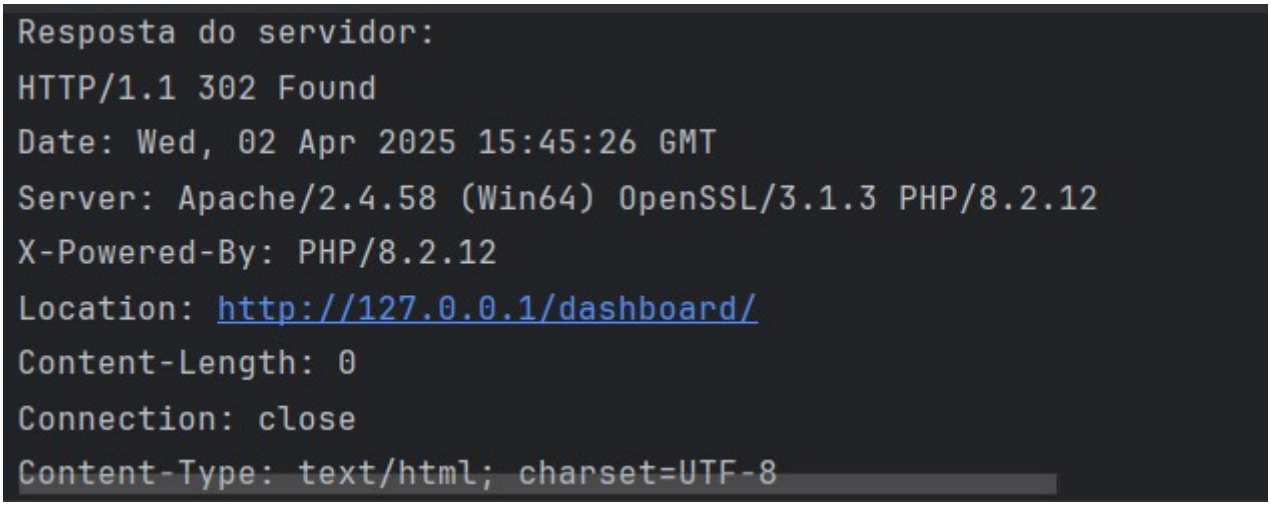


Resultados





Verificar o que vai ser escrito por favor não consigo visualizar ////



Resultados:

A imagem mostra um **redirecionamento HTTP 302**, informando que o cliente (navegador ou programa) deve acessar `http://127.0.0.1/dashboard/` em vez do endereço original. Esse comportamento é comum em sistemas de autenticação, onde o usuário é redirecionado para uma página de login ou dashboard após uma requisição inicial.

Conclusão

A execução do código resultou em uma conexão bem-sucedida com o servidor especificado, enviando uma requisição HTTP `GET` e recebendo a resposta do servidor. O cliente conseguiu imprimir no console o conteúdo da resposta, linha por linha. Caso houvesse algum erro ao conectar ou ao realizar a requisição, uma mensagem de erro apropriada seria exibida, indicando a falha na operação. O código foi capaz de realizar a tarefa de comunicação de forma eficiente, com o fechamento adequado dos recursos utilizados ao final da execução.