# Relatório da Primeira Versão da Programação em Sockets

**Lucas Vilanova[1], Renato Sayyed[1], Willian Domingues[1]**

[1].aluno@unipampa.edu.br

***Resumo.*** *A Internet como conhecemos, é uma ampla rede de computadores interligadas entre si, em todo o mundo. Graças a ela, somos capazes de nos comunicar com qualquer pessoa a partir de qualquer lugar. Mas para que uma simples mensagem seja enviada, existem protocolos que são utilizados para definirem regras e o formato de como as mensagens serão transmitidas entre os sistemas finais (e.g., clientes, servidores). Existem diferentes tipos de protocolos, cada um com suas funcionalidades e objetivos. O HTTP é um dos protocolos mais utilizados no mundo, fornecendo uma interface entre o cliente (e.g., usuário) e o servidor (e.g., página web).*

## 1. Introdução

O navegador Web utilizado por nós, usuários, é um cliente que envia solicitações constantemente ao seu servidor. Porém, para realizar uma simples requisição em um site é necessário utilizar protocolos que garantam a integridade do sistema. Protocolos são um conjunto de regras que definem a ordem e o formato de como as mensagens são transmitidas entre os *hosts*. O HTTP é um protocolo de transferência que permite com que o cliente possa visualizar o conteúdo que existe em uma página Web requisitada por ele a um servidor. Graças a esse protocolo, todas as informações da Internet se tornam acessíveis para nós.

O objetivo desse trabalho é desenvolver e entregar uma primeira versão de um servidor capaz de tratar conexões concorrentes (i.e., vários clientes simultaneamente), seguindo o protocolo HTTP 1.0. Utilizamos a linguagem de programação C para desenvolver este projeto. O código referente a esse trabalho está disponível em nosso repositório GitHub [1].

## 2. Metodologia

Para por em prática a metodologia, primeiro precisamos entender como funciona a relação entre cliente e servidor. Quando um cliente HTTP requisita um objeto (e.g., página Web) a um servidor, utilizando um protocolo TCP, é necessário que ambos estabeleçam uma conexão entre si primeiro. Para estabelecer a conexão, são necessários três passos:

1. Primeiro, o cliente envia uma requisição ao servidor, informando que quer se comunicar com ele;
2. Segundo, o servidor recebe a requisição do cliente, e responde com um pacote de reconhecimento (i.e., ACK), informando que recebeu o pacote do cliente e está disponível para se comunicar com ele;

[1]https://github.com/Overycall/Servidor_Redes

3. Terceiro, quando o cliente recebe a confirmação do reconhecimento do servidor, ele envia novamente um pacote contendo os dados que quer enviar, mas agora confirmando que a conexão já está estabelecida e pronta para a troca de mensagens entre os dois *hosts* (i.e., o *host* cliente e o *host* servidor).

O protocolo da camada de transporte utilizado é o TCP, que é orientado à conexão, i.e., fornece suporte para controlar o congestionamento de rede, controle de erros, perdas de pacotes e garante que os dados sejam recebidos na ordem em que foram enviados, fornecendo um serviço de segurança e confiabilidade, durante a conexão.

### 2.1. Definição de Requisitos

Para sermos capazes de implementar um servidor HTTP, precisamos definir requisitos que serão necessários para executar cada etapa do processo. Em relação a primeira versão do projeto, os requisitos funcionais definidos, são:

- implementar um servidor capaz de tratar conexões concorrentes, i.e., requisições de vários clientes simultaneamente, seguindo o protocolo HTTP 1.0 (i.e., conexões persistentes);
- utilização de bibliotecas necessárias para o funcionamento correto do servidor simulado (e.g., a biblioteca *pthread* que permite o uso de múltiplas threads);

Em relação aos requisitos não funcionais, temos:

- a implementação do servidor será realizada através da linguagem de programação C;
- a utilização do WSL 2 (i.e., distribuição Linux Ubuntu 20.04 LTS para Windows) para desenvolver o projeto;
- o nosso ambiente de desenvolvimento de código será o Visual Studio Code [2];
- a utilização da extensão *C/C++ Compile Run* do Visual Studio Code que permite a compilação e execução do código desenvolvido em C;
- a utilização da metodologia do modelo em espiral para desenvolver esse projeto;
- a utilização do repositório Github para gerenciar e versionar os códigos referentes a esse projeto [3].
- a utilização da ferramenta Google Meet [4] para marcar reuniões entre os integrantes do grupo;

### 2.2. Definição de Tarefas

A organização das tarefas foi planejada para entregar a primeira versão do servidor HTTP dentro do tempo estimado de 14 dias. Em um primeiro momento, definimos qual seria o melhor ambiente para desenvolvermos o projeto. Após isso, procuramos buscar um material que servisse como referência e os estudamos. Em seguida, começamos o desenvolvimento do código propriamente dito.

Nos organizamos de forma a sempre trabalharmos em conjunto. Por exemplo, quando um integrante do grupo implementa o código, outro procura o referencial,

[2] https://code.visualstudio.com
[3] https://github.com/Overycall/Servidor_Redes
[4] https://meet.google.com

dando novas ideias e auxiliando em pesquisas, enquanto outro membro escreve o artigo, e vice-versa. Vale ressaltar que todos os membros fazem tudo, passando por todas as etapas do projeto, revezando as tarefas entre si. Em paralelo a isso, os três integrantes estão sempre mantendo a comunicação do grupo, nunca permitindo que um único integrante fique travado em alguma etapa do processo, por exemplo. Marcamos reuniões semanais no Google Meet para podermos planejar como seria abordado a implementação do projeto.

### 2.3. Desenvolvimento

Utilizando os conceitos abordados em aula sobre sockets e programação com sockets, o mesmo é responsável por realizar a comunicação entre aplicações/processos e realiza a interface para a camada de aplicação, dando acesso aos recursos de rede presentes nas camadas de transporte, rede, enlace e física, como mostra a Figura 1. Em seguida, implementamos os recursos disponíveis nas camadas inferiores à camada de aplicação, pois como um dos requisitos do trabalho é a utilização do protocolo HTTP, temos por definição realizar uma conexão TCP. De acordo com Kurose (2017, pg 169) uma conexão TCP antes de realizar o trabalho principal, estabelece uma conexão segura entre o cliente e o servidor, para então realizar a troca de informação solicitada pelo usuário.

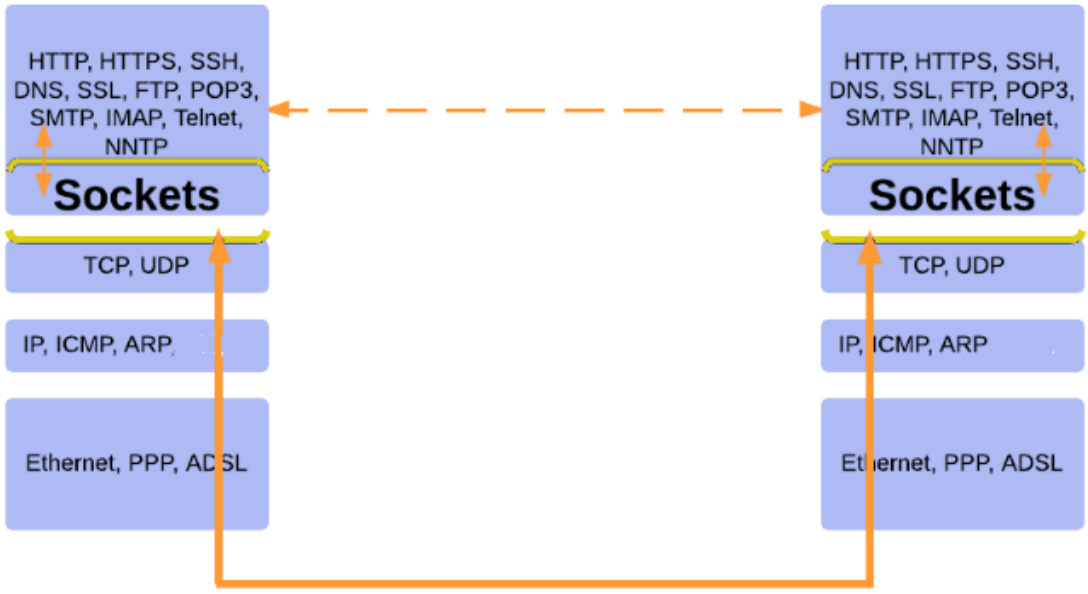


**Figura 1. Camadas TCP/IP e Sockets**

Para a utilização dos recursos de rede disponibilizados pelo sistema operacional, utilizamos algumas bibliotecas (*"sys/socket.h"* e *"netinet/in.h"*) que disponibilizam funções já definidas para a utilização de recursos de *hardware* disponíveis na máquina para então, realizar a escuta em uma porta definida a fim de receber requisições para a conexão/comunicação com outro dispositivo, como vemos na Figura 2.

```
/**
 * Principais funções para escrever programas com sockets
 */

getaddrinfo()  // Traduz nomes para endereços sockets
socket()       // Cria um socket e retorna o descritor de arquivo
bind()         // Associa o socket a um endereço socket e uma porta
connect()      // Tenta estabelecer uma conexão com um socket
listen()       // Coloca o socket para aguardar conexões
accept()       // Aceita uma nova conexão e cria um socket
send()         // caso conectado, transmite mensagens ao socket
recv()         // recebe as mensagens através do socket
close()        // desaloca o descritor de arquivo
shutdown()     // desabilita a comunicação do socket
```

**Figura 2. Funções para programação com sockets**

### 2.3.1. Código (Implementação)

Para a implementação do servidor HTTP 1.0, utilizou-se a linguagem de programação C, uma máquina virtual linux (i.e., Ubuntu 20.04 LTS) em um ambiente Windowns para realizar a compilação do código e também executá-lo. Utilizamos duas referências para estudar o caso e implementá-lo [Tudo, 2018, Trungams, 2020]. O primeiro passo foi declarar todas as bibliotecas e definir todos os rótulos principais do código, como mostrado na Figura 3.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define SERVERNAME "HTTP Server 1.0"
#define SERVERPORT (8085)
#define BUFFERSIZE (1024)
#define MAX_CLIENTS (10)
```

**Figura 3. Definição de bibliotecas e rótulos**

A porta comum para utilização do servidor HTTP é a porta 80, porém o sistema travou a utilização da porta, nos deixando a opção de definir outra porta, no caso a porta 8085, também definimos o tamanho do buffer como 1024 bytes e um máximo de 10 clientes simultâneos realizando requisições para o servidor. O passo seguinte é criar o socket do servidor utilizando a função *socket()*, disponibilizado pela biblioteca *"sys/socket.h"*. Após a instanciação do socket, é realizada a configuração do mesmo nas linhas 67 a 71, definindo o protocolo de comunicação, a porta do servidor e o IP do servidor (i.e., localhost), como mostrado na Figura 4.

```c
60    // Cria o socket do servidor - create()
61    if ((server_socket = socket(AF_INET, SOCK_STREAM, 0)) < 0)
62    {
63        perror("create()");
64        exit(EXIT_FAILURE);
65    }
66
67    // Configuração do socket
68    memset(server_addr.sin_zero, '\0', sizeof(server_addr.sin_zero)); //Zera o restante do struct
69    server_addr.sin_family = AF_INET; //Define o protocolo de comunicação como IPv4
70    server_addr.sin_port = htons(SERVERPORT); //Define a porta do servidor
71    server_addr.sin_addr.s_addr = INADDR_ANY; //Define o IP do servidor
72
```

**Figura 4. Criando e configurando socket do servidor**

O passo seguinte é realizar o link do socket criado e configurado com o endereço do servidor, onde utilizamos a função *bind()*. Em outras palavras, o *bind()*

define o endereço pro *socket* (i.e., IP + PORTA).

### 2.4. Casos de Uso

Um caso de uso define uma sequência de ações executadas pelo sistema que gera um resultado. Em uma transmissão de mensagens entre *hosts*, o servidor é um programa que está sempre em execução, escutando (i.e., esperando) através de uma porta a requisição do cliente.

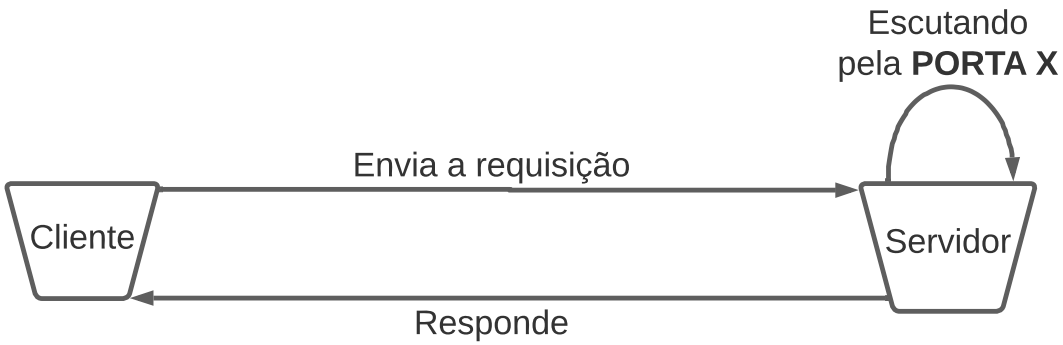


**Figura 5. Diagrama de Caso de Uso**

### 2.5. Casos de Teste

Os casos de testes abrangeram todos os tipos de arquivos definidos em código, sendo eles: html, gif, jpeg, png, au, wav, avi, mp3 e ico. Para cada um deles, foi utilizado um arquivo correspondente. Assim, o servidor foi executado utilizando a porta 8080 pelo terminal em modo super usuário. Após ele estar em execução, foram realizadas as requisições via browser.

- localhost:8080/file.html
- localhost:8080/file.gif
- localhost:8080/file.jpeg
- localhost:8080/file.png
- localhost:8080/file.au
- localhost:8080/file.wav
- localhost:8080/file.avi
- localhost:8080/file.mp3
- localhost:8080/file.ico

Para o html, gif, jpeg, png e ico, os dados são enviados e mostrados diretamente no navegador. Já para o au, wav, avi e mp3, os dados são enviados e o cliente realiza o download do arquivo.

### 2.6. Dificuldades e Tarefas Pendentes

Ainda não utilizamos a biblioteca *pthreads* da linguagem C para fazer com que os servidores tratem múltiplas conexões de clientes. Também implementamos apenas a requisição GET, nessa primeira versão. Além disso, desenvolvemos apenas o envio

das mensagens 200 (OK) e 404 (NOT FOUND) entre o servidor e o cliente. Porém vamos implementar mais mensagens, posteriormente.

Pretendemos fazer essas tarefas pendentes e entregá-las ao professor. Nos próximos dias, iremos ir incrementando essas funcionalidades e ajustando a versão do HTTP para a 1.1.

## 3. Glossário

- ***Host*** - Ou hospedeiro, é qualquer máquina ou computador conectado a uma rede, podendo oferecer informações, recursos, serviços e aplicações aos usuários ou outros nós na rede.
- **Cliente** - Representa uma entidade que consome os serviços de uma outra entidade servidora, em geral através do uso de uma rede de computadores numa arquitetura cliente-servidor.
- **Servidor** - É um programa que fica em execução, escutando (i.e., esperando) através de uma porta, alguma requisição do cliente. Em outras palavras, ele aceita e responde a solicitações feitas por um outro programa.
- **Porta** - É um software ou processo específico que serve como ponto final de comunicações em um sistema operacional hospedeiro de um computador. Uma porta tem associação com o endereço de IP do hospedeiro, assim como o tipo de protocolo usado para comunicação.
- ***Background*** - Processo que fica executando em segundo plano, esperando algum evento acontecer para executar alguma funcionalidade.

## 4. Conclusão

A partir desse relatório, fica claro as etapas que precisam ser processadas para implementar um servidor HTTP. Aprendemos e implementamos, na prática, o que nos foi ensinado em sala de aula, teoricamente. Construímos um servidor, utilizando o protocolo HTTP 1.0, que fica executando em *background* através de uma porta, qualquer requisição vinda de um cliente HTTP.

Também foi simulado a abertura da conexão entre cliente-servidor para a troca de mensagens, e após o término das requisições do cliente, o fechamento da conexão. Na troca de mensagens, conseguimos, como servidor, esperar a requisição de um cliente e fornecer o serviço em questão. Da perspectiva do cliente, pudemos observar que os arquivos (e.g., imagens e gifs) requisitados por nós (i.e., requisição GET), eram nos apresentados pelo servidor, como o esperado. Em relação a arquivos de áudio, quando o cliente o requisita, o arquivo é baixado para sua máquina.

Por fim, salientamos na Seção 2.6 que ficaram algumas tarefas pendentes que iremos concluí-las e enviá-las ao professor, passando por cada etapa da nossa metodologia com o modelo em espiral, para definirmos os próximos passos.

## Referências


Trungams (2020). A simple http server from scratch. `https://trungams.github.io/2020-08-23-a-simple-http-server-from-scratch/`.

Tudo, P. (2018). Servidor http: Tudo o que você precisa saber para construir um servidor http simples do zero. `https://medium.com/from-the-scratch/http-server-what-do-you-need-to-know-to-build-a-simple-http-server-from-scratch-d1ef8945e4fa`.