Primeiro Trabalho Prático de Redes de Computadores

Diego Cintra, Jainor Souza

23 de outubro de 2014

Sumário

1 Introdução **2**

2 Desenvolvimento do Projeto **2**
2.1 Definição da Linguagem 2
2.2 Definição de Conceitos 2
2.3 Servidor, *Request* e *Response* 3
2.3.1 **Servidor** 3
2.3.2 **Request** 3
2.3.3 **Response** 4

3 Experimentos e Resultados **6**

4 Conclusão **7**

5 Referências **7**

Resumo

Este documento refere-se ao primeiro trabalho prático da disciplina de Redes de Computadores, ministrada pela professora Hana Karina Salles Rubensztejn (Facom – UFMS), onde nos foi requisitada a implementação de um servidor simples (desenvolvido utilizando a linguagem C++), capaz de responder a requisições de navegadores regidas pelo protocolo HTTP (*HyperText Transfer Protocol*), realizando a manipulação de seus cabeçalhos de demanda e devolvendo uma resposta apropriada. O servidor deve ser capaz de enviar arquivos de texto e imagem em seus respectivos códigos de status (que, para esse trabalho, são 5), bem como executar sob a utilização de *threads* (para atender diversas requisições) e exibir uma lista de arquivos quando um diretório for pedido. A execução dessas tarefas tem como finalidade ampliar a compreensão do funcionamento de um servidor web e do protocolo HTTP.

1 Introdução

Dentro do modelo de camadas TCP/IP (ou até mesmo o ISO/OSI), temos a camada de aplicação, que basicamente é responsável por permitir a comunicação propriamente dita entre dois hosts. Intuitivamente, existem diversos protocolos implementados nessa camada, entre eles o HTTP (*HyperText Transfer Protocol*), que permite a transferência de arquivos no formato HTML(*HyperText Markup Language*). Ele é regido por um conjunto de regras, que são brevemente descritas a seguir: o protocolo define que existe uma requisição e uma resposta, onde a primeira vem do cliente (*client-side*, ou seja, a aplicação que requisita um arquivo HTML, como um navegador), possuindo um cabeçalho com informações imprescindíveis para o servidor, e a segunda é enviada pela aplicação do servidor (*server-side*), também contendo informações encapsuladas em um cabeçalho. Dentre essas informações, os atributos essenciais para a comunicação entre ambos incluem o tipo de requisição (GET, POST, entre outras), o sinal retornado (200, 301, 404, entre outros), o endereço do host e a versão do protocolo HTTP. São esses atributos e sua manipulação que serão implementadas no projeto e abordadas ao longo desse relatório.

2 Desenvolvimento do Projeto

Para a realização desse projeto, uma série de passos deve ser cumprida, e explicaremos cada um detalhadamente ao longo dessa seção. Para facilitar a compreensão, recomendamos a abertura dos códigos adjuntos a este relatório no momento da leitura. Para agilizar o processo de teste ao leitor, deixamos claro que sempre deve-se adicionar, após iniciar o servidor, o endereço IP da máquina aonde ele está sendo executado, dois pontos (":"), a porta associada ao programa e "/redes/", que é o diretório raiz, no navegador.

2.1 Definição da Linguagem

Em um primeiro momento, numa fase anterior à implementação, alguns aspectos foram esclarecidos entre a dupla, no que diz respeito ao escopo do sistema e quais as principais dificuldades que seriam encontradas. Podendo escolher entre duas linguagens, uma estrutural e outra orientada a objetos, optamos por utilizar o "C++", visto que o programa é composto de diversas partes que, quando fracamente acopladas, permitem um melhor entendimento e correção de possíveis erros. Essa escolha provou ser muito útil pois, ao progredir pela implementação, a manipulação de tipos "String" pôde ser feita com mais facilidade pelo uso da classe genérica "string" implementada na biblioteca "STL", e menos tempo foi dispendido nessas questões.

2.2 Definição de Conceitos

Após a definição da linguagem, o processo de pensamento nos levou a definir os conceitos que seriam utilizados e como se daria o relacionamento deles neste sistema. Com isso,

explicaremos os principais conceitos e suas implementações nessa seção.

2.3 Servidor, *Request* e *Response*

Ao analisarmos a situação, percebemos que um objeto deveria ser responsável por iniciar o servidor e manter seu socket, bem como receber uma requisição de conexão e tratá-la apropriadamente (através da criação de threads). Portanto, uma classe de nome "Servidor" foi criada. Para a captura da requisição enviada pelo navegador e a organização dos principais campos advindos dele, utilizamos a classe "Request". Por último, tendo um vínculo com "Request", a classe "Response" deve instanciar um objeto que seja capaz de analisar os campos da classe relacionada e organizar um cabeçalho de resposta, atribuindo um sinal e procurando, caso haja, os arquivos correspondentes solicitados.

2.3.1 Servidor

No "Servidor", temos como principais atributos o "server_addr " e "sckt" − o primeiro é do tipo "sockaddr_in" (usamos um pequeno "typedef " para minimizar o nome do tipo) e é responsável por preencher a estrutura do *socket* "sckt". No construtor, passamos como parâmetro a porta especificada pelo usuário (ou, caso não haja, o padrão "8080") e inicializamos a estrutura previamente mencionada, definindo características como tipo do protocolo IP, o endereço e a porta. Depois, inicializamos o *socket* "sckt", utilizamos da função "*bind*" para associar essa porta à aplicação e utilizamos "*listen*" para atender a requisições de conexões (que serão feitas pelos browsers), que são limitadas até 25. É importante explicar alguns outros métodos aqui descritos, como o "*run*" e o "*request* ", que serão explicados mais adiante.

No arquivo "Servidor.cpp", temos a implementação das assinaturas que acabamos de mencionar. O método "*run*" é responsável por inicializar o servidor e deixá-lo pronto para receber requisições, criando, nesse processo, uma *thread* e tratando o cliente. O método "*request*" é invocado dentro da *thread*, realizando as principais operações do escopo deste trabalho: receber o cabeçalho enviado pelo navegador, instanciar objetos do tipo "*Request*" e "*Response*" e enviar a resposta ao final. Percebemos que um objeto do tipo "*Response*" é instanciado e, se analisarmos o *header* de "*Response*", podemos ver o vínculo que acontece entre os conceitos do trabalho: um dos atributos é um objeto "*Request*", que é alocado dentro de seu construtor e recebe uma cópia (através da sobrecarga do operador "=") profunda de outro objeto passado como parâmetro. Os outros atributos têm seu entendimento facilitado pela nomenclatura, com exceção de "*buff*" e "fd " , sendo que o último serve para exibir o caminho do arquivo.

2.3.2 Request

A responsabilidade da classe *Request* é bem nítida, principalmente pela grande quantidade de atributos do tipo "std::string" (que substitui a utilização do tipo primitivo "char*"). Todos eles são bastante intuitivos, à exceção do "originalPath" e "alive", que serão explicados mais adiante. Temos dois construtores aqui, sendo que o primeiro (inline) simplesmente define padrões "nulos" para os atributos, tendo em vista que o vínculo entre "Request" e "Response" que havíamos falado irá invocar exclusivamente este método.

No arquivo "Request.cpp", temos um outro construtor (bem maior) responsável por extrair as informações essenciais do cabeçalho do navegador. Os parâmetros passados a ele definem o cabeçalho propriamente dito e a porta que o servidor associou para essa aplicação, respectivamente. Em um primeiro momento, extraímos o tipo do método requisitado - que, de acordo com o escopo do projeto, se limita a "GET " ou "POST". Caso nenhum dos dois seja encontrado, então um tipo "BAD" é associado, sendo interpretado pela classe "Response " mais adiante. No caso especial do "POST", extraímos da última linha do cabeçalho as informações essenciais para o preenchimento do arquivo em questão (que também serão abordadas adiante). Logo após, o campo "Accept" é extraído, e, por se

tratar de um servidor simples, consideramos apenas o primeiro tipo aceitável. Extraímos agora o nome do arquivo e seu caminho, utilizando o método “reverse” devido ao fato de termos lido o cabeçalho de trás para frente. Caso um diretório seja requisitado, então atribuímos ao nome do arquivo a denominação “directory”, para tratamento na outra classe já citada. O protocolo HTTP é capturado aqui (lembrando que só consideramos o “HTTP/1.1”), bem como o endereço IP do host e sua porta. Por fim, o que está escrito em “Connection” é armazenado, e caso seja “close”, o atributo “alive” é atualizado para false, permitindo a terminação do laço em que esses objetos estão sendo instanciados.

2.3.3 Response

Em “Response.cpp”, temos todos os métodos assinados em seu arquivo header “Response.h”. O método “mountResponse” é o principal da classe, que consequentemente tem a maior responsabilidade: analisar as entradas do objeto “req” e, com base nisso, montar um cabeçalho de resposta apropriado. Os desvios condicionais aqui presentes são bastante simples de entender, tendo o conhecimento dos sinais de resposta que foram implementados no projeto - a ordem, entretanto, deve ser considerada: caso a mensagem não seja entendida pelo servidor (sinal “400”), não há necessidade de analisar o tipo do protocolo. A mesma condição ocorre entre o sinal “301” e a listagem de diretórios, visto que um único arquivo específico foi requisitado para demonstrar o funcionamento do “Moved Permanently”. Para facilitar o preenchimento dos sinais que possuem um campo de dados padronizado(como “404”, “400 ” e “505”), métodos com nomes sugestivos (“notFound”, “badRequest” e “httpVersion”) foram adicionados, retornando um tipo “std::string” para preencher o atributo “data”. É importante lembrar também que, para o teste do sinal “400”, deixamos um comentário no arquivo “Servidor.cpp” que substitui a primeira letra da requisição por “B”, fazendo com que essa mensagem (“Bad Request ”) seja enviada. Poderíamos ter associado esse sinal em todas as ocasiões em que a leitura do cabeçalho por “req” foi mal sucedida, entretanto entendemos que o conceito aqui em questão é retornar essa mensagem caso o método requisitado seja diferente de “GET” ou “POST”. O mesmo acontece com o sinal “505” : os comentários que substituem o protocolo por “HTTP/1” devem ser ativados somente para o funcionamento da requisição do arquivo “index.html”. A mensagem, entretanto, será retornada a um navegador que suporte o protocolo 1.1 (visto que esses comentários foram inclusos para mero teste), portanto caso o leitor queira realmente utilizar um navegador com o protocolo antigo, esse cabeçalho de resposta deve ser alterado manualmente (logo citaremos qual método toma conta disso).

O método “searchFile”, como o próprio nome já diz, tenta buscar o arquivo que foi requisitado pelo browser, utilizando as funções da biblioteca “dirent.h” para a execução de tal tarefa, como “opendir” e “readdir ”. O parâmetro “flag” é utilizado para definir um comportamento adicional desse método, aonde os arquivos encontrados pelas funções supracitadas são adicionados ao atributo “files”, que é um vetor (“std::vector”) de elementos “std::string” ; sua utilização será abordada mais tarde. Atentando-se ao escopo em questão, no evento do arquivo encontrado (identificado pela variável “entry->d_name ”) ter o nome idêntico ao requisitado, então o sinal “200” é atribuído, e busca-se os dados dele, através do método “getData”. Pulando aqui, temos a abertura de um arquivo em modo binário, visto que foi requisitado que no mínimo uma página contendo uma imagem seja enviada. A utilização de nosso atributo “buff” definido como um vetor é aplicada agora: definimos o seu tamanho através do método “resize” (que recebe como parâmetro o comprimento do arquivo) e lemos todos os caracteres para cada respectiva posição do vetor, através da função “fread”. Após concluir a leitura do arquivo, construímos uma string auxiliar para encapsular todos os bytes capturados, para só então atribuir essa ao campo “data” (previamente construído em sua declaração).

Finalizado a busca do arquivo (e a confirmação de sua existência), informações adicionais ao preenchimento do cabeçalho são encontradas: o método “getDate”, como era de se esperar, busca a data e hora atual e as retorna para armazenamento feito pelo atributo

“date”, e o tamanho do arquivo também é buscado através de “getContentLength”. Agora, vamos analisar os métodos “moved”, “listDirectory” e “POSTResponse”.

O primeiro é invocado quando o arquivo “moved.html” é buscado na pasta principal (lembrando que, para ter acesso ao servidor, é preciso encontrar o IP da máquina onde ele está sendo executado e concatenar, depois de “:”, a porta e “/redes/ ”), e em sua composição, o código “301 Moved Permanently” é assinalado ao campo “code”, e há a chamada de “findDirectory ”: apesar de possuir um comportamento muito similar ao “searchFile ”, sua finalidade é diferente, pois ele é invocado recursivamente caso diretórios sejam encontrados. Ao longo de sua recursão, o parâmetro “caminho” é preenchido com exatamente o que seu nome diz, da pasta raiz até o diretório contendo o arquivo “moved.html”, tomando como base o atributo “path” do objeto “req”. No evento desse caminho não estar vazio, alteramos a variável “fd” para conter o caminho correto concatenado com o nome do arquivo, e buscamos o conteúdo dele novamente através de “getData”.

Já para “listDirectory”, a utilização da denominação “directory” é útil nesse momento, servindo de condição para chamada do método. Dentro dele, tudo o que temos a fazer é invocar o “searchFile” com o sinal “1”, executando o procedimento adicional de armazenamento dos arquivos encontrados no atributo “files”, que é um vetor de “std::string”. Um tratamento adicional é feito para diretórios, que é a inclusão do backslash - como a requisição do projeto obrigou-o a ser executado em um ambiente Linux, a barra invertida (de uso do sistema operacional Windows) pôde ser desconsiderada. Após o fim do método, voltamos ao “listDirectory ”, onde começamos a criar um documento HTML 5 que, iterando por todos os elementos de “file”, cria uma tag âncora onde o atributo de link é preenchido com o atributo “originalData”. Isto foi feito pois, caso o usuário quisesse navegar pelos diretórios (ou abrir arquivos), o método “getCurrentDir” se encarregará de preencher o restante do caminho (caso não houvesse “originalData”, então seriam inseridas duas vezes o caminho). Por fim, o documento é finalizado e associado ao “dat a”. O sinal “200” também é associado aqui.

Temos, finalmente, a utilização do “POSTResponse”, que é invocado mediante a solicitação do arquivo “post.html”, que contém em seu documento as tags “input” que permitem armazenar as informações do cliente e repassá-las ao servidor pelo cabeçalho (identificado exclusivamente pelo comando “POST”). Sendo novamente um projeto simples, não foram abordados aqui a manipulação das variáveis correspondentes a esse método por arquivos com extensão “php” - apesar do arquivo final conter ao fim “.php”, em sua essência o documento é do tipo “html”. Dado isso, dentro de “POSTResponse”, extraímos os campos correspondentes do atributo “post” (do objeto “req”) e armazenamo- os nas variáveis “name” e “value”, que respectivamente indicam o que o usuário passou como valor e o atributo “name” da tag “input”. Após esse preenchimento, o método “fillIn ” é ativado, permitindo com que a frase “$_POST” contendo entre colchetes o “value” correspondente seja substituída por “name”, e como o campo “data” já contém esse arquivo, essa troca acontece aí mesmo. O processo é repetido até que não hajam mais palavras do atributo “post” a serem substituídas.

Após todas essas informações serem processadas, cabe apenas ao objeto organizar adequadamente o cabeçalho de resposta e retorná-lo a quem invocou seus métodos em “Servidor.cpp”, e aqui entra o método “toStr”. Em seu corpo, temos a instanciação de uma variável auxiliar “buffer2” (que armazenará o cabeçalho organizado) e a conversão da variável “size” do tipo “int” para “char*”, através da função “snprintf” (a utilização de “atoi” é depreciada em C++). É necessário testar apenas se o código gerado é igual a “301”, já que o cabeçalho deste possui um campo adicional, “Location”, que aqui é embutido (lembrando que, de acordo com o que foi especificado, o navegador deve automaticamente utilizar esse campo para fazer de imediato a requisição da página). O cabeçalho do restante é basicamente o mesmo. Nesse momento, temos a questão do tipo de protocolo HTTP aberta: caso o leitor queira realmente testar o sinal “505” em um navegador que utilize o protocolo depreciado, pode-se mudar aqui mesmo, substituindo o

método “this->req->getHttp_version()” por “HTTP/1”. O “header_size” finalmente tem sua atribuição feita, sendo preenchido com o tamanho do cabeçalho sem o campo “data”, e esse é adicionado, concluindo o cabeçalho que é então enviado para o objeto da classe “Servidor”.

Tendo a mensagem organizada e recebida, tudo que se faz aqui é enviar para o browser requisitante a resposta, utilizando o somatório dos atributos “header_size” e “size” de “resposta” para definir o tamanho da mensagem. Por fim, como parte das requisições, um pequeno arquivo de log é aberto (ou criado, caso não exista) e a ele são escritos o endereço IP do navegador que fez a solicitação, que tipo de método foi requisitado (“GET” ou “POST”), o arquivo (que no caso será preenchido com “directory” caso uma pasta tenha sido pedida) e a data e hora da requisição, o que acontece dentro de um semáforo, visto que a abertura dele simultaneamente por duas diferentes threads poderia ocasionar problemas. O laço “while” só é finalizado ou ao expirar-se o tempo de espera definido na função “setsockopt”, que recebeu esse tempo da variável “option” ou ao campo “Connection” ter sido enviado com “close”. De volta ao método “run”, o servidor roda indefinidamente até que um sinal SIGINT seja enviado, terminando a execução do programa.

3 Experimentos e Resultados

Após a definição dos conceitos e a fixação do escopo do trabalho, o processo de implementação começou. A manipulação de uma requisição do navegador acabou se tornando uma tarefa monótona, visto que deve haver um parsing para cada um dos atributos necessários à montagem da resposta, o que exigiu uma quantidade de tempo considerável no desenvolvimento desse projeto, principalmente por diversos pequenos erros que surgiram. Outro problema que tivemos foi em relação aos navegadores, que às vezes requisitava o arquivo “favicon.ico”, que acabava trazendo, em algumas situações, um comportamento inesperado do programa.

A questão dos arquivos de imagem (que deveriam ser passados pelo servidor) foi algo bastante problemático, pois apesar da implementação que obtém a imagem ser teoricamente funcional, isso não se mostrou verdadeiro durante a execução, e bastante tempo foi gasto para poder sanar esse problema. Tivemos também que lidar com alguns problemas relativos aos cookies dos navegadores, pois às vezes a análise do atributo “Last modified since” permitia ao navegador recuperar a mesma página que ele havia baixado, o que impedia testes de funcionamento do programa. Talvez a maior dificuldade desse projeto tenha sido a implementação da resposta ao se requisitar um diretório, pois cada arquivo deve ser encapsulado em uma tag “<a> ”da linguagem HTML, permitindo com que o usuário realize uma outra requisição ao clicar sobre a mesma. No fim do projeto, a última das adversidades encontradas foi relativa à implementação do programa com *threads*, visto que a depuração dessas é bastante difícil de ser realizada, e questões como seção crítica e eliminação de *threads* tiveram que ser consideradas. A implementação final desse tópico ficou duvidosa, pois algumas requisições não são corretamente atendidas simultaneamente.

Dado que omitimos o envio por parte do servidor do sinal “Connection”, a única ocasião em que a conexão é fechada é quando o tempo específico para timeout é atingido, visto que quase nenhum navegador envia o campo “Connection” com “close”. Apesar disso, como uma decisão de projeto, optamos por deixar o servidor dessa maneira mesmo, visto que aqui estamos com o “HTTP/1.1”, que preza a persistência da conexão. Portanto, algumas peculiaridades de execução podem ser encontradas caso o usuário faça diferentes requisições nesse intervalo de tempo de timeout. Fora esses quesitos, o projeto pôde ser devidamente implementado e concluído com todas as suas funcionalidades.

4 Conclusão

Tendo como base as aulas de "Redes de Computadores" ministradas no curso de Ciência da computação na Universidade Federal do Mato Grosso do Sul, pudemos facilmente abstrair os principais aspectos necessários para se começar a implementar o projeto. Entendemos o que acontece sob o protocolo HTTP quando se faz uma requisição via URL, não só do lado do cliente como do lado do navegador, e percebemos o quão importante a tarefa do servidor realmente é.

Fomos capazes, portanto, de abstrair sobre a importância de um navegador em nosso âmbito, o comportamento desse, seus gargalos e principais características, sendo ao fim capazes de compreender, mesmo que com alguns comportamentos inesperados de execução, o funcionamento de um server e, na medida do possível, implementar um para manipulação e entrega de arquivos server-side.

5 Referências

HALL, Brian "Beej Jorgensen". (2012) "Beej's Guide to Networking Programming Using Internet Sockets", http://beej.us/guide/bgnet/output/html/singlepage/bgnet.html, Julho.

Lawrence Livermore National Laboratory e BARNEY, Blaise. (2014) "POSIX Threads Programming", Julho.

Cplusplus.com. (2014) "The C++ Resources Network", http://www.cplusplus.com, Outubro.

Stack Exchange Inc. (2014) "Stack Overflow", http://stackoverflow.com, Outubro.