Instituto Tecnológico de Aeronáutica
Divisão de Ciência da Computação (IEC)
Ces-35 – Redes De Computadores e Internet

Laboratório: Implementando Cliente e Servidor HTTP
Em grupos de até 3 alunos
Fazer uso do git (ou github): programas sem histórico de versão não serão considerados

Neste laboratório, o objetivo é ensinar programação de sockets através do design de um protocolo HTTP básico e por meio de desenvolvimento de um servidor e cliente Web simples. Todas as implementações devem ser escritas em C ou C++ usando a biblioteca padrão sockets BSD. Para o laboratório não será permitido bibliotecas de abstração de rede de mais alto nível como Boost.ASIO ou similares). O laboratório pode ser feito com uso de abstrações incluídas por padrão em C++11, como por exemplo, análise de string, multi-processos e multi-threading.

O laboratório possui 2 trechos de código a serem desenvolvidos: um arquivo contendo a implementação do servidor Web e um outro arquivo contendo o cliente Web. O servidor Web aceita uma solicitação HTTP, analisa a solicitação e procura o arquivo solicitado dentro do diretório de arquivos local. Se o arquivo solicitado existir, o servidor retornará o conteúdo do arquivo como parte de uma resposta HTTP, caso contrário, deverá retornar a resposta HTTP apropriada, com o código correspondente. O cliente ao recuperar a resposta Web deve salva-la em arquivo no diretório local. A ideia é implementar o equivalente ao HTTP 1.0, onde as conexões não são persistentes, é preciso um socket por objeto.

Como elaborar o código?

1) Abstração da mensagem HTTP por meio de objeto em C++.

Uma ideia inicial para abordar esse projeto é voce desenvolver um conjunto de classes auxiliares que podem ajudá-lo a analisar e construir uma mensagem HTTP, que pode ser tanto uma solicitação HTTP ou uma resposta HTTP. Para esse projeto só é preciso implementar a solicitação GET.

HTTPReq request;
HTTPResp response;

Alguns métodos possíveis para essa implementação seriam:

request.setURL(...);
request.setMethod (...);
vetor <uint8_t> bytecode = request.encode();

response.setStatusCode (...);

Também seria conveniente ter um método de construção da requisição HTTP (objeto HTTPReq) a partir do binário recebido pelo socket.

request.parse(bytecode);

2) Servidor Web

Após finalizar as classes de abstração, a ideia é desenvolver o servidor Web como um arquivo binário sendo executado pela linha de comando, e aceitar 3 argumentos: nome do host do servidor web, número de porta e nome de diretório onde ficarão os arquivos a serem servidos.

$ web-server [host] [port] [dir]

Um possível exemplo, para nome do host do servidor web, que está usando o nome especial "localhost" que faz um loop para a própria máquina, e que esteja escutando na porta 3000 e servidor arquivos no diretório /tmp.

$ ./web-server localhost 3000 /tmp

(se quiser pode configurar argumentos padrão, como esses acima).

Algumas atividades que o servidor web precisa realizar incluem: a) converter o nome do host do servidor em endereço IP, abrir socket para escuta neste endereço IP e no número de porta especificado. b) por meio do socket "listen" aceitar solicitações de conexão dos clientes, e estabelecer conexões com os clientes. c) Fazer uso de programação de redes que lide com conexões simultâneas (por exemplo, por meio de multiprocess e multithreads). Ou seja, o servidor web deve poder receber solicitações de vários clientes ao mesmo tempo.

Sobres os testes, uma vez implementado o servidor Web simples, voce pode testar o mesmo através do uso de um browser web comum, como Firefox, ou o comando wget do linux no ambiente de desenvolvimento.

3) Cliente Web

A última parte desse laboratório é desenvolver o cliente Web que será um arquivo binário sendo executado pela linha de comando, que aceitará como argumentos várias URLs. Cada argumento será uma URL.

$ ./web-client [URL] [URL] ...

Um possível exemplo, poderia ser a solicitação de um arquivo index.html do servidor web local, na mesma máquina. Usando um comando assim.

$ ./web-client http://localhost:4000/index.html

O cliente Web precisa então, 1) segmentar a string da URL e interpretar os parametros da mesma como hostname, porta e o objeto a ser solicitado. 2) o cliente precisa se conectar por TCP com o servidor Web. 3) Assim que a conexão for estabelecida, o cliente precisa construir uma solicitação HTTP e enviar ao servidor Web e ficar bloqueado aguardando uma resposta. 4) Após receber a resposta, o cliente precisa analisar se houve sucesso ou falha, por meio de análise do código de resposta. Se houver sucesso, ele deve salvar o arquivo correspondente no diretório atual usando o mesmo nome interpretado pela URL.

Sobre os testes, voce pode testar sua implementação buscando dados de servidores web reais usando a sua implementação de cliente como se fosse um browser.

4) Observações sobre a implementação

Lembre-se que uma solicitação GET é delimitada no socket pelo recebimento da linha em branco (em binário "\r\n\r\n"). Já a resposta 200 OK é mais envolvida e requer um parsing de um cabeçalho HTTP para descobrir o tamanho do conteúdo e a leitura de N bytes correspondentes a esse tamanho.

Os arquivos de teste não devem ser maiores que 1MB.

A implementação deve suportar somente os seguintes códigos de status: 200 OK, HTTP 400 Bad Request, 404 Not Found.

Se a solicitação tiver somente um "/" na URL, deve ser capturado o arquivo index.html no servidor web, se ele tiver esse arquivo. Do contrário, 404.

5) Ambiente de Desenvolvimento

O mais fácil para realizar o projeto é usar um ambiente de desenvolvimento baseado em Linux padronizado. Para tanto, basta ter Virtualbox e Vagrant instalados. Assista esse vídeo [1] se não souber instalar. E depois em um diretório novo, chamado lab02, copiar esse conteúdo abaixo, em um arquivo chamado Vagrantfile.

```
Vagrant.configure(2) do |config|
 config.vm.box = "ubuntu/bionic64"
 config.vm.provision "shell", inline: <<-SHELL
   sudo apt-get update
   sudo apt-get install -y build-essential
   sudo apt-get install -y vim
   sudo apt-get install -y emacs
 SHELL
end
```

Depois executar o comando:

```
$ vagrant up
$ vagrant ssh
```

Uma vez dentro da VM, com ssh, criar , usando vim, emacs ou nano, os dois arquivos web-server.cpp e web-client.cpp com esse template e gerar um arquivo Makefile. De modo que voce possa dar os comandos:

```
$ make web-server
$ make web-client
$ make
$ make tarball
```

```
#include <string>
#include <thread>
#include <iostream>
```

```
int main() {
}
```

E um Makefile

```
CXX=g++
CXXOPTIMIZE= -O2
CXXFLAGS= -g -Wall -pthread -std=c++11 $(CXXOPTIMIZE)
USERID=SEU_NOME_AQUI
CLASSES=SUA_LIB_COMUM

all: web-server web-client

web-server: $(CLASSES)
	$(CXX) -o $@ $^ $(CXXFLAGS) $@.cpp

web-client: $(CLASSES)
	$(CXX) -o $@ $^ $(CXXFLAGS) $@.cpp

clean:
	rm -rf *.o *~ *.gch *.swp *.dSYM web-server web-client *.tar.gz

tarball: clean
	tar -cvf $(USERID).tar.gz *
```

---

6) Nota

Ao final do desenvolvimento do projeto, gerar o tarball e submeter via Dropbox. Para implementação single-threaded, o grupo de alunos receberá a nota 7.0. Para implementação multi-threaded será considerada notas de 8 a 10, dependendo se todas as características pedidas forem implementadas.