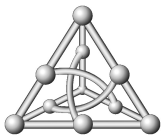

UNIVERSIDADE FEDERAL DE MATO GROSSO DO SUL
Faculdade de Computação

Trabalho 01: Cliente e Servidor

Fundamentos de Redes de Computadores

1 Descrição

Este trabalho deve ser entregue no Moodle até a data correspondente de entrega. Envie sua resposta somente em texto a não ser que outro formato seja absolutamente necessário. Sinta-se encorajado para trabalhar em grupo nesta atividade (contudo, sua entrega é individual).

Este trabalho é composto de uma parte a ser implementada em Python e outra em C/C++. Sua atividade é responder as perguntas em **negrito**.

2 Python

2.1 Cliente

Execute o Cliente HTTP Python [4]. Escolha três páginas web e execute o programa para estas páginas. Servidores Web geralmente usam a porta 80 para conexão.

```
#!/usr/bin/python
# A simple http client that retrieves the first page of a web site

import socket, sys

if len(sys.argv)!=3 and len(sys.argv)!=2:
    print "Usage : ",sys.argv[0]," hostname [port]"
hostname = sys.argv[1]
if len(sys.argv)==3 :
    port=int(sys.argv[2])
else:
    port = 80

READBUF=16384
s=None

# size of data read from web server
for res in socket.getaddrinfo(hostname, port, socket.AF_UNSPEC, socket.SOCK_STREAM):
    af, socktype, proto, canonname, sa = res
    # create socket
    try:
        s = socket.socket(af, socktype, proto)
    except socket.error:
        s = None
        continue
    # connect to remote host
```

```
try:
    print "Trying "+sa[0]
    s.connect(sa)
except socket.error, msg:
    # socket failed
    s.close()
    s = None
    continue
if s :
    print "Connected to "+sa[0]
    s.send('GET / HTTP/1.1\r\nHost:'+hostname+'\r\n\r\n')
    finished=False
    count=0
    while not finished:
        data=s.recv(READBUF)
        count=count+1
        if len(data)!=0:
            print repr(data)
        else:
            finished=True
    s.shutdown(socket.SHUT_WR)
    s.close()
    print "Data was received in ",count," recv calls"
    break
```

Listing 1: Cliente HTTP simples em Python.

1. Anote as URLs das páginas que você escolheu e o número de requisições de chamada para obter a página principal de cada página web.

Se você não tem familiaridade com Python, entenda que Python usa a identação para mostrar a estrutura de um programa, logo, tenha certeza que a identação do seu código segue a mesma identação apresentada aqui. Lembrem-se que copiar e colar pode mudar a identação e, nesse caso, você precisará ajustar isso manualmente.

Se você executar este exercício corretamente, deverá ver um cabeçalho de resposta HTTP enviado pelo servidor, assim como o seu conteúdo. Você precisará dessa informação para a parte 2.5. Algumas páginas enviam devolta uma grande quantidade de dados, você precisará rolar a barra para trás ou procurar uma página web que envie menos dados.

Se você observar com cuidado, notará que o cabeçalho termina com a sequência `\r\n\r\n`, que é um fim de linha seguido por uma linha em branco.

2.2 Modificando o Cliente

Modifique o cliente da parte 2.1 para que ele possa requisitar uma URL arbitrária. Você precisará modificar a linha que começa com `s.send` para requisitar um caminho diferente de somente "/". Você precisará também escrever algum código que separe a URL de coisas como:

`http://prof.facom.ufms.br/~brivaldo/pages/project/`

em seus componentes: o endereço do host é `prof.facom.ufms.br` e o caminho requisitado é `/~brivaldo/pages/project/`. O método `split` de strings pode ser útil. Por exemplo, você pode usar: `parts = url.split("/")`.

2. Modifique e envie seu código. Se seu código funcionar, coloque no seu relatório a quantidade de requisições necessárias para obter: "https://www.ufms.br/cursos/graduacao/".

2.3 Cliente de Alto Nível

Modifique o segundo cliente 2 para receber uma URL como argumento e salvar o resultado em um arquivo chamado: `dados_web.html` (mesmo que para algumas URLs, os dados não sejam HTML).

```
#!/usr/bin/python
# A simple http client that retrieves the first page of a web site, using
# the standard httplib library

import httplib, sys

if len(sys.argv)!=3 and len(sys.argv)!=2:
    print "Usage : ",sys.argv[0]," hostname [port]"
    sys.exit(1)

path = '/'
hostname = sys.argv[1]
if len(sys.argv)==3 :
    port = int(sys.argv[2])
else:
    port = 80

conn = httplib.HTTPConnection(hostname, port)
conn.request("GET", path)
r = conn.getresponse()
print "Response is %i (%s)" % (r.status, r.reason)
print r.read()
```

Listing 2: Cliente HTTP avançado em Python.

3. Modifique, envie seu código e identifique se ele funciona.

2.4 Servidor

Execute o servidor HTTP simples 3 na sua máquina local e conecte nele usando vários clientes web, incluindo seu próprio código anteior e pelo menos um navegador de Internet normal. Se você estiver executando seu servidor na porta X (por exemplo, na porta 6789), você pode usar a URL `http://localhost:6789` para conectar no servidor usando a mesma máquina.

Agora modifique o servidor para imprimir cada requisição, assim você será capaz de ver que requisições seus clientes estão enviando.

4. Informe o nome do cliente web que você usou para conectar no seu servidor web (Safari, Edge, Firefox, etc.) e determine todas as requisições que o navegador fez ao servidor. Pode ser apenas uma requisição ou podem ser várias. Apenas observe cuidadosamente e informe quais foram as requisições.

```
# An extremely simple HTTP server
import socket, sys, time

# Server runs on all IP addresses by default
HOST=''
# 8080 can be used without root priviledges
PORT=8080
BUFLEN=8192 # buffer size

s = socket.socket(socket.AF_INET6, socket.SOCK_STREAM)
try:
    print "Starting HTTP server on port ", PORT
    s.bind((HOST,PORT,0,0))
except socket.error :
    print "Cannot bind to port :",PORT
    sys.exit(-1)

s.listen(10) # maximum 10 queued connections

while True:
    # a real server would be multithreaded and would catch exceptions
    conn, addr = s.accept()
    print "Connection from ", addr
    data=''
    while not '\n' in data : # wait until first line has been received
        data = data+conn.recv(BUFLEN)
    if data.startswith('GET'):
        # GET request
        conn.send('HTTP/1.0 404 Not Found\r\n')
        # a real server should serve files
    else:
        # other type of HTTP request
        conn.send('HTTP/1.0 501 Not implemented\r\n')
    now = time.strftime("%a, %d %b %Y %H:%M:%S", time.localtime())
    conn.send('Date: ' + now +'\r\n')
    conn.send('Server: Dummy-HTTP-Server\r\n')
    conn.send('\r\n')
    conn.shutdown(socket.SHUT_RDWR)
    conn.close()
```

Listing 3: Servidor HTTP em Python.

2.5 Modificando o Servidor

Modifique o servidor para retornar o conteúdo de um único arquivo chamado `index.html`. Você deve criar manualmente este arquivo para testar em conjunto com o seu servidor. Veja um exemplo de um código HTML se você nunca viu um antes:

```
<html>
<head><title>Web Page do Meu Servidor Web</title></head>
<body>Esta &eacute; uma p&aacute;gina Web.</body>
</html>
```

O cabeçalho que o seu servidor retorna deve conter pelo menos algum dos cabeçalhos que vimos na parte 2.1, especificamente: `Connection: close`, `Server: X` (sendo X o nome que você escolheu para o servidor), `Content-Type: text/html` e `Content-Length: Y` (sendo Y o número de *bytes* obtidos do arquivo `index.html`).

5. Envie seu código do servidor modificado.

3 C/C++

3.1 Cliente

Execute o Cliente HTTP C [4]. Escolha três páginas web e execute o programa para estas páginas. Servidores Web geralmente usam a porta 80 para conexão.

```c
/* hw2-simple-client.c: program to connect to web server. */
/* compile with: gcc -Wall -o hw2-simple-client hw2-simple-client.c */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <arpa/inet.h>
#include <netdb.h>

/* maximum size of a printed address -- if not defined, we define it here */
#ifndef INET6_ADDRSTRLEN
#define INET6_ADDRSTRLEN        46
#endif /* INET6_ADDRSTRLEN */

#define BUFSIZE 1000

/* print a system error and exit the program */
static void error (char * s)
{
  perror (s);
  exit (1);
}
```

```c
static void usage (char * program)
{
  printf ("usage: %s hostname [port]\n", program);
  exit (1);
}

static char * build_request (char * hostname)
{
  char header1 [] = "GET / HTTP/1.1\r\nHost: ";
  char header2 [] = "\r\n\r\n";
  /* add 1 to the total length, so we have room for the null character --
   * the null character is never sent, but is needed to make this a C string */
  int tlen =  strlen (header1) + strlen (hostname) + strlen (header2) + 1;
  char * result = malloc (tlen);
  if (result == NULL)
    return NULL;
  snprintf (result, tlen, "%s%s%s", header1, hostname, header2);
  return result;
}

/* must be executed inline, so must be defined as a macro */
#define next_loop(a, s) { if (s >= 0) close (s); a = a->ai_next; continue; }

int main (int argc, char ** argv)
{
  int sockfd;
  struct addrinfo * addrs;
  struct addrinfo hints;
  char * port = "80";  /* default is to connect to the http port, port 80 */

  if ((argc != 2) && (argc != 3))
    usage (argv [0]);
  char * hostname = argv [1];
  if (argc == 3)
    port = argv [2];
  bzero (&hints, sizeof (hints));
  hints.ai_family = AF_UNSPEC;
  hints.ai_socktype = SOCK_STREAM;
  if (getaddrinfo (hostname, port, &hints, &addrs) != 0)
    error ("getaddrinfo");
  struct addrinfo * original_addrs = addrs;

  while (addrs != NULL) {
    char buf [BUFSIZE];
    char prt [INET6_ADDRSTRLEN] = "unable to print";
    int af = addrs->ai_family;
    struct sockaddr_in * sinp = (struct sockaddr_in *) addrs->ai_addr;
    struct sockaddr_in6 * sin6p = (struct sockaddr_in6 *) addrs->ai_addr;

    if (af == AF_INET)
      inet_ntop (af, &(sinp->sin_addr), prt, sizeof (prt));
    else if (af == AF_INET6)
      inet_ntop (af, &(sin6p->sin6_addr), prt, sizeof (prt));
```

```
    else {
      printf ("unable to print address of family %d\n", af);
      next_loop (addrs, -1);
    }

    if ((sockfd = socket (af, addrs->ai_socktype, addrs->ai_protocol)) < 0) {
      perror ("socket");
      next_loop (addrs, -1);
    }
    printf ("trying to connect to address %s, port %s\n", prt, port);
    if (connect (sockfd, addrs->ai_addr, addrs->ai_addrlen) != 0) {
      perror ("connect");
      next_loop (addrs, sockfd);
    }
    printf ("connected to %s\n", prt);
    char * request = build_request (hostname);
    if (request == NULL) {
      printf ("memory allocation (malloc) failed\n");
      next_loop (addrs, sockfd);
    }
    if (send (sockfd, request, strlen (request), 0) != strlen (request)) {
      perror ("send");
      next_loop (addrs, sockfd);
    }
    free (request);  /* return the malloc'd memory */
    /* sometimes causes problems, and not needed
    shutdown (sockfd, SHUT_WR);  */
    int count = 0;
    while (1) {
      /* use BUFSIZE - 1 to leave room for a null character */
      int rcvd = recv (sockfd, buf, BUFSIZE - 1, 0);
      count++;
      if (rcvd <= 0)
        break;
      buf [rcvd] = '\0';
      printf ("%s", buf);
    }
    printf ("data was received in %d recv calls\n", count);
    next_loop (addrs, sockfd);
  }
  freeaddrinfo (original_addrs);
  return 0;
}
```

Listing 4: Cliente HTTP em C.

1. Anote as URLs das páginas que você escolheu e o número de requisições de chamada para obter a página principal de cada página web.

Se você executar este exercício corretamente, deverá ver um cabeçalho de resposta HTTP enviado pelo servidor, assim como o seu conteúdo. Você precisará dessa informação para a parte 2.5. Algumas páginas enviam devolta uma grande quantidade de dados, você precisará rolar a barra para trás ou procurar uma página web que envie menos dados.

3.2 Modificando o Cliente

Modifique o cliente da parte 3.1 para que ele possa requisitar uma URL arbitrária. Você precisará usar operações de *string* em C para inserir parte do caminho (*path*) da URL no restante da requisição. Para analisar a URL você poderá usar funções como `index`, `rindex`, `strstr`, e outras similares.

Recomendo fortemente o uso de `strncat` e `strncpy` ao invés de `strcat` ou `strcpy`. Como alternativa, sinta-se livre para usar o `snprintf` como pode ser visto no código.

Você precisará escrever algum código que separe a URL de coisas como:

```
http://prof.facom.ufms.br/~brivaldo/pages/project/
```

em seus componentes: o endereço do host é `prof.facom.ufms.br` e o caminho requisitado é `/~brivaldo/pages/project/`.

2. Modifique e envie seu código. Se seu código funcionar, coloque no seu relatório a quantidade de requisições necessárias para obter: "https://www.ufms.br/cursos/graduacao/".

3.3 Servidor

Execute o servidor HTTP simples 5 na sua máquina local e conecte nele usando vários clientes web, incluindo seu próprio código anteior e pelo menos um navegador de Internet normal. Se você estiver executando seu servidor na porta X (por exemplo, na porta 6789), você pode usar a URL `http://localhost:6789` para conectar no servidor usando a mesma máquina.

Agora modifique o servidor para imprimir cada requisição, assim você será capaz de ver que requisições seus clientes estão enviando. **Lembre-se que os dados que você recebe da rede NÃO são finalizados em nulo, ou seja, NÃO é uma *string* em C**. Para torná-la uma *string* em C, você deve adicionar o caractere nulo '\0' depois de todos os caracteres de dados (isso já pode estár no código, somente faça essa modificação se necessário).

3. Informe o nome do cliente web que você usou para conectar no seu servidor web (Safari, Edge, Firefox, etc.) e determine todas as requisições que o navegador fez ao servidor. Pode ser apenas uma requisição ou podem ser várias. Apenas observe cuidadosamente e informe quais foram as requisições.

```c
/* hw2-simple-server.c: program to send a constant HTTP 404/501 response. */
/* compile with: gcc -Wall -o hw2-simple-server hw2-simple-server.c */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <time.h>
#include <arpa/inet.h>
```

```c
/* maximum size of a printed address -- if not defined, we define it here */
#ifndef INET6_ADDRSTRLEN
#define INET6_ADDRSTRLEN        46
#endif /* INET6_ADDRSTRLEN */

#define BUFSIZE 1000

/* print a system error and exit the program */
static void error (char * s)
{
  perror (s);
  exit (1);
}

/* terminates the buffer with a null character and: */
/* returns 0 if the first line has not yet been read */
/* returns -1 if the first line fills (overflows) the buffer */
/* returns 1 if the first line has been read and starts with GET */
/* returns 2 if the first line has been read and starts with something else */
static int parse_request (char * buf, int len, int maxlen)
{
  if (len >= maxlen)    /* overflow */
    return -1;
  buf [len] = '\0';     /* terminate, i.e. make it into a C string */
  if (index (buf, '\n') == NULL)    /* not finished reading the first line */
    return 0;
  if (strncmp (buf, "GET", 3) == 0)
    return 1;
  return 2;
}


/* must be executed inline, so must be defined as a macro */
#define next_loop(s) { if (s >= 0) close (s); continue; }

int main (int argc, char ** argv)
{
  int port = 8080;  /* can be used without root privileges */
  int server_socket = socket (AF_INET, SOCK_STREAM, 0);
  if (server_socket < 0)
    error ("socket");

  struct sockaddr_in sin;
  sin.sin_family = AF_INET;
  sin.sin_port = htons (port);
  sin.sin_addr.s_addr = INADDR_ANY;
  struct sockaddr * sap = (struct sockaddr *) (&sin);
  printf ("starting HTTP server on port %d\n", port);
  if (bind (server_socket, sap, sizeof (sin)) != 0)
    error ("bind");

  listen (server_socket, 10);

  while (1) {   /* infinite server loop */
```

```
    struct sockaddr_storage from;
    struct sockaddr_in * from_sinp = (struct sockaddr_in *) (&from);
    struct sockaddr    * from_sap  = (struct sockaddr    *) (&from);
    socklen_t addrlen = sizeof (from);
    int sockfd = accept (server_socket, from_sap, &addrlen);
    if (sockfd < 0)
      error ("accept");
    if (from_sap->sa_family != AF_INET) {
      printf ("accepted connection in address family %d, only %d supported\n",
              from_sap->sa_family, AF_INET);
      next_loop (sockfd);
    }
    char prt [INET6_ADDRSTRLEN] = "unable to print";
    inet_ntop (AF_INET, &(from_sinp->sin_addr), prt, sizeof (prt));
    printf ("accepted a connection from %s\n", prt);

    char buf [BUFSIZE];
    int received = 0;
    while (parse_request (buf, received, sizeof (buf)) == 0) {
      int r = recv (sockfd, buf + received, sizeof (buf) - received, 0);
      if (r <= 0) {
        printf ("received %d\n", r);
        next_loop (sockfd);
      }
      received += r;
    }
    int parse = parse_request (buf, received, sizeof (buf));
    if (parse == -1) {  /* first line longer than buffer */
      printf ("error: first line longer than %ld\n", sizeof (buf));
      next_loop (sockfd);
    }
    char * code = "HTTP/1.0 404 Not Found\r\n";
    if (parse == 2)
      code = "HTTP/1.0 501 Not implemented\r\n";
    send (sockfd, code, strlen (code), 0);

    char date_buf [BUFSIZE];
    time_t now = time (NULL);
    char * time_str = asctime (gmtime (&now));
    snprintf (date_buf, sizeof (date_buf), "Date: %s\n", time_str);
    /* time_str ends with \n.  We replace it with \r to give \r\n */
    * (index (date_buf, '\n')) = '\r';
    send (sockfd, date_buf, strlen (date_buf), 0);

    char server_id [] = "Server: dummy HTTP server\r\n";
    send (sockfd, server_id, strlen (server_id), 0);
    send (sockfd, "\r\n", 2, 0);
    shutdown (sockfd, SHUT_WR);  /* not useful, since we close right away */
    close (sockfd);
  }
  return 0;
}
```

Listing 5: Servidor HTTP em C.

3.4 Modificando o Servidor

Modifique o servidor para retornar o conteúdo de um único arquivo chamado `index.html`. Você deve criar manualmente este arquivo para testar em conjunto com o seu servidor. Veja um exemplo de um código HTML se você nunca viu um antes:

```
<html>
<head><title>Web Page do Meu Servidor Web</title></head>
<body>Esta &eacute; uma p&aacute;gina Web.</body>
</html>
```

O cabeçalho que o seu servidor retorna deve conter pelo menos algum dos cabeçalhos que vimos na parte 3.1, especificamente: `Connection: close`, `Server: X` (sendo X o nome que você escolheu para o servidor), `Content-Type: text/html` e `Content-Length: Y` (sendo Y o número de *bytes* do arquivo `index.html`), que pode ser obtido invocando as funções `stat` ou `fstat`.

4. Envie seu código do servidor modificado.

Documentação da API de *Sockets*

Você poderá encontrar a documentação da API de *sockets* no Unix/Linux nas "man" pages (páginas de manual invocadas pelo comando de terminal `man`). Algumas funções como `snprintf` e `index` estão na seção 3, e o uso de endeços está na seção 7 (IP e IPv6).