```c
/* ServEco
 *
 * MC833 - Programacao em Redes de Computadores
 * Exercicio 7: Servidor de Eco TCP/UDP
 *
 * Autor: Raul Rabelo Carvalho, 105607
 */

#include "myNetworking.h"

int main(int argc, char *argv[])
{
  /* Variaveis para estabelecimento dos sockets: */
  int port;
  int sock_tcp, sock_udp, sock_tcp_c;
  int optval_tcp = 1, optval_udp = 1;
  struct sockaddr_in srv_tcp, srv_udp, cli_tcp, cli_udp;
  unsigned int cli_tcp_sz, cli_udp_sz;

  /* Variaveis para fork do processo: */
  struct sigaction sa;
  pid_t pid;

  /* Variaveis para selecionar TCP ou UDP: */
  fd_set rfds0, rfds1;
  struct timeval tv;
  int maxfd, rval;

  /* Variaveis gerais: */
  bool isClosing;
  char buf_str[BUFSIZE];
  int  buf_len;

/* ------------------------------------------------------------------------ */

  isClosing = FALSE;

  /* Configura o handler para encerrar os procesos-zumbis: */
  signalHandler(&sa);

  /* Configura a porta a ser usada: */
  port = srvArgs(argc, argv);

  /* Cria e configura o socket TCP: */
  sock_tcp = Socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);

  Setsockopt(sock_tcp, SOL_SOCKET, SO_REUSEADDR, &optval_tcp, sizeof(int));

  memset(&srv_tcp, 0, sizeof(srv_tcp));
  srv_tcp.sin_addr.s_addr = INADDR_ANY;
  srv_tcp.sin_port        = htons(port);
  srv_tcp.sin_family      = PF_INET;

  Bind(sock_tcp, (SA *)&srv_tcp, sizeof(srv_tcp));

  Listen(sock_tcp, BACKLOG);

  /* Cria e configura o socket UDP: */
  sock_udp = Socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP);

  Setsockopt(sock_udp, SOL_SOCKET, SO_REUSEADDR, &optval_udp, sizeof(int));

  memset(&srv_udp, 0, sizeof(srv_udp));
  srv_udp.sin_addr.s_addr = INADDR_ANY;
  srv_udp.sin_port        = htons(port);
  srv_udp.sin_family      = PF_INET;

  Bind(sock_udp, (SA *)&srv_udp, sizeof(srv_tcp));
```



```c
  /* Configurando o select para os sockets TCP e UDP: */
  maxfd = (sock_tcp > sock_udp) ? sock_tcp : sock_udp;
  FD_ZERO(&rfds0);
  FD_SET(sock_tcp, &rfds0);
  FD_SET(sock_udp, &rfds0);
  tv.tv_sec = 0;
  tv.tv_usec = 0;

  memset(buf_str, 0, sizeof(buf_str));

  while (1)
  {
    rfds1 = rfds0;

    /* Verifica se ha conexoes TCP ou mensagens UDP: */
    rval = Select(maxfd + 1, &rfds1, NULL, NULL, &tv);

    if (rval > 0 && FD_ISSET(sock_tcp, &rfds1))
    /* Caso TCP: */
    {
      rval--;

      /* Aceita a conexao com o cliente TCP: */
      cli_tcp_sz = sizeof(cli_tcp);
      sock_tcp_c = Accept(sock_tcp, (SA *)&cli_tcp, &cli_tcp_sz);

      /* Cria um processo filho para atender ao cliente TCP: */
      pid = Fork();

      if (pid == 0)
      /* Caso seja o processo-filho: */
      {
        /* Fecha a conexao de escuta: */
        close(sock_tcp);

        /* Loop sobre as mensagens recebidas: */
        while ((buf_len = Recv(sock_tcp_c, buf_str, sizeof(buf_str), 0)) > 0)
        {
          /* Encerra a conexao: */
          if ((isClosing = isExit(buf_str)))
            break;

          /* Envia o eco. */
          Send(sock_tcp_c, buf_str, buf_len, 0);

          memset(buf_str, 0, sizeof(buf_str));
        }

        /* Fecha a conexao com o cliente: */
        close(sock_tcp_c);
        isClosing = TRUE;
      }
      else
      /* Caso seja o processo-pai: */
      {
        /* Encerra a conexao com o cliente TCP: */
        close(sock_tcp_c);
      }
    }

    if (rval > 0 && FD_ISSET(sock_udp, &rfds1))
    /* Caso UDP: */
    {
      rval--;

      /* Recebe mensagem UPD: */
      cli_udp_sz = sizeof(cli_udp);
      buf_len = Recvfrom(sock_udp, buf_str, sizeof(buf_str), 0, (SA *)&cli_udp,
&cli_udp_sz);
```

```
      /* Envia o eco: */
      Sendto(sock_udp, buf_str, strlen(buf_str), 0, (SA *)&cli_udp, cli_udp_sz);

      memset(buf_str, 0, sizeof(buf_str));
    }

    /* Encerra o servidor-filho a pedido do cliente TCP: */
    if (isClosing) break;
  }

  close(sock_udp);

  return 0;
}
```

```c
/* Cliente
 *
 * MC833 - Programacao em Redes de Computadores
 * Exercicio 7: Servidor de Eco TCP/UDP
 *
 * Autor: Raul Rabelo Carvalho, 105607
 */

#include "myNetworking.h"

int main(int argc, char *argv[])
{
  /* Variaveis com os dados do servidor: */
  proto_t proto;
  char *addr;
  int port;

  /* Variaveis para conexao de rede: */
  int sock;
  int optval = 1;
  struct sockaddr_in srv_addr;

  /* Variaveis gerais: */
  char line[MAXLINE];
  char buf_str[BUFSIZE];
  int  buf_len;

/* ------------------------------------------------------------------------ */

  /* Coleta os dados do servidor: */
  proto = cliArgs(&addr, &port, argc, argv);

  /* Cria o socket com o protocolo escolhido: */
  if (proto == TCP)
    sock = Socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
  else if (proto == UDP)
    sock = Socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP);

  /* Configura o socket para reutilizar o mesmo endereco IP: */
  Setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &optval, sizeof(int));

  /* Configura os dados do servidor: */
  memset(&srv_addr, 0, sizeof(srv_addr));
  srv_addr.sin_addr.s_addr = inet_addr(addr);
  srv_addr.sin_port        = htons(port);
  srv_addr.sin_family      = PF_INET;

  /* Conecta ao servidor: */
  Connect(sock, (SA *)&srv_addr, sizeof(srv_addr));

  /* Loop sobre a entrada de mensagens: */
  while (fgets(line, MAXLINE, stdin) != NULL)
  {
    /* Envia a mensagem ao servidor: */
    Send(sock, line, strlen(line), 0);

    if (isExit(line)) break;

    /* Recebe o eco do servidor: */
    buf_len = Recv(sock, buf_str, BUFSIZE, 0);

    /* Imprime o eco: */
    buf_str[buf_len++] = 0;
    printf("%s", buf_str);
  }

  return 0;
}
```

```
/* myNetworking header
 *
 * MC833 - Programacao em Redes de Computadores
 * Exercicio 7: Servidor de Eco TCP/UDP
 *
 * Autor: Raul Rabelo Carvalho, 105607
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <ctype.h>
#include <signal.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/select.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <netdb.h>
#include <poll.h>

/* Tamanho maximo dos buffers: */
#define MAXLINE 4096
#define BUFSIZE 8192

/* Porta de comunicacao padrao: */
#define STD_PORT_NUM  49151
#define STD_PORT_STR  "49151"

#define BACKLOG 1024

#define SA  struct sockaddr

/* Macros para o segundo parametro de Shutdown: */
#define SHUTRECV 0 /* further receives are disallowed */
#define SHUTSEND 1 /* further sends are disallowed */
#define SHUTBOTH 2 /* further sends and receives are disallowed */

#ifndef MYNETWORKING_H_INCLUDED
#define MYNETWORKING_H_INCLUDED

/* Definicao de um tipo booleano: */
typedef enum { FALSE = 0, TRUE = 1 } bool;

/* Definicao de um tipo protocolo: */
typedef enum { TCP = 0, UDP = 1 } proto_t;

/* Pragmas das funcoes auxiliares: */
int srvArgs(int argc, char *argv[]);
proto_t cliArgs(char **addr, int *port, int argc, char *argv[]);
void signalHandler(struct sigaction *sa);
pid_t Fork();
bool isExit(const char *msg);

/* Pragmas das funcoes wrapper de rede: */
int Accept(int fd, struct sockaddr *sa, socklen_t *salenptr);
void Bind(int fd, const struct sockaddr *sa, socklen_t salen);
void Connect(int fd, const struct sockaddr *sa, socklen_t salen);
void Listen(int fd, int backlog);
ssize_t Recv(int fd, void *ptr, size_t nbytes, int flags);
ssize_t Recvfrom(int fd, void *ptr, size_t nbytes, int flags, struct sockaddr *s
a, socklen_t *salenptr);
int Select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struc
t timeval *timeout);
void Send(int fd, const void *ptr, size_t nbytes, int flags);
void Sendto(int fd, const void *ptr, size_t nbytes, int flags, const struct sock
```



```
addr *sa, socklen_t salen);
void Setsockopt(int fd, int level, int optname, const void *optval, socklen_t op
tlen);
int Socket(int family, int type, int protocol);

#endif
```

```c
/* AuxF
 *
 * MC833 - Programacao em Redes de Computadores
 * Exercicio 7: Servidor de Eco TCP/UDP
 *
 * Autor: Raul Rabelo Carvalho, 105607
 */

#include "myNetworking.h"

/* ------------------------------------------------------------------------ */

char *strdup(const char *s)
{
  size_t len = 1 + strlen(s);
  char *p = malloc(len);
  return p ? memcpy(p, s, len) : NULL;
}

/* --- srvArgs() ------------------------------------------------------------
 *
 * desc   : Verifica e processa os argumentos do servidor.
 *
 * params : 1. Numero de argumentos passados.
 *          2. Vetor com os argumentos.
 *
 * output : Inteiro com o numero da porta de conexao do servidor.
 */
int srvArgs(int argc, char *argv[])
{
  int  port;

  if (argc != 1 && argc != 2)
  {
    perror("Error! Usage: 'servidor <PORT>' ou 'servidor'");
    exit(EXIT_FAILURE);
  }

  port = argc > 1 ? atoi(argv[1]) : 0;

  if (1024 > port || port > 49151)
    port = STD_PORT_NUM;

  return port;
}

/* --- cliArgs() ------------------------------------------------------------
 *
 * desc   : Verifica e processa os argumentos do cliente.
 *
 * params : 1. (Saida da funcao.) Ponteiro para string com o endereco IP.
 *          2. (Saida da funcao.) Ponteiro para o inteiro com a porta.
 *          3. Numero de argumentos passados.
 *          4. Vetor com os argumentos.
 *
 * output : Tipo proto_t indica qual protocolo deve ser usado:
 *          TCP equivale ao inteiro 0 e UDP ao inteiro 1.
 */
proto_t cliArgs(char **addr, int *port, int argc, char *argv[])
{
  proto_t proto;
  char *c;

  if (argc != 3 && argc != 4)
  {
    perror("Error! Usage: 'cliente <PROTOCOL> <IP_ADDR> <PORT>'");
    exit(EXIT_FAILURE);
  }
```



```c
  c = argv[1];
  for ( ; *c; ++c) *c = tolower(*c);

  /* Seleciona o protocolo: */
  if (strncmp(argv[1], "tcp", 3) == 0)
    proto = TCP;
  else if (strncmp(argv[1], "udp", 3) == 0)
    proto = UDP;
  else
  {
    perror("Error! The protocol must be either TCP or UDP");
    exit(EXIT_FAILURE);
  }

  /* Retora endereco IP: */
  *addr = strdup(argv[2]);

  /* Seleciona a porta: */
  *port = argc > 3 ? atoi(argv[3]) : 0;
  if (1024 > *port || *port > 49151)
    *port = STD_PORT_NUM;

  return proto;
}

/* --- signalHandler() ------------------------------------------------------
 *
 * desc   :  Configura o processo para ser encerrado caso fique inativo.
 *
 * params : 1. Estrutura na qual se configura como sinais do UNIX serÃ£o
 *             tratados.
 *
 * output : Nenhuma.
 *
 */
void signalHandler(struct sigaction *sa)
{
  sa->sa_handler = SIG_DFL;
  sigemptyset(&sa->sa_mask);
  sa->sa_flags = SA_NOCLDWAIT;
  if (sigaction(SIGCHLD, sa, NULL) == -1) {
    perror("sigaction error");
    exit(1);
  }
}

/* --- Fork() ---------------------------------------------------------------
 *
 * desc   : Cria uma copia do processo corrente.
 *
 * params : Nenhum.
 *
 * output : Identificador do processo-filho.
 */
pid_t Fork()
{
  pid_t pid = fork();
  if (pid < 0) {
    perror("fork");
    exit(EXIT_FAILURE);
  }

  return pid;
}

/* --- isExit() -------------------------------------------------------------
 *
 * desc   : Verifica se uma string contem a palavra 'exit'.
 *
```

```
 * params : 1. String a ser verificada.
 *
 * output : Tipo bool: TRUE ou FALSE.
 */
bool isExit(const char *msg)
{
  bool r = FALSE;
  int len = strlen(msg);

  if (strncmp(msg, "exit\n", len) == 0 ||
      strncmp(msg, "exit\r", len) == 0 ||
      strncmp(msg, "exit\r\n", len) == 0)
    r = TRUE;

  return r;
}
```

```
/*
 *   A implementacao das funcoes wrapper abaixo foram retiradas da pagina web
 * http://www.cs.odu.edu/~cs779/stevens2nd/lib/wrapsock.c e foram alteradas
 * devido as necessidades particulares do Servidor de Eco.
 */

#include  "myNetworking.h"

int
Accept(int fd, struct sockaddr *sa, socklen_t *salenptr)
{
  int    n;

again:
  if ( (n = accept(fd, sa, salenptr)) < 0) {
#ifdef  EPROTO
    if (errno == EPROTO || errno == ECONNABORTED)
#else
    if (errno == ECONNABORTED)
#endif
      goto again;
    else
      perror("accept error");
  }

  return(n);
}

void
Bind(int fd, const struct sockaddr *sa, socklen_t salen)
{
  if (bind(fd, sa, salen) < 0)
    perror("bind error");
}

void
Connect(int fd, const struct sockaddr *sa, socklen_t salen)
{
  if (connect(fd, sa, salen) < 0)
    perror("connect error");
}

void
Getpeername(int fd, struct sockaddr *sa, socklen_t *salenptr)
{
  if (getpeername(fd, sa, salenptr) < 0)
    perror("getpeername error");
}

void
Getsockname(int fd, struct sockaddr *sa, socklen_t *salenptr)
{
  if (getsockname(fd, sa, salenptr) < 0)
    perror("getsockname error");
}

void
Getsockopt(int fd, int level, int optname, void *optval, socklen_t *optlenptr)
{
  if (getsockopt(fd, level, optname, optval, optlenptr) < 0)
    perror("getsockopt error");
}

int
Isfdtype(int fd, int fdtype)
{
  int    n;

  if ( (n = isfdtype(fd, fdtype)) < 0)
```



```
    perror("isfdtype error");
  return(n);
}

/* include Listen */
void
Listen(int fd, int backlog)
{
  char  *ptr;

    /*4can override 2nd argument with environment variable */
  if ( (ptr = getenv("LISTENQ")) != NULL)
    backlog = atoi(ptr);

  if (listen(fd, backlog) < 0)
    perror("listen error");
}
/* end Listen */

int
Poll(struct pollfd *fdarray, unsigned long nfds, int timeout)
{
  int    n;

  if ( (n = poll(fdarray, nfds, timeout)) < 0)
    perror("poll error");

  return(n);
}

ssize_t
Recv(int fd, void *ptr, size_t nbytes, int flags)
{
  ssize_t    n;

  if ( (n = recv(fd, ptr, nbytes, flags)) < 0)
    perror("recv error");
  return(n);
}

ssize_t
Recvfrom(int fd, void *ptr, size_t nbytes, int flags,
     struct sockaddr *sa, socklen_t *salenptr)
{
  ssize_t    n;

  if ( (n = recvfrom(fd, ptr, nbytes, flags, sa, salenptr)) < 0)
    perror("recvfrom error");
  return(n);
}

ssize_t
Recvmsg(int fd, struct msghdr *msg, int flags)
{
  ssize_t    n;

  if ( (n = recvmsg(fd, msg, flags)) < 0)
    perror("recvmsg error");
  return(n);
}

int
Select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
       struct timeval *timeout)
{
  int    n;

  if ( (n = select(nfds, readfds, writefds, exceptfds, timeout)) < 0)
    perror("select error");
```

```
  return(n);    /* can return 0 on timeout */
}

void
Send(int fd, const void *ptr, size_t nbytes, int flags)
{
  if (send(fd, ptr, nbytes, flags) != nbytes)
    perror("send error");
}

void
Sendto(int fd, const void *ptr, size_t nbytes, int flags,
     const struct sockaddr *sa, socklen_t salen)
{
  if (sendto(fd, ptr, nbytes, flags, sa, salen) != nbytes)
    perror("sendto error");
}

void
Sendmsg(int fd, const struct msghdr *msg, int flags)
{
  int      i;
  ssize_t    nbytes;

  nbytes = 0;  /* must first figure out what return value should be */
  for (i = 0; i < msg->msg_iovlen; i++)
    nbytes += msg->msg_iov[i].iov_len;

  if (sendmsg(fd, msg, flags) != nbytes)
    perror("sendmsg error");
}

void
Setsockopt(int fd, int level, int optname, const void *optval, socklen_t optlen)
{
  if (setsockopt(fd, level, optname, optval, optlen) < 0)
    perror("setsockopt error");
}

void
Shutdown(int fd, int how)
{
  if (shutdown(fd, how) < 0)
    perror("shutdown error");
}

int
Sockatmark(int fd)
{
  int    n;

  if ( (n = sockatmark(fd)) < 0)
    perror("sockatmark error");
  return(n);
}

/* include Socket */
int
Socket(int family, int type, int protocol)
{
  int    n;

  if ( (n = socket(family, type, protocol)) < 0)
    perror("socket error");
  return(n);
}
/* end Socket */

void
```



```
Socketpair(int family, int type, int protocol, int *fd)
{
  int    n;

  if ( (n = socketpair(family, type, protocol, fd)) < 0)
    perror("socketpair error");
}
```