

# Programação Orientada a Objetos para Redes de Computadores

Prof. Miguel Elias Mitre Campista

<http://www.gta.ufrj.br/~miguel>

## PARTE 2

### Programação em C++ - Sockets

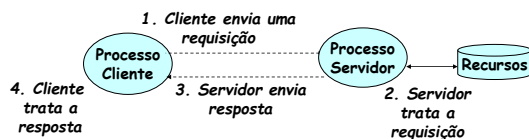
## Comunicação entre Processos

- O que é um processo?
  - Programa em execução em um sistema hospedeiro (*host*)
- Quando dois processos diferentes precisam se comunicar...
  - Se os dois estiverem em execução no mesmo *host*:
    - Eles usam comunicação entre processos definidos pelo próprio sistema operacional - Facilidade por estar na mesma máquina
  - Se os dois estiverem em *hosts* diferentes:
    - Eles usam troca de mensagens pela rede de computadores

## Modelo Cliente-Servidor

- Cliente: **Processo que inicia a comunicação**
  - Solicita um serviço para o servidor
  - Deve sempre saber alguma coisa do servidor
    - Pelo menos onde ele está localizado
- Servidor: **Processo que espera ser contactado**
  - Provê o serviço para o cliente
  - Não precisa saber nada do cliente
    - Nem mesmo se ele existe
- Em situações mais comuns...
  - Há um único servidor para múltiplos clientes

## Modelo Cliente-Servidor



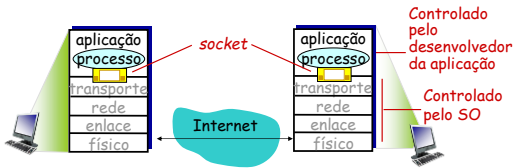
Clientes e servidores são processos em execução no mesmo sistema hospedeiro ou em sistemas hospedeiros diferentes

## O que é um Socket?

- Segundo a visão do sistema operacional...
  - Os sockets são os pontos-finais de uma comunicação
- Segundo a visão da aplicação...
  - Os sockets são uma forma de falar com outros programas usando descritores de arquivos
- Padronização de sockets através de interfaces de programação (APIs)
  - Permite que haja comunicações entre processos diferentes executados em sistemas hospedeiros diversos

## O que é um Socket?

- Pode ser visto como a "porta de uma residência"
  - Processos empurram mensagens pela porta
  - Processos confiam na infraestrutura de transporte para entregar as suas mensagens até a porta do destino



POO para Redes de Computadores - COPPE-PEE/UFRJ

Prof. Miguel Campista

## O que é um Socket?

- No UNIX, um socket é um arquivo identificado através de um descritor (p. ex., um número inteiro)
  - O UNIX é organizado em diretórios e arquivos
    - O que não é diretório, é arquivo (caso do socket)
  - Uso do socket se resume a escrever e ler em arquivos
    - Funções do tipo `read()` e `write()` poderiam ser usadas no lugar das funções `send()` e `recv()`

**Funções `send()` e `recv()` oferecem maior controle na transmissão de dados e por isso são usadas**

POO para Redes de Computadores - COPPE-PEE/UFRJ

Prof. Miguel Campista

## Tipos de Sockets

- Existem vários tipos de sockets
  - Escolha depende dos requisitos das aplicações
    - Integridade dos dados
    - Vazão
    - Atraso
    - Etc.
- Baseado nos requisitos, pode-se escolher o tipo de socket a ser usado
  - O tipo do socket define o protocolo da camada de transporte e, conseqüentemente, os serviços esperados da rede

POO para Redes de Computadores - COPPE-PEE/UFRJ

Prof. Miguel Campista

## Tipos de Sockets

- Socket de fluxo contínuo (*Stream sockets*)
  - Usado em comunicações que exijam conexão confiável bidirecional
    - Se o servidor envia um fluxo "1,2 e 3", o cliente deve receber "1, 2 e 3" em ordem e sem perdas
      - Tais características são oferecidas pelo TCP
- Socket de datagrama (*Datagram sockets*)
  - Usado em comunicações que exijam sobretudo alta vazão - Não há conexão e nem requisitos de confiabilidade
    - Tais características são oferecidas pelo UDP

POO para Redes de Computadores - COPPE-PEE/UFRJ

Prof. Miguel Campista

## Endereço de um Socket

- Recepção de mensagens
  - Todos os processos devem possuir um identificador
- Dispositivo na máquina hospede tem um endereço IP único de 32 bits

**O uso do endereço IP já seria suficiente para identificar o processo?**

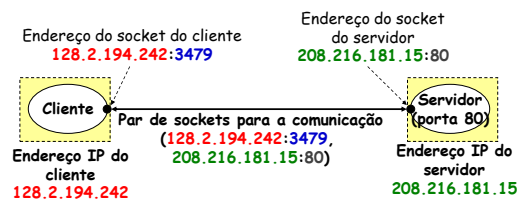
- Não, pois múltiplos processos podem estar em execução na mesma máquina...

POO para Redes de Computadores - COPPE-PEE/UFRJ

Prof. Miguel Campista

## Endereço de um Socket

- Identificador inclui o endereço IP e a porta
  - Par (Endereço IP, porta) forma o "endereço do socket"
  - Exemplo:
    - Servidor HTTP: Endereço IP 128.119.245.12 e porta 80



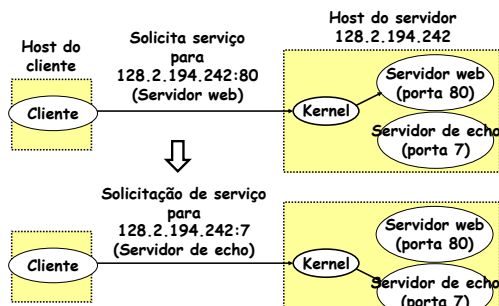
## Socket e Processo Cliente

- Como um cliente encontra um servidor?
  - O endereço IP do endereço do socket do servidor identifica a sistema hospedeiro
  - A número da porta (conhecido) do endereço do socket identifica o serviço e, portanto, implicitamente identifica o processo em execução no servidor
  - Exemplos de número de portas conhecidas
    - Porta 7: Servidor de Echo
    - Porta 23: Servidor Telnet
    - Porta 25: Servidor de e-mail
    - Porta 80: Servidor web

POO para Redes de Computadores - COPPE-PEE/UFRJ

Prof. Miguel Campista

## Socket e Processo Cliente



POO para Redes de Computadores - COPPE-PEE/UFRJ

Prof. Miguel Campista

## Socket e Processo Servidor

- Processos servidores são executados continuamente (*daemons*)
  - Criados durante o boot da máquina pelo processo `init` (primeiro processo a ser executado)
- Cada servidor espera a chegada de requisições em uma porta conhecida associada a um serviço particular
  - Porta 7: Servidor de Echo
  - Porta 23: Servidor Telnet
  - Porta 25: Servidor de e-mail
  - Porta 80: Servidor web
  - Outras aplicações devem usar portas entre 1024 e 65535

POO para Redes de Computadores - COPPE-PEE/UFRJ

Prof. Miguel Campista

## Clichês de Programação em Sockets

- Ordenação do byte de rede (*network byte order*)
  - A ordenação do byte de rede é *big-endian*, enquanto o host pode ser *big* ou *little-endian*
    - **Big-endian**: Byte mais significativo é armazenado primeiro
      - Ex.: Hexa 0xb34f, 0xb3 é armazenado antes do 0x4f
    - **Little-endian**: Byte menos significativo é armazenado primeiro
      - Ex.: Hexa 0xb34f, 0x4f é armazenado antes do 0xb3

POO para Redes de Computadores - COPPE-PEE/UFRJ

Prof. Miguel Campista

## Clichês de Programação em Sockets

- Ordenação do byte de rede (*network byte order*)
  - Formato *big-endian* é mais comum e portanto é o usado em comunicações em rede
    - Teoricamente, o *little-endian* foi proposto posteriormente para facilitar o uso em registradores sequenciais...

Ordem de escrita começa pelo número menos significativo



POO para Redes de Computadores - COPPE-PEE/UFRJ

Prof. Miguel Campista

## Clichês de Programação em Sockets

- Ordenação do byte de rede (*network byte order*)
  - A conversão entre *big-endian* e *little-endian* para ordenação de byte de rede é importante por questões de compatibilidade
  - Funções operam em palavras de 16 bits (*short*) e de 32 bits (*long*)
    - `htons()/htonl()`: Converte ordenação de host para ordenação de rede
    - `ntohs()/ntohl()`: Converte ordenação de rede para ordenação de host
  - Usar essas funções para converter endereços de rede, portas, ...

POO para Redes de Computadores - COPPE-PEE/UFRJ

Prof. Miguel Campista

## Primitivas de Sockets

- **GETADDRINFO**: `int getaddrinfo (const char *node, const char *service, const struct addrinfo *hints, struct addrinfo **res);`
  - *node*: nome ou endereço IP do host que será conectado
  - *service*: nome de um serviço (ex. http) ou número da porta (ex. 80)
  - *hints*: aponta para uma estrutura com informações sobre o tipo de serviço
  - *res*: resultado da execução do `getaddrinfo`
  - *retorna*: sucesso (0) ou falha (diferente de 0)

## Primitivas de Sockets

```
struct addrinfo {
    int         ai_flags;      // AI_PASSIVE, AI_CANONNAME, etc.
    int         ai_family;    // AF_INET, AF_INET6, AF_UNSPEC
    int         ai_socktype;  // SOCK_STREAM, SOCK_DGRAM
    int         ai_protocol;  // use 0 for "any"
    size_t      ai_addrlen;   // size of ai_addr in bytes
    struct sockaddr *ai_addr; // struct sockaddr_in or_in6
    char        *ai_canonname; // full canonical hostname
    struct addrinfo *ai_next; // linked list, next node
};
```

## Primitivas de Sockets

```
/****** SERVER *****/
int status;
struct addrinfo hints;
struct addrinfo *servinfo; // will point to the results

memset(&hints, 0, sizeof(hints)); // make sure the struct is empty
hints.ai_family = AF_UNSPEC; // don't care IPv4 or IPv6
hints.ai_socktype = SOCK_STREAM; // TCP stream sockets
hints.ai_flags = AI_PASSIVE; // fill in my IP for me

if ((status = getaddrinfo(NULL, "3490", &hints, &servinfo)) != 0) {
    fprintf(stderr, "getaddrinfo error: %s\n", gai_strerror(status));
    exit(1);
}
// servinfo now points to a linked list of 1 or more struct addrinfos
// ... do everything until you don't need servinfo anymore ...
freeaddrinfo(servinfo); // free the linked-list
```

## Primitivas de Sockets

```
/****** CLIENT *****/
int status;
struct addrinfo hints;
struct addrinfo *servinfo; // will point to the results

memset(&hints, 0, sizeof(hints)); // make sure the struct is empty
hints.ai_family = AF_UNSPEC; // don't care IPv4 or IPv6
hints.ai_socktype = SOCK_STREAM; // TCP stream sockets

// get ready to connect
status = getaddrinfo("www.example.net", "3490", &hints, &servinfo);
// servinfo now points to a linked list of 1 or more struct addrinfos
// etc.
```

## Primitivas de Sockets

- **SOCKET**: `int socket (int domain, int type, int protocol);`
  - *domain*: `AF_INET` (protocolo IPv4)
  - *type*: (`SOCK_DGRAM` ou `SOCK_STREAM`)
  - *protocol*: (`IPPROTO_UDP` ou `IPPROTO_TCP`)
    - Valor 0 escolhe o protocolo de acordo com o *type*
  - *retorna*: descritor do socket (*sockfd*)
    - -1 é um erro

Usado tanto por cliente quanto por servidor

## Primitivas de Sockets

```
int s;
struct addrinfo hints, *res;

// do the lookup
// [pretend we already filled out the "hints" struct]
getaddrinfo("www.example.com", "http", &hints, &res);

// [again, you should do error-checking on getaddrinfo(), and walk
// the "res" linked list looking for valid entries instead of just
// assuming the first one is good (like many of these examples do.)
// See the section on client/server for real examples.]

s = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
```

## Primitivas de Sockets

- **BIND:** `int bind (int sockfd, struct sockaddr *my_addr, int addrlen);`
  - `sockfd`: descritor do socket (retornado de `socket()`)
  - `my_addr`: socket address, `struct sockaddr_in` é usado
    - Um dos dados da estrutura é o número da porta
  - `addrlen`: `sizeof (struct sockaddr)`
  - `retorno`: sucesso (0) ou falha (-1)

Usado pelo servidor, já que é ele que escuta em uma dada porta conhecida...

## Primitivas de Sockets

- **BIND:** `int bind (int sockfd, struct sockaddr *my_addr, int addrlen);`
  - `sockfd`: descritor do socket (retornado de `socket()`)
  - `my_addr`: socket address ou `struct sockaddr_in` é usado
    - `sockaddr_in` contém a estrutura usada na Internet, mas como a função recebe `sockaddr`, pode-se usar um cast
  - `addrlen`: `sizeof (struct sockaddr)`
  - `retorno`: sucesso (0) ou falha (-1)

```
struct sockaddr_in {
    unsigned short  sin_family; /* address family (always AF_INET) */
    unsigned short  sin_port; /* port num in network byte order */
    struct in_addr  sin_addr; /* IP addr in network byte order */
    unsigned char   sin_zero[8]; /* pad to sizeof(struct sockaddr) */
};
struct in_addr {
    uint32_t s_addr; /* 32-bit int */
};
```

## Primitivas de Sockets

```
struct addrinfo hints, *res;
int sockfd;

// first, load up address structs with getaddrinfo():
memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC; // use IPv4 or IPv6, whichever
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE; // fill in my IP for me

getaddrinfo(NULL, "3490", &hints, &res);

// make a socket:
sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);

// bind it to the port we passed in to getaddrinfo():
bind(sockfd, res->ai_addr, res->ai_addrlen);
```

## Primitivas de Sockets

- **LISTEN:** `int listen (int sockfd, int backlog);`
  - `backlog`: Número de conexões que se quer enfileirar
  - `retorno`: sucesso (0) ou falha (-1)

Usada apenas pelo servidor TCP, já que é ele que escuta e, posteriormente, aceita CONEXÃO. Inicia a máquina de estados do TCP para que seja possível receber até backlog conexões...

## Primitivas de Sockets

- **ACCEPT:** `int accept (int sockfd, void *addr, int *addrlen);`
  - `addr`: Onde o endereço do socket do cliente é escrito
    - Com essas informações pode-se saber quem está chamando (endereço e porta)
  - `addrlen`: Tamanho do endereço do cliente
  - `retorno`: Um novo descritor de socket (para o socket temporário) ou -1 em caso de erro

Usada apenas pelo servidor TCP, já que é ele que escuta e, posteriormente, aceita CONEXÃO...

```
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define MYPOR 3490 // the port users will be connecting to
#define BACKLOG 10 // how many pending connections queue will hold

int main(void) {
    struct sockaddr_storage their_addr;
    socklen_t addr_size;
    struct addrinfo hints, *res;
    int sockfd, new_fd;

    // !! don't forget your error checking for these calls !!

    // first, load up address structs with getaddrinfo():
    memset(&hints, 0, sizeof hints);
    hints.ai_family = AF_UNSPEC; // use IPv4 or IPv6, whichever
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags = AI_PASSIVE; // fill in my IP for me

    getaddrinfo(NULL, MYPOR, &hints, &res);

    // make a socket, bind it, and listen on it:
    sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
    bind(sockfd, res->ai_addr, res->ai_addrlen);
    listen(sockfd, BACKLOG);

    // now accept an incoming connection:
    addr_size = sizeof their_addr;
    new_fd = accept(sockfd, (struct sockaddr *)&their_addr, &addr_size);

    // ready to communicate on socket descriptor: new_fd!
    ...
}
```

## Primitivas de Sockets

- **CONNECT**: `int connect (int sockfd, struct sockaddr *serv_addr, int addrlen);`
  - Os parâmetros são os mesmos do `bind()`
    - Porta e endereço do servidor que se quer conectar

Usada apenas pelo cliente TCP, já que é ele que conecta ao servidor que, por sua vez, aceita a conexão

## Primitivas de Sockets

```
struct addrinfo hints, *res;
int sockfd;

// first, load up address structs with getaddrinfo():

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;

getaddrinfo("www.example.com", "3490", &hints, &res);

// make a socket:

sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);

// connect!

connect(sockfd, res->ai_addr, res->ai_addrlen);
```

## Primitivas de Sockets

- **SEND**: `int send (int sockfd, const void *msg, int len, int flags);`
  - `msg`: Mensagem que se quer enviar
  - `len`: Tamanho da mensagem
  - `flags`: 0
  - `retorno`: Número de bytes que, de fato, foram enviados ou -1 em caso de erro

Usada no socket de fluxo contínuo

## Primitivas de Sockets

- **SEND**: `int send (int sockfd, const void *msg, int len, int flags);`
  - `msg`: Mensagem que se quer enviar
  - `len`: Tamanho da mensagem
  - `flags`: 0
  - `retorno`: Número de bytes que, de fato, foram enviados ou -1 em caso de erro

```
char *msg = "Hello world!";
int len, bytes_sent;
...
len = strlen(msg);
bytes_sent = send(sockfd, msg, len, 0);
...
```

## Primitivas de Sockets

- **RECEIVE**: `int recv(int sockfd, void *buf, int len, unsigned int flags);`
  - `buf`: Buffer para receber a mensagem
  - `len`: Tamanho do buffer
  - `flags`: 0
  - `retorno`: O número de bytes recebidos ou -1 em caso de erro

Usada no socket de fluxo contínuo

## Primitivas de Sockets

- **SEND (estilo DATAGRAMA)**: `int sendto (int sockfd, const void *msg, int len, int flags, const struct sockaddr *to, int tolen);`
  - `msg`: Mensagem que se quer enviar
  - `len`: Tamanho da mensagem
  - `flags`: 0
  - `to`: Endereço do socket do processo remoto
  - `tolen`: = `sizeof(struct sockaddr)`
  - `retorno`: Número de bytes que, de fato, foram enviados

Usada no socket de datagrama

## Primitivas de Sockets

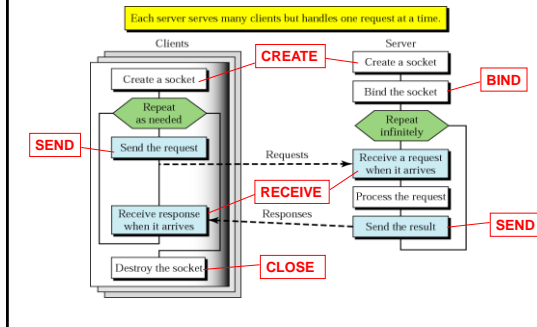
- **RECEIVE (estilo DATAGRAMA):** `int recvfrom (int sockfd, void *buf, int len, unsigned int flags, struct sockaddr *from, int *fromlen);`
  - `buf`: Buffer para receber a mensagem
  - `len`: Tamanho do buffer
  - `from`: Endereço do socket do processo que enviou dados
  - `fromlen`: `sizeof(struct sockaddr)`
  - `flags`: 0
  - **retorno**: O número de bytes recebidos ou -1 em caso de erro

Usada no socket de datagrama

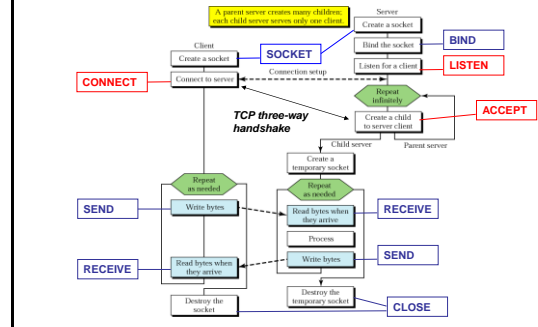
## Primitivas de Sockets

- **CLOSE:** `close (socketfd);`
  - Fecha o socket

## Cliente+Servidor: Sem Conexão



## Cliente+Servidor: Com Conexão



## EchoClient.c

```
#include <stdio.h> /* for printf() and fprintf() */
#include <sys/socket.h> /* for socket(), connect(), send(), and recvfrom() */
#include <arpa/inet.h> /* for sockaddr_in and
                        inet_addr() */
#include <stdlib.h> /* for atoi() and exit() */
#include <string.h> /* for memset() */
#include <unistd.h> /* for close() */

#define ECHOMAX 255 /* Longest string to echo */

int main(int argc, char *argv[])
{
    int sock; /* Socket descriptor */
    struct sockaddr_in echoServAddr; /* Echo server address */
    struct sockaddr_in fromAddr; /* Source address of echo */
    unsigned short echoServPort = 7; /* Echo server port */
    unsigned int fromSize; /* address size for recvfrom() */
    char *servIP="172.24.23.4"; /* IP address of server */
    char *echoString="Does it work?"; /* String to send to echo server */
    char echoBuffer[ECHOMAX+1]; /* Buffer for receiving echoed string */
    int echoStringLen; /* Length of string to echo */
    int respStringLen; /* Length of received response */

```

## EchoClient.c

```
/* Create a datagram/UDP socket */
sock = socket(AF_INET, SOCK_DGRAM, 0);

/* Construct the server address structure */
memset(&echoServAddr, 0, sizeof(echoServAddr)); /* Zero out structure */
echoServAddr.sin_family = AF_INET; /* Internet addr family */
echoServAddr.sin_addr.s_addr = htonl(servIP); /* Server IP address */
echoServAddr.sin_port = htons(echoServPort); /* Server port */

/* Send the string to the server */
sendto(sock, echoString, echoStringLen, 0, (struct sockaddr *) &echoServAddr,
sizeof(echoServAddr));
/* Recv a response */

fromSize = sizeof(fromAddr);
recvfrom(sock, echoBuffer, ECHOMAX, 0, (struct sockaddr *) &fromAddr, &fromSize);
/* Error checks like packet is received from the same server*/

/* null-terminate the received data */
echoBuffer[echoStringLen] = '\0';
printf("Received: %s\n", echoBuffer); /* Print the echoed arg */
close(sock);
exit(0);
} /* end of main () */
```

## EchoServer.c

```
int main(int argc, char *argv[])
{
    int sock; /* Socket */
    struct sockaddr_in echoServAddr; /* Local address */
    struct sockaddr_in echoClntAddr; /* Client address */
    unsigned int cliAddrLen; /* Length of incoming message */
    char echoBuffer[ECRMAX]; /* Buffer for echo string */
    unsigned short echoServPort =7; /* Server port */
    int recvMsgSize; /* Size of received message */
    /* Create socket for sending/receiving datagrams */
    sock = socket(AF_INET, SOCK_DGRAM, 0);
    /* Construct local address structure */
    memset(&echoServAddr, 0, sizeof(echoServAddr)); /* Zero out structure */
    echoServAddr.sin_family = AF_INET; /* Internet address family */
    echoServAddr.sin_addr.s_addr = htonl("172.24.23.4");
    echoServAddr.sin_port = htons(echoServPort); /* Local port */

    /* Bind to the local address */
    bind(sock, (struct sockaddr *) &echoServAddr, sizeof(echoServAddr);
```

## EchoServer.c

```
for (;;) /* Run forever */
{
    cliAddrLen = sizeof(echoClntAddr);

    /* Block until receive message from a client */
    recvMsgSize = recvfrom(sock, echoBuffer, ECRMAX, 0,
        (struct sockaddr *) &echoClntAddr, &cliAddrLen);

    printf("Handling client %s\n", inet_ntoa(echoClntAddr.sin_addr));

    /* Send received datagram back to the client */
    sendto(sock, echoBuffer, recvMsgSize, 0,
        (struct sockaddr *) &echoClntAddr, sizeof(echoClntAddr);
}

} /* end of main () */
```

## Leitura Recomendada

- Brian "Beej Jorgensen" Hall, *"Beej's Guide to Network Programming - Using Internet Sockets"*  
- Acessível em <http://beej.us/guide/bgnet/>