MC833AB – Programação de Redes de Computadores

Professor Nelson Fonseca

http://www.lrc.ic.unicamp.br/mc833/

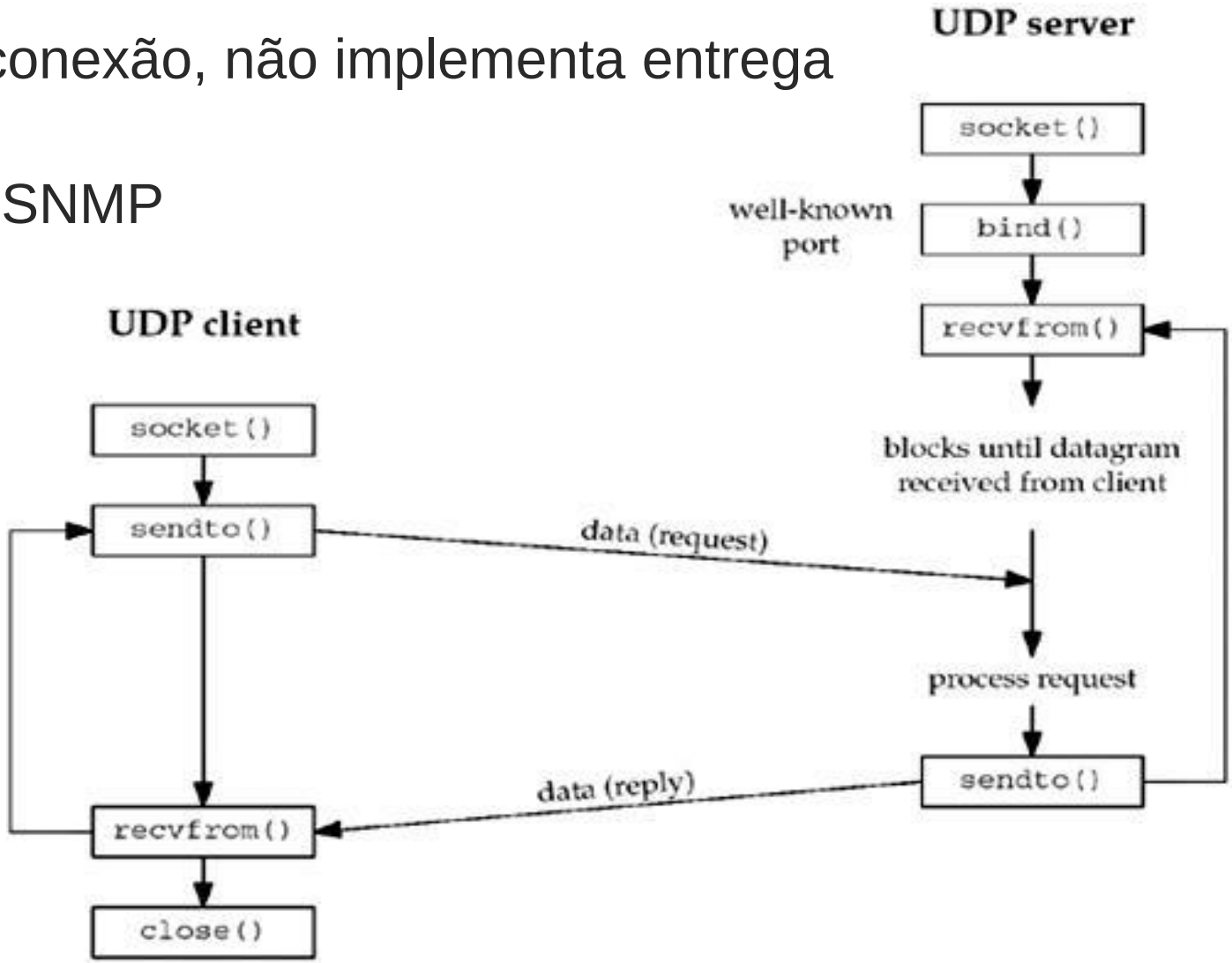
Roteiro

- **Objetivo: explicar como implementar um servidor e um cliente que utilize sockets UDP (Capítulo 8 do livro texto)**

- Sockets UDP
- `recvfrom` e `sendto`
- Erros assíncronos
- `connect` com UDP
- Atividade prática 5

Socket UDP

- UDP: sem conexão, não implementa entrega confiável
- DNS, NFS, SNMP



recvfrom e sendto

```
#include <sys/socket.h>

ssize_t recvfrom(int sockfd, void *buff, size_t nbytes, int
    flags, struct sockaddr *from, socklen_t *addrlen);

ssize_t sendto(int sockfd, const void *buff, size_t nbytes,
    int flags, const struct sockaddr *to, socklen_t addrlen);

  Ambos retornam: número de bytes lidos ou escritos se OK, -1
                                                    p/ erro
```

- **sockfd**: descritor, **buff**: buffer de leitura/escrita, **nbytes**: quantidade de bytes leitura/escrita, **flags**: por enquanto = 0, **to/from**: estrutura do endereço, **addrlen**: tamanho da estrutura com o endereço

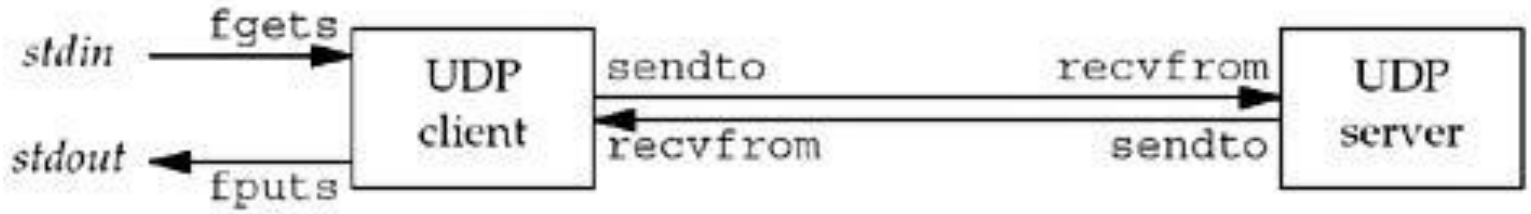
recvfrom e sendto

- Escrever datagrama com tamanho zero, ou receber, é válido (Só terá 20 bytes do cabeçalho IPv4 + 8 bytes do cabeçalho UDP)

  Útil se quiser somente avisar que está "vivo"

- **from** e **addrlen** podem ser ambos nulos – se não importa quem enviou
- **recvfrom** e **sendto** podem ser usados por TCP – exemplo T / TCP – TCP for Transaction

Servidor de echo UDP



- Não tem mais `listen`
- Não tem mais `accept`
- Muda a chamada do socket:

```
sockfd = Socket(AF_INET, SOCK_DGRAM, 0);
```

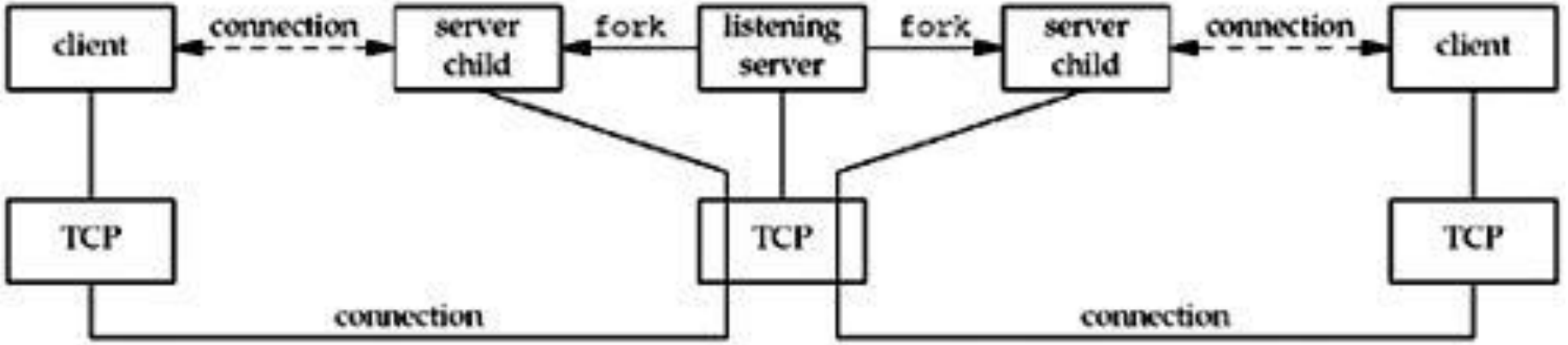
Servidor de echo UDP

- Leitura e escrita agora usam `recvfrom` e `sendto` (ao invés do `read` e `write` do servidor TCP)

```
n = Recvfrom(sockfd, mesg, MAXLINE, 0, pcliaddr, &len);
Sendto(sockfd, mesg, n, 0, pcliaddr, len);
```

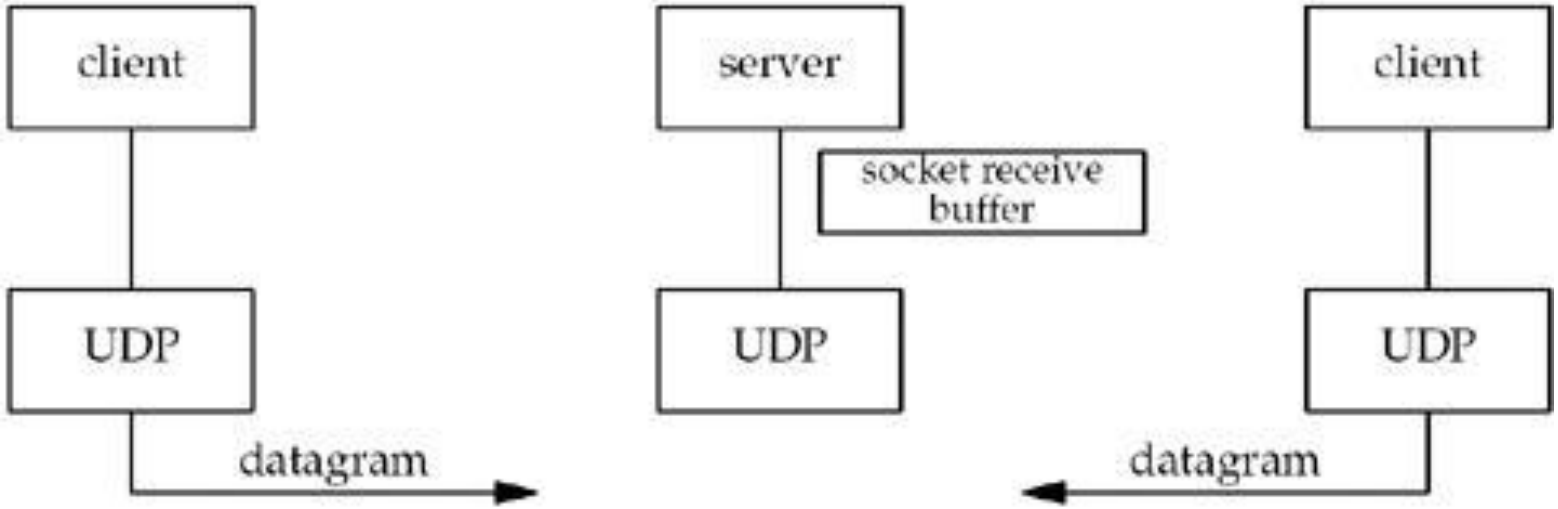
- Não há fim de arquivo

Servidor TCP “é” concorrente



- Um buffer por socket conectado

Servidor UDP é iterativo



- Apenas um processo servidor e único socket
- Um único buffer no qual todos os datagramas chegam

Cliente de echo UDP

- Não precisa de `connect`
- Muda chamada ao socket

```
sockfd = Socket(AF_INET, SOCK_DGRAM, 0);
```

- Substitui `write` e `read` por `sendto` e `recvfrom`
- A escolha da porta efêmera é feita na primeira chamada ao `sendto` (caso não haja `connect`)

UDP não é confiável!

- Se datagrama ou ACK (em nível de aplicação) se perdem, cliente-servidor ficam bloqueados para sempre (se o servidor morre o cliente nunca percebe)
  - Pode colocar um temporizador para evitar isso
  - Mas não é suficiente (Precisa saber se o problema foi no envio ou na recepção)
- Qualquer processo pode enviar datagramas para a porta efêmera do cliente! (Não precisa de `listen`, então o cliente acaba como um servidor)
  - **`recvfrom`** retorna IP de quem enviou. Assim tem que comparar se foi o servidor mesmo e ignorar datagramas "maliciosos" (cuidado com servidores multi-home)
  - Importante especificar porta do servidor e alocar estrutura para salvar info sobre endereço IP

Servidor parado

- Cliente executa mas servidor não. Saída do `tcpdump`:

```
1 0.0 arp who-has freebsd4 tell macosx
2 0.003576 ( 0.0036) arp reply freebsd4 is-at 0:40:5:42:d6:de
3 0.003601 ( 0.0000) macosx.51139 > freebsd4.9877: udp 13
4 0.009781 ( 0.0062) freebsd4 > macosx: icmp: freebsd4 udp
    port    9877 unreachable
```

- `sendto` – retorna sucesso se houve espaço no buffer de envio para datagrama. Então, cliente não “vê” o erro

Erro assíncrono

- Erro ICMP ocorre depois da chamada da função de envio
- Não há como saber endereço IP e porta do servidor que não respondeu quando um único socket é usado para envio de datagrama para vários servidores

  Erro não é retornado para o processo
- `connect` socket UDP recebe de exatamente um peer. Soluciona o problema

  Erro retornado para o processo

`connect` com UDP

- Não há three-way handshaking – estabelecimento da conexão
- O Kernel guarda o endereço IP e a porta do peer, armazenados na estrutura passada na chamada para **`connect`**
- Deve-se usar **`write`** ou **`send`** ao invés de **`sendto`**
  - Com isso: Sockets UDP conectados X Sockets UDP não conectados

`connect` com UDP

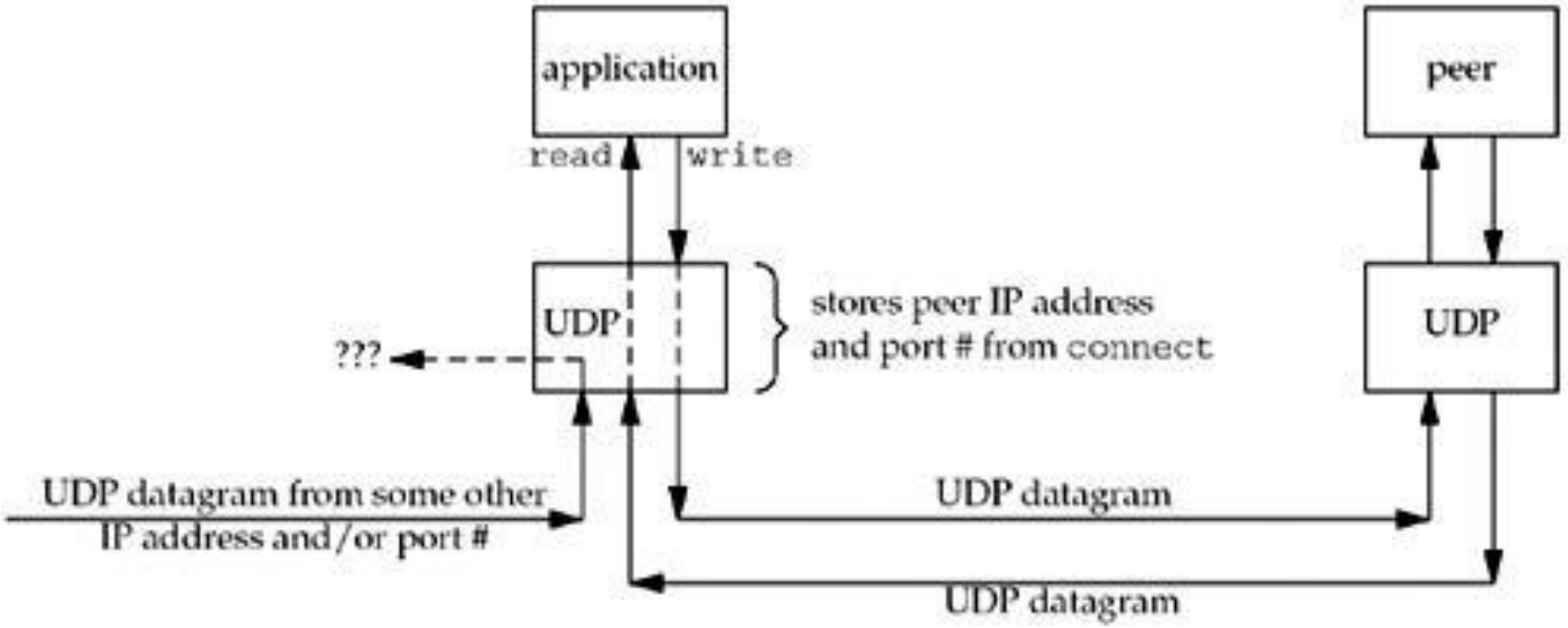
- Até pode usar **`sendto`** mas o ponteiro para a estrutura (quinto argumento) deve ser nulo bem como o tamanho da estrutura se sockets (sexto argumento)
- Não se usa **`recvfrom`**. Usa-se **`read`** ou **`recv`**.
- Datagramas recebidos com endereço IP e porta diferentes do especificado em **`connect`** não são repassados

  Troca de informação com um peer somente
- Erros assíncronos são retornados para processo **`connect`** em socket UDP

`connect` com UDP

Type of socket	`write` or `send`	`sendto` that does not specify a destination	`sendto` that specifies a destination
TCP socket	OK	OK	`EISCONN`
UDP socket, connected	OK	OK	`EISCONN`
UDP socket, unconnected	`EDESTADDRREQ`	`EDESTADDRREQ`	OK

`connect` com UDP



`connect` com UDP

- Um processo pode chamar **`connect`** várias vezes para:
  - Especificar outro endereço IP e porta
  - Desconectar socket
  - Para desconectar chama-se **`connect`** com parâmetro **`sin_family`** = AF_UNSPEC

    Não é padronizado nos SOs

Melhora o desempenho do cliente

Controle de fluxo com UDP

- Em nível de aplicação
- Atribuir valores para o comprimento de cada datagrama (terceiro argumento do `sendto`) e o número de dagramas para enviar (um laço para enviar todos)
- Quando usar o `recvfrom`, contar os datagramas recebidos

Controle de fluxo com UDP

- Pode confirmar a quantidade de datagramas com o `netstat`

```
netstat -s --udp
Udp:
    911230 packets received
    7010 packets to unknown port received.
    0 packet receive errors
    916931 packets sent
```

  - Pode usar para verificar se está tudo ok com o servidor e o cliente (roda antes e roda depois, desde que não haja outras aplicações UDP em execução)
  - Pode modificar o tamanho dos buffers usando opções do socket também