INF01151 – Sistemas Operacionais II

Aula 12 - Estudo de caso: sockets API

Prof. Alberto Egon Schaeffer Filho

Introdução

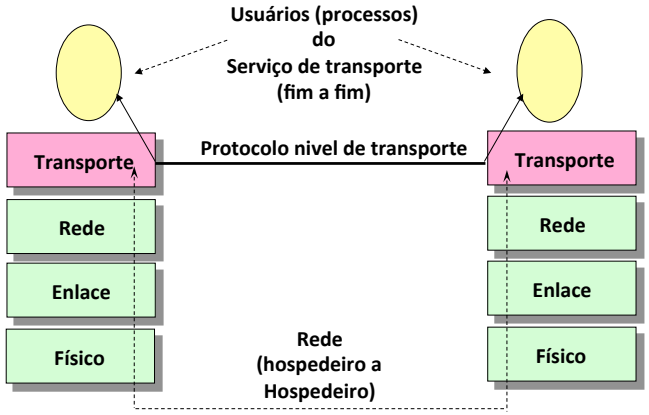
- Mecanismo de comunicação **bidirecional** entre processos por troca de mensagens
  - Entre **aplicações** (processos não-relacionados) e **diferentes máquinas**
  - Baseado nos protocolos de transporte da Internet (UDP e TCP)
- Interface de programação (API) para ambientes de redes
  - Independente de rede específica, mas **IP é o uso mais popular** de sockets
  - Conjunto de funções implementadas por uma biblioteca
  - Características desejáveis:
    - Genérica/uniforme
    - Independente de sistema operacional
    - Suporte a comunicações orientadas a mensagens e a conexão (flexibilidade)
    - Representação externa de dados (inteiros, strings, *byte order* etc.)
- Aplicações em rede seguindo modelo cliente/servidor

Mas antes: uma "revisão rápida" sobre redes de computadores...

- Camada de rede na Internet
  - Endereço IP
    - Identificador de máquina
    - Embute informações de roteamento
- Camada de transporte na Internet
  - Comunicação entre *endpoints* (processo a processo)
    - Processos são identificados por portas
  - Serviços:
    - Orientado a conexão (Transmission Control Protocol – TCP)
    - Não orientado a conexão (User Datagram Protocol – UDP)

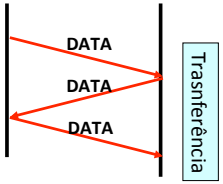
Socket provê abstração para endpoint

Contexto dos protocolos das camadas de rede e de transporte



User Datagram Protocol (UDP)

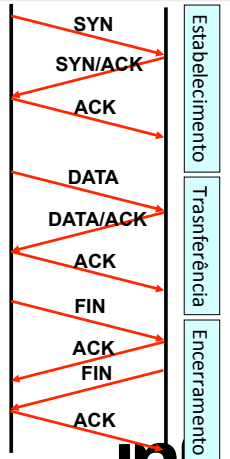
- Não orientado a conexão
- Noção de mensagem (*datagram*)
  - Dados são delimitados na T-PDU
- Modelo de falhas:
  - **Validade**: falhas por omissão, isso é, não entrega
  - **Integridade**: pode haver perdas, erros de ordenamento e duplicação



PDU = *Protocol Data Unit* (equivale a um "pacote" com cabeçalho e dados)
T = camada de transporte

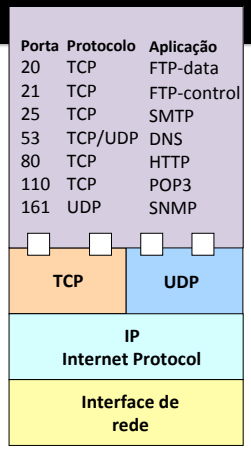
Transmission Control Protocol (TCP)

- Orientado a conexão
- Noção de fluxo de dados (*byte stream*)
  - Dados não são delimitados pela T-PDU
- Modelo falhas:
  - **Validade**: garante a entrega das mensagens
  - **Integridade**: garante ordenamento e não duplicação de mensagens
- Conexão fornece garantia de entrega, não duplicação e ordem



O conceito de porta

- Número de 16 bits usado como identificador
- Três tipos de portas
  - Bem conhecidas (*well-know ports*): 1 a 1023
  - *Registered ports*: 1024 a 49151
  - *Dynamics and/or private ports*: 49152 a 65535
- Uma aplicação é **completamente identificada** por:
  - **End. IP + Porta + Protocolo** (TCP ou UDP)
  - Uma porta não pode ser compartilhada por vários processos (exceção: *multicast* IP)
  - Portas TCP são independentes de Portas UDP
    - Porta 100 (TCP) ≠ Porta 100 (UDP), mas se convenciona "alocar" as duas simultaneamente para uma mesma aplicação



Modelo cliente-servidor

- Paradigma muito usado na implementação de aplicações em rede
  - Pode ser usado com serviços orientados a conexão ou não
- Modelo assimétrico (diferentes papéis)
  - Cliente envia requisições para o servidor
  - Servidor envia respostas para o cliente
- Necessário identificar
  - **Máquina: endereço IP**
  - **Protocolo: TCP ou UDP**
  - **Processo: Porta**

- **Cliente**
  - Usuário de um serviço provido por um processo (servidor)
  - Em serviço orientado a conexão, cliente solicita abertura da conexão
  - Solicita execução do serviço no servidor e espera uma resposta

Implementação lógica: **tipo de nomeação**?

- **Servidor**
  - Fornece um serviço (aplicação) a outros processos (clientes)
  - Espera passivamente a solicitação de clientes

Arquitetura TCP/IP e programação

- Arquitetura TCP/IP não define uma interface de programação
- Funções necessárias:
  - Especificar ponto de comunicação local e remoto
  - Iniciar uma comunicação, se TCP:
    - Esperar por um pedido de abertura de conexão
    - Encerrar uma conexão de forma negociada (graciosa) ou não
  - Enviar e receber dados
  - Tratamento de erro
- Existem várias formas de se programar usando TCP/IP
  - Sockets (padrão POSIX), TLI, *winsock*, MacTCP etc

Sockets

- Abstração de um ponto de comunicação (*endpoint*)
- Características básicas:
  - Permite comunicação bidirecional
  - Prove suporte a vários tipos de protocolos
  - Confiabilidade da comunicação depende
    - Orientado a conexão (TCP)
    - Não orientado a conexão (UDP)
- API *socket* é genérica contendo
  - Funções para criar, nomear, conectar e destruir *sockets*
  - Esperar pedido de conexão
  - Funções para envio e recepção
  - Configuração, gerenciamento e controle de *sockets*
  - Tratamento de erros

Implementação de sockets no UNIX

- Mecanismos de comunicação bidirecional
- Dois domínios básicos:
  - Protocolos de rede da Internet (IPv4 = AF_INET)
    - Canal de comunicação é definido pela associação (5-tupla)
      - [IP_destino, Porta_destino, IP_fonte, Porta_fonte, Protocolo]
    - Envolve um par de sockets (*endpoint* local e *endpoint* remoto)
  - Protocolo de comunicação local UNIX (AF_UNIX)
    - Permite comunicação com interface sockets em um mesmo host
    - Outro mecanismo de IPC local
    - Vantagem é o desempenho (se comparado com AF_INET)
    - Canal de comunicação é um "nome" no sistema de arquivos
      - Semelhantes a **named pipes**

Interface de sockets

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

- Socket é um descritor de arquivo
  - Paradigma abrir-ler-escrever-fechar
- As primitivas básicas são:
  - `socket( )`
  - `bind( )`
  - `listen( )`
  - `accept( )`
  - `connect( )`
  - `write( )`
  - `sendto( )`
  - `read( )`
  - `recvfrom( )`
  - `close( )`

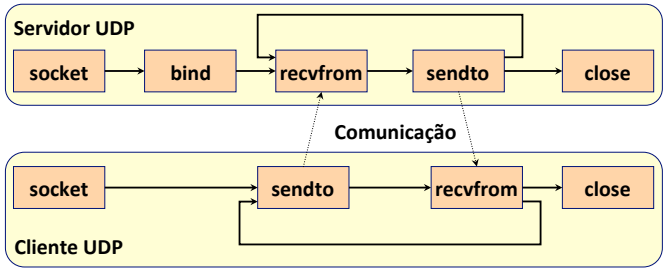
	TCP	UDP
Cliente	socket() connnect() read() write() close()	socket() recvfrom() sendto() close()
Servidor	socket() bind() listen() accept() read() write() close()	socket() bind() recvfrom() sendto() close()

Comunicação UDP (por datagrama)

- Baseado em unidade de transmissão (*datagram*)
  - Tamanho limitado
  - Transmitido sem confirmações ou tentativas de reenvio
    - Mensagens podem não chegar
- Aspectos a serem considerados:
  - Necessidade de especificar o tamanho da área de recepção
    - Se menor que o necessário → mensagem truncada
  - Bloqueio: `send` é não bloqueante e o `receive` é bloqueante
    - Time-out: limitar o tempo de espera do receive configurável via *sockoption*
  - Recepção anônima
- Correspondência entre itens de dados
  - Processos devem "concordar" quanto ao conteúdo dos dados
    - Se um envia *n* bytes como sendo um inteiro, outro deve ler como tal

Como tratar a **representação externa de dados**?

Clientes e servidores UDP

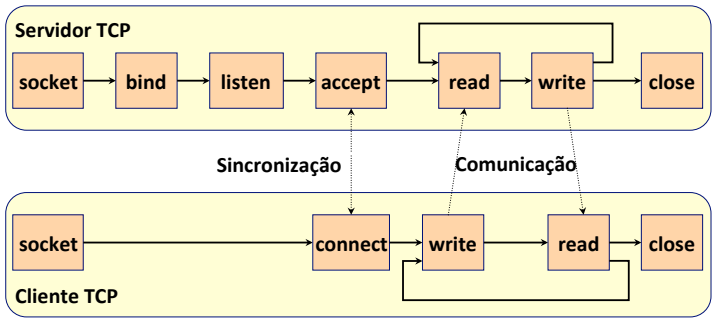


Comunicação TCP (por fluxo)

- Baseado em abstração de fluxo (*stream*)
  - Não define limites de tamanho para as mensagens (tamanho variável)
  - Características de um fluxo TCP
    - Garantia de entrega, entrega ordenada e a não-duplicação
- Aspectos a serem considerados:
  - Bloqueio
    - Processo destino é bloqueado se tenta ler dados não disponíveis
    - Processo remetente é bloqueado pelo controle de fluxo do TCP, se não houver espaço disponível no destinatário para recepção
  - Threads
    - Recomendável para simplificar a programação (bloqueante)
- Correspondência entre itens de dados
  - Processos devem "concordar" quanto ao conteúdo dos dados
    - Se um envia *n* bytes como sendo um inteiro, o outro deve ler como tal

Como tratar a **representação externa de dados**?

Clientes e servidores TCP



socket() – Obtendo o descritor

- Executado por qualquer processo que queira utilizar um socket

```
int socket(int domain, int type, int protocol)
```

  - **domain** especifica família de endereçamento
    - AF_INET (IPv4)
    - AF_INET6 (IPv6)
    - AF_UNIX (canal local)
    - AF_NS (Xerox)
  - **type** (para AF_INET)
    - Não orientado a conexão (UDP): SOCK_DGRAM
    - Orientado a conexão (TCP): SOCK_STREAM
  - **protocol**: protocolo a ser usado para aquele tipo de serviço
    - Caso exista mais de um protocolo oferecendo o mesmo tipo de serviço
      - Valor 0 indica *default*

bind() – Associando endereço/porta

- Executado pelo **servidor** para atribuir uma identidade ao socket
  - Sockets devem funcionar com diversos tipos de comunicação
    - Necessário ser parametrizado com estrutura de endereço

```
int bind(int socket, struct sockaddr *address, socklen_t address_len)
```

  - **socket** especifica o descritor do socket
    - Obtido através da chamada a função `socket()`
  - **address**
    - Struct `sockaddr` é um container genérico
    - Diversas variações dependendo da família
      - para AF_INET `sockaddr_in`; para AF_UNIX `sockaddr_un`
    - Contém elementos pertinentes aquele tipo de comunicação
      - Estrutura deve ser preenchida antes de se chamar o `bind()`
  - **address_len**
    - Necessário especificar tamanho da estrutura

Endereço de um socket AF_UNIX

- Definido em uma estrutura especial (`struct sockaddr_un`)
  - Passado para as chamadas da biblioteca *socket*
  - Família AF_UNIX
  - Pathname: nome do arquivo
- Endereço de um socket no domínio UNIX é string de caracteres
  - Essencialmente é uma entrada no sistema de arquivos

```
struct sockaddr_un {
      sa_family_t   sun_family;    /* AF_UNIX */
      char          sun_path[ ];   /* Pathname */
};
```


Endereço de um socket AF_INET

- Definido em uma estrutura especial (`struct sockaddr_in`)
- Necessário posicionar:
  - Endereços IP (32 bits para o IPv4)
  - Endereço porta (16 bits)

```
struct sockaddr_in {
      uint_t               sin_len;
      sa_family_t          sin_family;          /* AF_INET */
      in_port_t            sin_port;         /* Port number */
      struct in_addr       sin_addr;    /* Internet address */
      char                 sin_zero[8];
};
```

```
struct in_addr {
      in_addr_t     s_addr;
};
```

listen() – Disponível para receber pedidos

- Executado pelo **servidor** para informar que socket pode receber conexões
  - Apenas para serviços **orientados a conexão** (SOCK_STREAM)
- Necessário armazenar pedidos de aberturas de conexão até que eles possam ser atendidos
  - Implica em ter uma fila (e um tamanho) desses pedidos
    - Em caso de overflow, os pedidos de conexão são perdidos

```
int listen(int socket, int backlog)
```

  - **socket**
    - Especifica o descritor do socket
  - **backlog**
    - Define o número máximo de conexões que podem ficar pendentes

accept() – Aceitando conexões

- Executado pelo **servidor** para receber conexão
  - Apenas para serviços **orientados a conexões** (SOCK_STREAM)
    - Recupera o primeiro pedido da fila de requisições (`listen`)
  - Cria novo socket para tratar uma determinada requisição
    - Socket **original** usado **apenas** para **aceitar conexões**, não para troca de dados
  - Default é bloquear se a fila estiver vazia
    - Mas é possível modificar com opção O_NONBLOCK passada via chamada `fcntl()`

```
int accept(int socket, struct sockaddr *address, size_t address_len)
```

  - **socket**
    - Socket disponível para aceitar conexões (listen)
  - **address**
    - Estrutura que é preenchida (automaticamente) com o endereço do cliente
  - **address_len**
    - Tamanho da estrutura de endereçamento

connect() – Requisitando conexões

- Executado pelo **cliente** para solicitar abertura de conexão
  - Apenas para serviços **orientados a conexões** (SOCK_STREAM)
  - Default é **bloquear** até a conexão ser aceita ou expirar timeout
    - Mas é possível modificar comportamento via `fcntl()` com opção O_NONBLOCK

```
int connect(int socket, struct sockaddr *address,size_t address_len)
```

  - **socket** especifica o descritor do socket
    - Obtido através da chamada a função `socket()`
  - **address**
    - Endereço e porta do (socket do) servidor remoto
    - Struct `sockaddr` é um container genérico
    - Diversas variações dependendo da família
      - para AF_INET `sockaddr_in`; para AF_UNIX `sockaddr_un`
  - **address_len**
    - Necessário especificar tamanho da estrutura

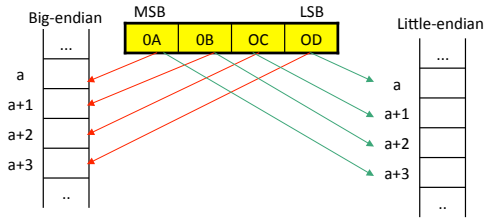
close() – Encerrando um socket

- Terminar uma conexão entre um cliente e um servidor
- Chamada `close()`
  - Similar ao que se faz com um arquivo
  - Deve-se chamar o encerramento em ambos os lados (cliente e servidor) para liberar descritores
- O encerramento pode não ser imediato se ainda houver dados em trânsito

Exemplos de código:

`server_tcp.c`	`server_udp.c`
`client_tcp.c`	`client_udp.c`

Uma armadilha: *network byte order*



- Protocolos Internet empregam *big-endian* (*network order*)
  - Necessário fazer programas que funcionem independente da representação interna da máquina (*host order*)
    - Tipicamente as informações de porta e de endereço IP
  - Funções específicas para auxiliar nesse tratamento

Uma armadilha: *network byte order*

- Os valores a serem armazenados em `sockadd_in` devem ser fornecidos em **network byte order**
  - `sin_port` e `sin_addr`
- Funções específicas para tratar esse aspecto

```
uint16_t        htons(uint16_t);
uint16_t        ntohs(uint16_t);

uint32_t        htonl(uint32_t);
uint32_t        ntohl(uint32_t);
```

h: *host byte order*
n: *network byte order*
s: *short* (16 bits)
l: *long* (32 bits)

Funções de conversão

- Conversão de endereços IP (ASCII) em formato de rede (binário)
- Funções para IPv4
  - `inet_aton`: end. IPv4 (ASCII) para binário
  - `inet_ntoa`: binário para end. IPv4 (ASCII)
  - `inet_addr` (obsoleta): similar a inet_aton
- Funções para IPv4 e IPv6
  - `inet_pton`: presentation to network
  - `inet_ntop`: network to presentation

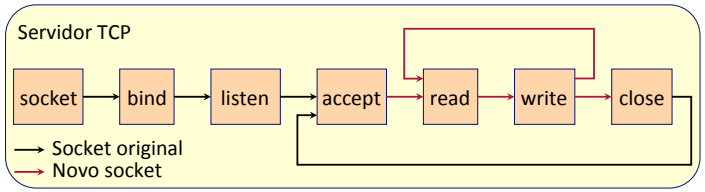
Necessário indicar a família (AF_INET ou AF_INET6)

Servidor iterativo vs. Servidor concorrente

- Em função de como o servidor executa o tratamento de requisições
  - Servidor iterativo (**single-threaded**)
    - Trata requisições de um único cliente a cada instante
    - Implementado como um único processo
  - Servidor concorrente (**multi-threaded**)
    - Trata simultaneamente requisições de vários clientes
    - Implementado com vários processos ou threads independentes
      - Cada thread trata individualmente requisições de um determinado cliente

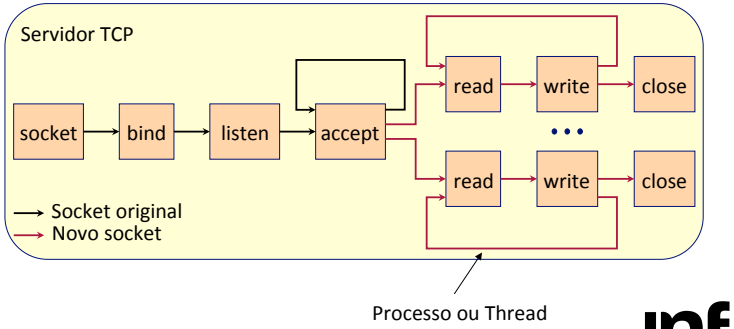
Servidor iterativo

- Adequado para serviços com reduzida taxa de requisições e serviços que possuem baixa carga de processamento



Servidor concorrente

- Adequado para serviços com elevada taxa de requisições e serviços que possuem alta carga de processamento



Leituras adicionais

- Couloris, G.; Dollimore, J.; Kindberg, T. and Blair, G.– *"Distributed Systems: Concepts and Design"* (5th edition), Addison Wesley, 2012
  - Capítulo 4 (seção 4.2)
- Silberschatz, A. and Galvin, P. *"Operating Systems Concepts"*. Wiley, 8th edition, 2009.
  - Capítulo 3 (seção 3.6)
- Beej's Guide to Network Programming - Using Internet Sockets
  - http://beej.us/guide/bgnet/output/html/singlepage/bgnet.html
- Introduction to Sockets Programming
  - http://www.cs.rutgers.edu/~pxk/rutgers/notes/sockets