Sockets: Sumário

- *Sockets* TCP
  - Modêlo.
  - API de Java.
  - API da biblioteca C.
- Avaliação Crítica de *Sockets*.

Resumo das Propriedades de UDP e de TCP

Propriedade	UDP	TCP
Abstracção	Mens.	*Stream*
Baseado em Conexão	N	S
Fiabilidade (perda & duplicação)	N	S
Ordem	N	S
Controlo de Fluxo	N	S
Número de Receptores	n	1

Estabelecimento de Conexão com *Sockets*

- Modêlo assimétrico:
  - Um dos processos é *activo*: toma a iniciativa de estabelecer a conexão com um *socket* remoto;
  - O outro processo é *passivo*: *escuta* num *socket* local, à espera que algum processo tente estabelecer uma conexão:
    * Não conhece *a priori* os processos que estabelecem a conexão.
    * Quando recebe e aceita um pedido, cria um novo *socket* para transferência de dados.
      · O novo *socket* é a extremidade do canal baseado em conexão.

Sockets: Sumário

- *Sockets* TCP
  - Modêlo.
  - **API de Java.**
  - API da biblioteca C.
- Avaliação Crítica de *Sockets*.

Comunicação *Com* Conexão em Java

- Java define 2 classes especificamente para comunicação com conexão:

  `Socket` representa um *socket* TCP usado para transferência de dados.

  `ServerSocket` representa um *socket* TCP usado para receber pedidos de conexão.

- Tipicamente, uma aplicação:
  - Cria um `Socket`/`ServerSocket`.
  - Estabelece uma conexão:
    - O número de invocações de métodos é inferior ao de funções na interface *sockets* BSD.
  - Transfere a informação:
    - Tipicamente, *associando* um *stream* do pacote `java.io` ao *socket*.
  - Termina a conexão.

Classe `ServerSocket`

- Suporta um conjunto de construtores, alguns dos quais permitem:
  - Atribuir um nome ao *socket* TCP criado.
- Suporta operações para:
  - Estabelecer uma conexão,
  - Configurar diferentes parâmetros dos *sockets*, como p.ex.:
    - tamanho de *buffers*;
    - valores de temporização.

Classe `ServerSocket`

- Construtores:

  `ServerSocket(int port)` Cria um *socket* TCP que aceita pedidos de conexão no porto especificado.

  `ServerSocket(int port, int backlog, InetAddress addr)` Cria um *socket* TCP que aceita pedidos de conexão no endereço e porto especificados, até um máximo de pedidos pendentes.

- Métodos:

  `Socket accept()` aceita um pedido de conexão;

  `void close()` fecha o *socket*, passando a rejeitar pedidos de conexão;

  `int getReceiveBufferSize()` retorna o tamanho do *buffer* de recepção associado ao *socket*.

Classe `Socket`

- Construtores, entre outros:

  `Socket()` Cria um *socket* TCP sem estabelecer conexão.

  `Socket(InetAddress addr, int port)` Cria um *socket* TCP para transferências de dados e estabelece conexão com o *socket* cujo nome é passado como argumento.

- Métodos, entre outros:

  `InputStream getInputStream()` obtém *stream* para receber dados;

  `OutputStream getOutputStream()` obtém *stream* para transmitir dados;

  `void close()` fecha o *socket*, passando a rejeitar pedidos de conexão.

  `void shutdownOutput()` fecha lado *stream* de transmissão.

Transferência de Dados Com *Sockets* TCP

- Requer o uso do pacote `java.io`, concebido para E/S em geral, e acesso a ficheiros em particular.
- Os métodos `getInputStream()` e `getOutputStream()`, retornam *streams* de *bytes*, muito simples.
- Pode usar-se outras classes de daquele pacote para obter *streams* com funcionalidade adicional:

Observações	Leitura	Escrita
Classes abstractas	InputStream	OutputStream
Classe útil	InputStreamReader	
Texto formatado	BufferedReader	PrintWriter
Tipos primitivos	DataInputStream	DataOutputStream
Objectos	ObjectInputStream	ObjectOutputStream

Comunicação *Com* Conexão em Java: Exemplo (1/2)

```
import java.net.*;
import java.io.*;

public class EchoServer {
    public static void main(String[] args) throws IOException {
        ServerSocket srvSocket;
        Socket echoSocket = null;
        try {
            srvSocket = new ServerSocket(4445);
        } catch (IOException e) {
            System.out.println("Could not listen on port: 4445");
            System.exit(-1);
        }
        try {
            echoSocket = srvSocket.accept();
        } catch (IOException e) {
            System.err.println("Accept failed: 4445'');
            System.exit(1);
        }
```

Comunicação *Com* Conexão em Java: Exemplo (2/2)

```
        PrintWriter out = null;
        BufferedReader in = null;
        in = new BufferedReader(new InputStreamReader(
                                        echoSocket.getInputStream()));
        out = new PrintWriter(echoSocket.getOutputStream(), true);

        out.println(in.readLine());

        out.close();
        in.close();
        echoSocket.close();
        srvSocket.close();
    }
}
```


Sockets: Sumário

- *Sockets* TCP
  - Modêlo.
  - API de Java.
  - **API da biblioteca C.**
- Avaliação Crítica de *Sockets*.

Estabelecimento de Conexão com *Sockets*

- Modêlo assimétrico:
  - Um dos processos é *activo*: toma a iniciativa de estabelecer a conexão com um *socket* remoto;
  - O outro processo é *passivo*: *escuta* num *socket* local, à espera que algum processo tente estabelecer uma conexão:
    - Não conhece *a priori* os processos que estabelecem a conexão.
    - Quando recebe e aceita um pedido, cria um novo *socket* para transferência de dados.
      - O novo *socket* é a extremidade do canal baseado em conexão.

Passos para Comunicação com *Sockets* TCP

1. Criar um *socket*:
   - tal como em *sockets* UDP;
2. Atribuir-lhe um *nome*:
   - tal como em *sockets* UDP;
3. Estabelecer uma conexão:
   - Do lado passivo, `listen()` primeiro e `accept()` depois;
   - Do lado activo, `connect()`.
4. Transferir informação:
   - `write()/read()` ou `send()/recv()`
5. Fechar a conexão:
   - `close()` ou, de preferência, `shutdown()`.

Estabelecimento de conexões: lado passivo

1. Assinalar a disponibilidade do processo para estabelecer conexões (chamada ao sistema `listen()`):
   - deve ser invocado ”antes” dum processo remoto invocar `connect()`;
   - pedidos “simultâneos” são processados por ordem de chegada;
2. Aceitar pedidos de conexão (chamada ao sistema `accept()`):
   - estabelece a conexão;
   - cria um *socket* que é usado para a comunicação dos dados nessa conexão.

Chamada ao sistema `listen()`

```
int listen(int s, int backlog);
```

onde:

`s`: identificador do *socket* local que fica à escuta de pedidos de conexão - *socket* cujo nome deve ser usado em `connect()`;

`backlog`: comprimento máximo da fila de pedidos de conexão - se a fila estiver cheia quando um pedido chegar, o pedido é ignorado;

Obs: `listen` retorna imediatamente, com o valor 0 (zero) em caso de sucesso e -1, caso contrário.

Chamada ao sistema `accept()`

```
int accept(int s, struct sockaddr *addr,
           socklen_t *addrlen);
```

`s`: identificador do *socket* local que fica à escuta de pedidos de conexão (ver `connect()`);

`addr`: endereço duma `struct sockaddr` que será inializada com o nome do *socket* remoto;

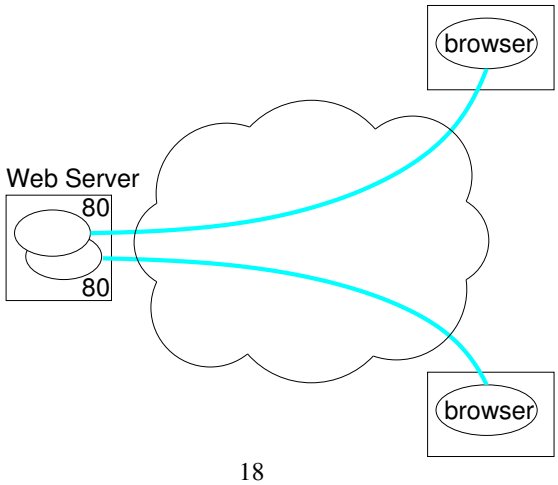
`addrlen`: endereço dum inteiro inicializado com o tamanho de `*addr`. `accept()` altera este valor para o comprimento da estrutura de dados com o nome do *socket* remoto;

Retorna: o identificador do *socket* a usar para transferência de dados:

- num mesmo computador, pode haver vários *sockets* TCP com o mesmo nome – essencial para suportar comunicação baseada em conexão concorrente.

Reuso de Nomes de *Sockets* em TCP)

- Ao contrário do que acontece com UDP, canais TCP no mesmo computador podem ter o mesmo número de porto:
  - Um canal TCP é identificado pelos pares *(IP Address, TCP Port)* das duas extremidades;
  - Permite o atendimento concorrente de vários clientes, em aplicações cliente-servidor, p.ex. *Web*:



Lado activo: chamada ao sistema `connect()`

```
int  connect(int sock, const struct sockaddr *peer_addr,
             socklen_t addrlen);
```

onde:

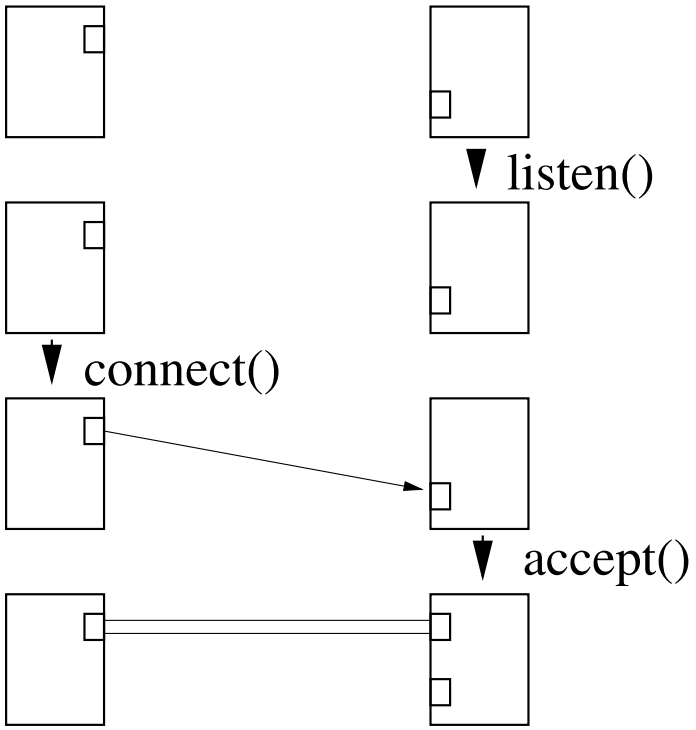
`sock`: identificador do *socket* local;

`peer_addr`: endereço duma `struct sockaddr` com o nome do *socket* remoto;

`addrlen`: é o comprimento da estrutura de dados apontada por `peer_addr`

IMP: o processo que invoca `connect()` bloqueia até que a conexão seja estabelecida.

Estabelecimento de conexões com *sockets*



send()

- `int  send(int s, const void *msg, size_t len, int flags);`

  s: identificador do *socket* local: usado como argumento em `connect()` ou retornado por `accept()`;

  msg: endereço do *buffer* contendo a mensagem a transmitir;

  len: tamanho da mensagem a transmitir;

  flags: *bitmask* especificando diferentes opções;

- `send` retorna o número de *bytes* transmitidos (-1 se ...)
- **IMP-** Se o *socket* usar TCP, o *kernel* pode não transmitir imediatamente a mensagem por razões de eficiência.
- **Obs.-** Não há necessidade de especificar o nome do *socket* remoto: é a outra extremidade da conexão cuja extremidade local é `s`.

recv()

- `int  recv(int s, const void *buf, size_t len, int flags);`

  s: identificador do *socket* local: usado como argumento em `connect()` ou retornado por `accept()`;

  buf: endereço do *buffer* a inicializar com a mensagem recebida;

  len: tamanho em bytes do buffer `buf`;

  flags: *bitmask* especificando diferentes opções;

- `recv` retorna o número de *bytes* recebidos(-1 se ...)
- **IMP-** Se o protocolo especificado fôr TCP, o número de bytes recebidos por um `recv()` não é necessariamente idêntico ao do `send()` correspondente.

Terminação duma conexão: `shutdown()` e `close`

`close()`: termina uma conexão – normalmente retorna imediatamente, mas o *kernel* tenta enviar quaisquer dados que lhe foram previamente passados:

```
int close(int so);
```

`so`: identificador do *socket* local: usado como argumento em `connect()` ou retornado por `accept()`;

`shutdown()`: termina uma conexão parcial ou totalmente:

```
int shutdown(int s, int how);
```

`so`: identificador do *socket* local: usado como argumento em `connect()` ou retornado por `accept()`;

`how`: se 0, inibe recepção; se 1, inibe transmissão; se 2, inibe recepção e transmissão.

IMP.- Em ambos os casos, o *socket* remoto é ”notificado”.

`shutdown()`: terminação ordenada duma conexão

Problema: `close()` fecha a conexão podendo conduzir à perda de informação ainda não recebida;

Solução: usar `shutdown()`:

- se um processo não tem mais informação para transmitir, fecha apenas o lado de transmissão usando `shutdown()`, mas continua a receber até ser notificado que o *socket* remoto foi fechado;
- quando um processo é notificado que o *socket* remoto foi fechado **e** não tem mais informação para transmitir, fecha o seu *socket* (usando `shutdown()` ou `close()`);

Sockets: Sumário

- *Sockets* TCP
  - Modêlo.
  - API de Java.
  - API da biblioteca C.
- **Avaliação Crítica de *Sockets*.**

Programação com *Sockets*: Crítica (1/2)

- Baixo nível de abstracção. Programador tem que lidar com:
  - mensagens;
  - conexões;
  - pormenores de endereços IP e portos:
    - diferentes formatos de representação dos dados.
- A interface *socket* Unix/Linux é pesada de usar devido à flexibilidade excessiva:
  - No início dos anos 80, não era seguro que TCP/IP se tornaria *o protocolo* (aliás, sofreram ainda algumas alterações desde então).
- O desenvolvimento de aplicações distribuídas requer:
  - a especificação, e
  - a implementação

  dum protocolo para comunicação entre processos.
- A programação com *sockets* só permite comunicação se os processos transmissor e receptor existirem simultaneamente.

Programação com *Sockets*: Crítica (2/2)

- De qualquer modo, *sockets* são *inevitáveis*:
  - são a abstracção da rede oferecida pelo SO.
- As outras abstracções para desenvolvimento de aplicações distribuídas, são construídas sobre *sockets*.
- A generalidade dos serviços/protocolos fundamentais da Internet é programado directamente sobre *sockets*:
  - DNS;
  - SMTP (*email*);
  - SNMP (*network management*);
  - NTP (sincronização de relógios);
  - FTP;
  - HTTP.