Comunicação entre Processos – parte 1

Raquel mini - raquelmini@pucminas.br

Sistemas Operacionais

Sistemas Operacionais (SO) de rede

- Possuem recurso de interligação em rede incorporado
- Podem ser usados para acessar recursos remotos
- Os nós em um SO de rede mantêm a autonomia no gerenciamento de seus próprios recursos de processamento
- Um nó não escalona processos em outros nós

Sistemas Operacionais

Sistemas operacionais distribuídos

- Existe uma imagem única do sistema
- O SO teria o controle sobre todos os nós do sistema e dispararia novos processos, de forma transparente, no nó mais conveniente, de acordo com sua política de escalonamento
- Por exemplo, o SO poderia criar um novo processo no nó menos carregado do sistema para evitar sobrecarga em alguns nós
- Produz uma imagem única do sistema para todos os recursos de um sistema distribuído

Sistemas Operacionais

Não existem SO distribuídos para uso geral

- Usuários têm muito investimento feito em software aplicativo que frequentemente atende suas necessidades atuais
- Usuários preferem ter certo grau de autonomia em suas máquinas

Os sistemas distribuídos são construídos com a utilização de *middleware*

- Camada de software que funciona como uma ponte entre o sistemas operacional e as aplicações

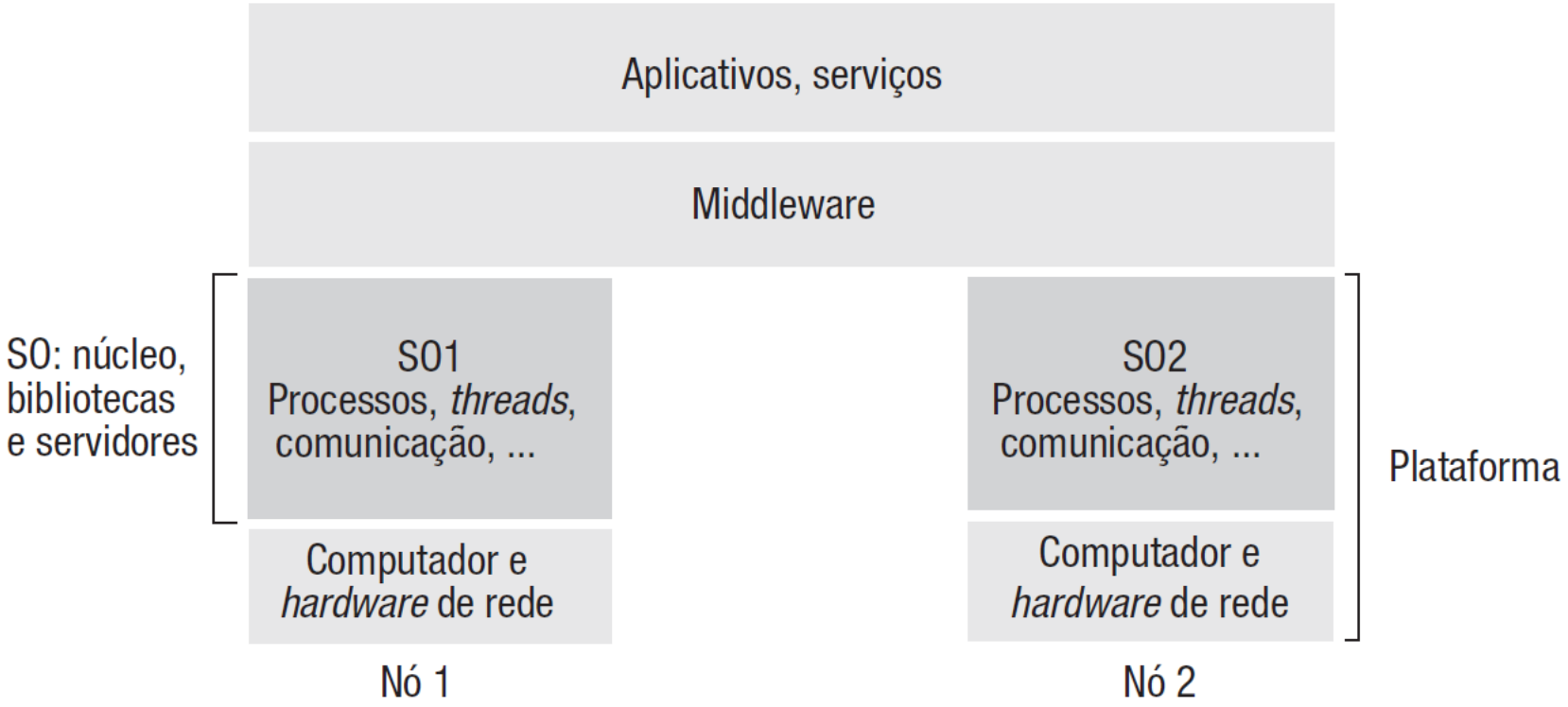
Middleware + SO de Rede

O *middleware* fornece um modelo computacional uniforme para ser usado pelos programadores de serviços e de aplicativos distribuídos

A combinação *middleware* e sistemas operacionais de rede proporciona um equilíbrio aceitável entre os requisitos de **autonomia** e **acesso aos recursos**

Sistemas Distribuídos

O *middleware* fornece comunicação remota entre objetos ou processos nos nós de um sistema distribuído



Comunicação entre Processos

A comunicação entre processos na Internet fornece tanto comunicação por datagrama como por fluxo (*stream*)

Introdução aos Serviços de Rede

Modelo cliente/servidor:

- Servidor:
  - programa que provê um serviço, torna algum recurso disponível a outros programas em qualquer lugar da rede
  - Servidor é passivo
  - Normalmente é inicializado em tempo de *boot* e fica esperando uma requisição de uso dos recurso que ele controla

Introdução aos Serviços de Rede

Modelo cliente/servidor:

- Cliente:
  - Programa que usa o recurso disponibilizado pelo servidor, não importando a localização
  - Procura ativamente na rede onde o recurso está

- Servidores e clientes podem estar na mesma máquina ou em qualquer lugar da rede

Introdução aos Serviços de Rede

Modelo cliente/servidor visa permitir que um determinado recurso seja disponibilizado através da rede a qualquer aplicação que precise

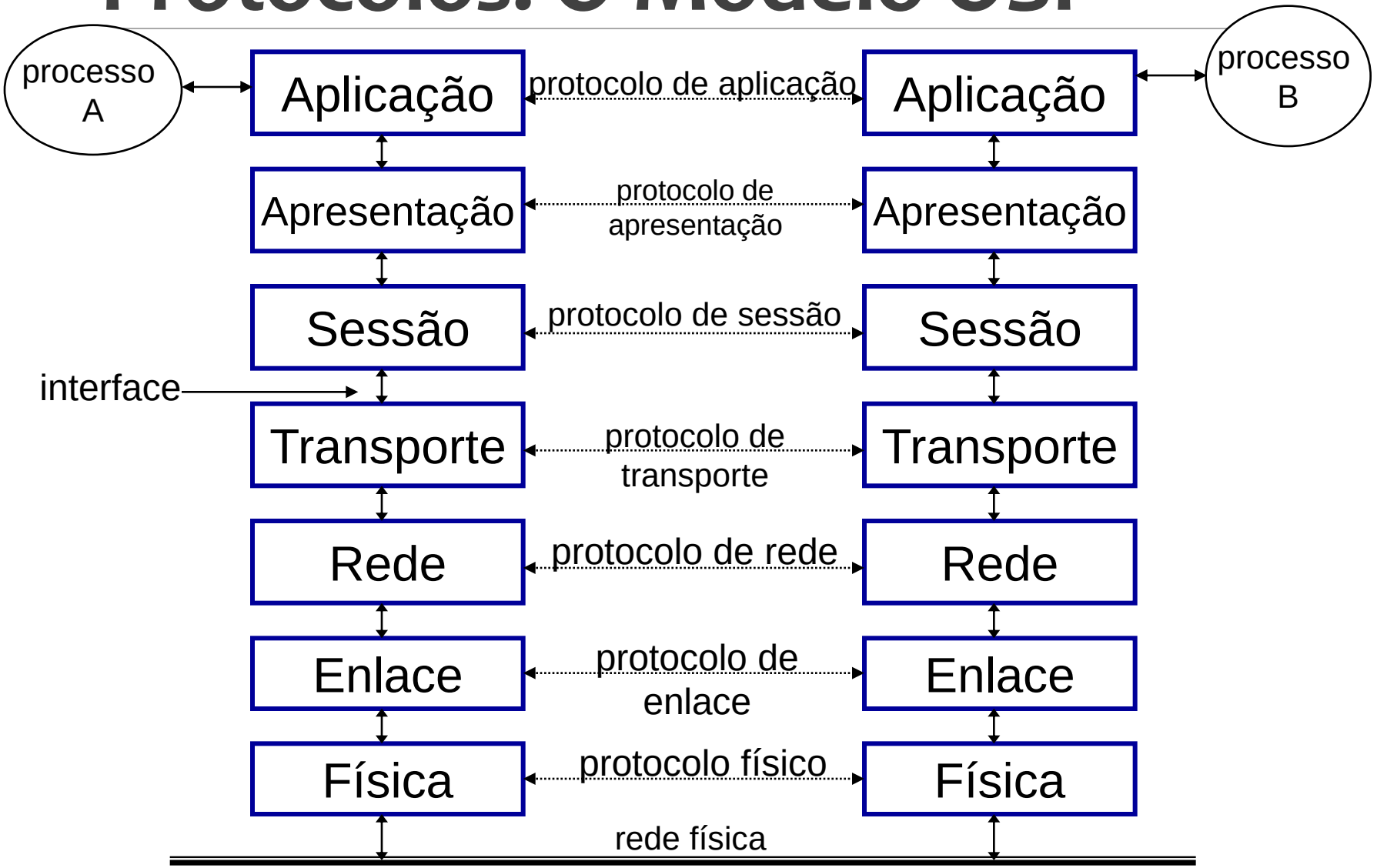
Daemon (lê-se dímon):

- Um servidor permanentemente ativo
- Normalmente possui um endereço padrão conhecido

Protocolo:

- Conjunto de regras descrevendo como um cliente e um servidor interagem
- Define de forma precisa os comandos aceitos pelo servidor e a forma das mensagens usadas na comunicação

Protocolos: O Modelo OSI



Comunicação entre Processos

Comunicação em Sistemas:

- Monolíticos: funções, variáveis globais etc
- Distribuídos: troca de mensagens

Troca de Mensagens:

- Conceito primitivo e de muito baixo nível
- Primitivas *send* e *receive*
- Modalidades de Comunicação:
  - Síncrona
  - Assíncrona

Comunicação Síncrona

Síncrona: *send* e *receive* bloqueantes

- Processo que emitiu *send* permanece bloqueado até que outro processo execute *receive*
- Processo que emitiu *receive* permanece bloqueado até a chegada de uma mensagem

Comunicação Síncrona

Vantagem:

- Simplicidade de programação

Desvantagem:

- Desempenho (CPU fica bloqueada durante transmissão e recepção)
- Possível solução: múltiplas *threads*

Normalmente o *receive* é bloqueante

- É executada por uma *thread* enquanto outras *threads* do processo permanecem ativas

Comunicação Assíncrona

Assíncrona: *send* e *receive* não bloqueantes

- Processo que emitiu *send* prossegue a execução assim que a mensagem é copiada para um *buffer* local
- Processo que emitiu *receive* prossegue a execução mesmo que não exista mensagem
  - *receive* fornece o endereço de um *buffer* que será preenchido em *background*
  - Processo é notificado da chegada de uma mensagem via *polling* ou via interrupção

Comunicação Assíncrona

Vantagem:

- Melhor desempenho (processamento prossegue durante transmissão e recepção)

Desvantagem:

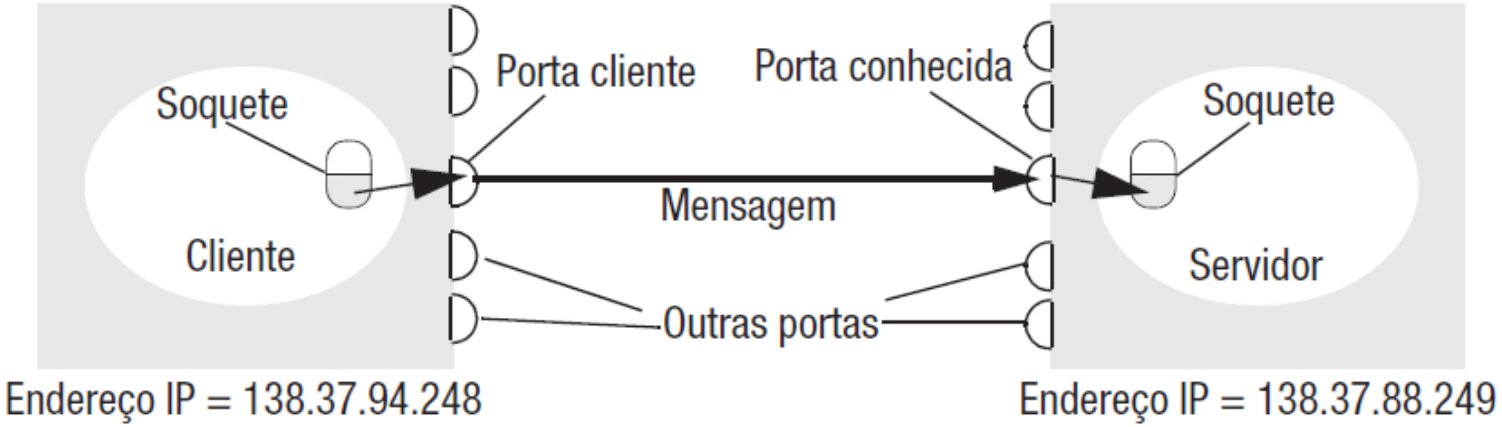
- Tratamento do recebimento de mensagens é mais complexo (ocorre fora do fluxo normal de execução)

Normalmente o *send* é não bloqueante

Comunicação entre Processos

Nos protocolos Internet, as mensagens são enviadas para destinos identificados pelo par (`endereço IP`, `porta local`) chamado soquete

A comunicação entre processos consiste em transmitir uma mensagem entre um soquete de um processo e um soquete de outro processo



Modelo TCP/IP ou UDP/IP

IP

- Protocolo da camada de rede
- Endereça duas máquinas

UDP

- Sem conexões
- Transferência não confiável de dados

TCP

- Orientado à conexões
- Transferência confiável de dados
- Controle de congestionamento
- Controle de fluxo

UDP

Protocolo de transporte não confiável

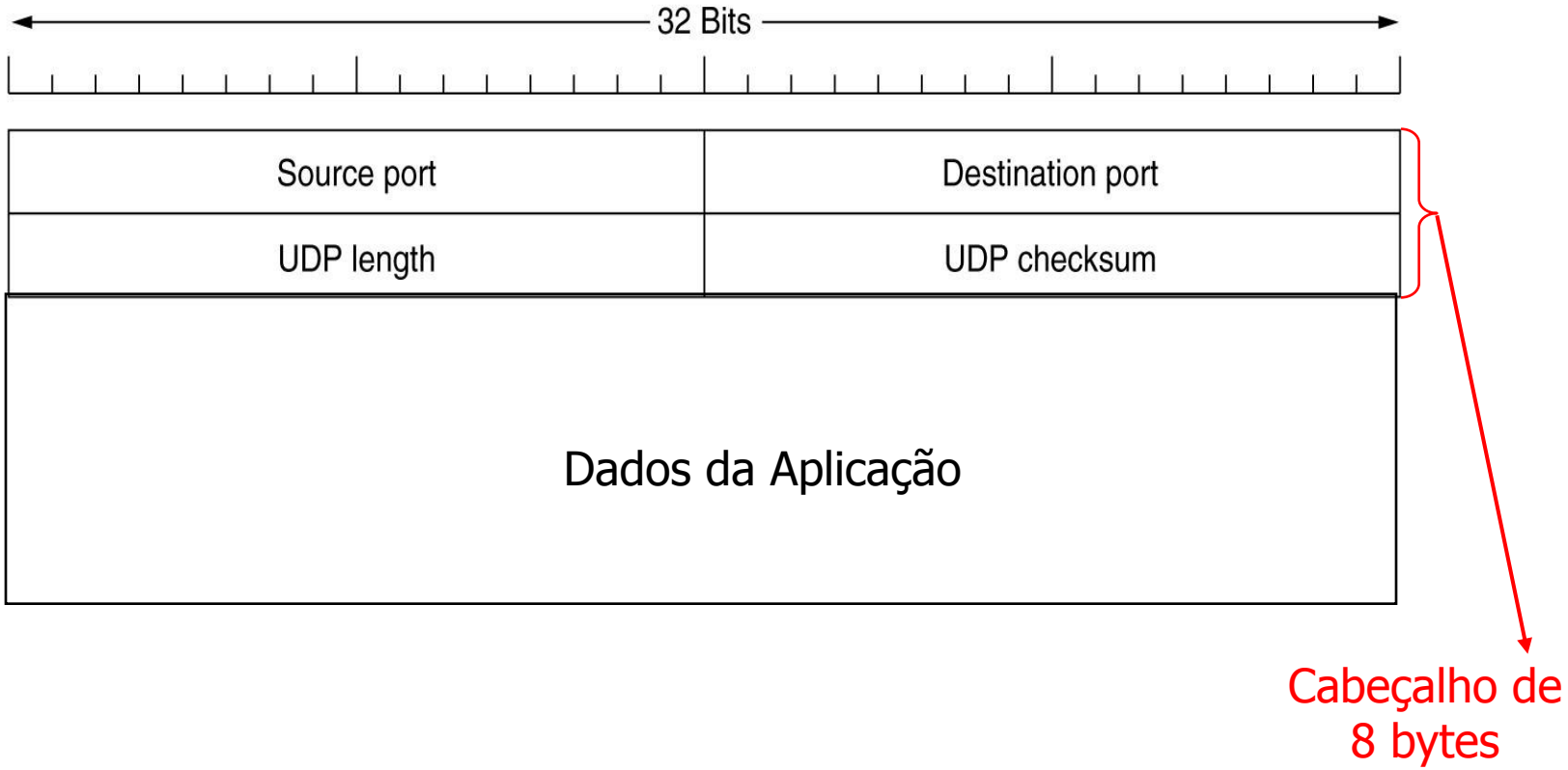
Não há estabelecimento de conexão

- Não há apresentação entre o UDP transmissor e o receptor
- Cada segmento UDP é tratado de forma independente dos outros

Serviço *best-effort*, segmentos UDP podem ser:

- Perdidos
- Entregues fora de ordem para a aplicação

Estrutura do Segmento UDP



Estrutura do Segmento UDP

Source Port e *Destination Port*:

- Identificam os pontos extremos (processos) nas máquinas de origem e destino
- Quando um pacote UDP chega, sua carga útil é entregue ao processo associado à porta destino

UDP *length*:

- Tamanho total do segmento (incluindo o cabeçalho) em bytes

UDP *checksum*:

- Campo opcional, armazenado como 0 se não for calculado

Estrutura do Segmento UDP

UDP *checksum* :

- Transmissor:
  - Calcula o complemento de 1 da soma de todas as palavras de 16 bits do segmento
- Receptor:
  - Computa o total de verificação do segmento recebido
  - Verifica se o total de verificação é igual ao valor do campo no cabeçalho
    - Não: erro detectado
    - Sim: não há erros

UDP

O UDP não faz:

- Controle de fluxo
- Controle de erros
- Retransmissão após a recepção de um segmento incorreto

O UDP apenas fornece uma interface para o protocolo IP com o recurso adicional de multiplexação/demultiplexação de vários processos que utilizam as portas

UDP

Porque existe um UDP?

- Não há estabelecimento de conexão e portanto não introduz nenhum atraso
- Não há estado de conexão nem no transmissor e nem no receptor
- Pequeno *overhead* no cabeçalho do pacote
- Taxa de envio não regulada (não há controle de congestionamento)

UDP

Onde é usado?

- Aplicações cliente-servidor onde existem apenas uma requisição e uma resposta
- O custo para estabelecer uma conexão é alto quando comparado com a transferência de dados
- Aplicações de multimídia contínua (*streaming*)
  - Tolerantes à perda
  - Sensíveis à taxa

TCP

Fornece um fluxo de bytes fim-a-fim confiável em uma sub-rede não confiável

Foi projetado para se adaptar dinamicamente às propriedades da sub-rede e ser robusto diante dos muitos tipos de falhas que podem ocorrer

Principal protocolo de transporte da arquitetura TCP/IP

- A maior parte das aplicações na Internet são baseadas no TCP

TCP – Características

1. Transferência confiável de dados
   - TCP garante que os dados serão entregues da forma que foram enviados
2. Orientado à conexões
   - Conexões são gerenciadas nos sistemas finais
3. Controle de fluxo
   - Transmissor não esgota a capacidade do receptor
4. Controle de congestionamento
   - Transmissor não esgota os recursos da rede

O Modelo de Serviço do TCP

A camada TCP aceita fluxos de dados das aplicações, dividi-os em partes de no máximo 65.495 bytes (fluxo de bytes e não fluxo de mensagens)

- Na prática, com frequência, temos 1.460 bytes de dados, para que ele possa caber em um único quadro Ethernet com os cabeçalhos IP e TCP

Envia cada parte para a camada IP como um segmento distinto

Ao chegar à camada IP da máquina destino, os dados são entregues à entidade TCP, que restaura os fluxos de bytes originais

O Modelo de Serviço do TCP

A entidade TCP receptora retorna um segmento (com ou sem dados) com um número de confirmação igual ao próximo número de sequência que espera receber

Se o *timer* do transmissor expirar antes de a confirmação ser recebida, o segmento será retransmitido

O Modelo de Serviço do TCP

Usa um serviço não confiável para prover um serviço de entrega de dados confiável para as aplicações

Deve ser capaz de compensar perdas e atrasos na sub-rede de comunicação de tal forma a prover o transporte de dados fim-a-fim de forma eficiente

Deve ser capaz de executar essas tarefas sem sobrecarregar a sub-rede de comunicação e os roteadores

O Modelo de Serviço do TCP

O serviço TCP necessita que uma **conexão** seja explicitamente estabelecida entre um soquete da máquina transmissora e um soquete da máquina receptora

- Estabelecimento da conexão
- Transferência de dados
- Término da conexão

O Modelo de Serviço do TCP

As conexões são identificadas pelos soquetes das máquinas transmissora e receptora, ou seja, cada conexão é identificada por:

- Endereço IP origem
- Porta origem
- Endereço IP destino
- Porta destino

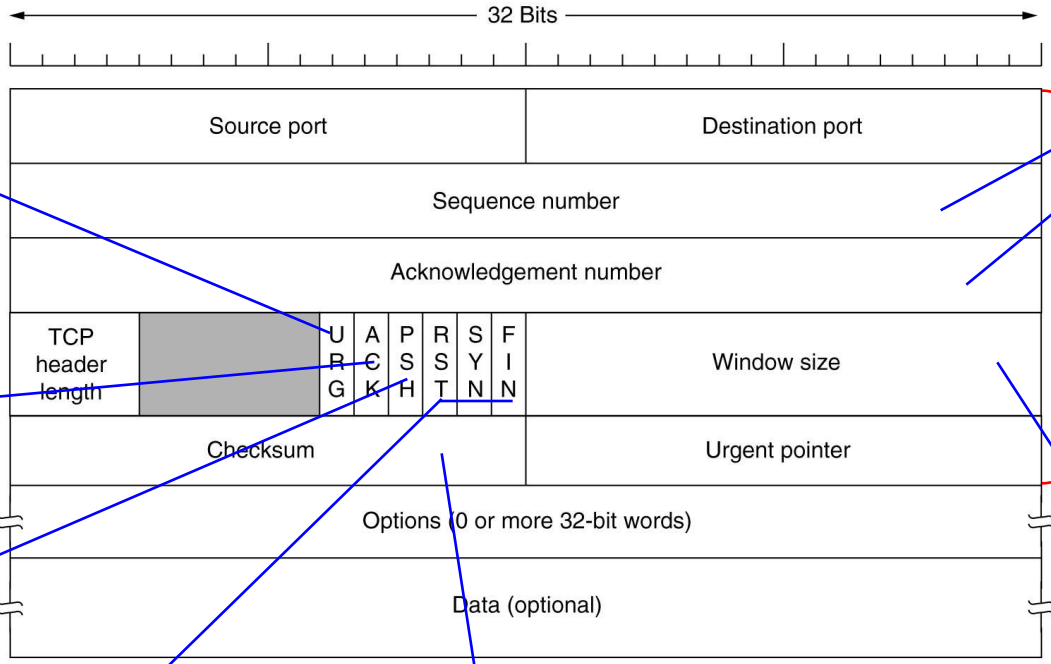
Um soquete pode ser utilizado por várias conexões ao mesmo tempo (duas ou mais conexões podem terminar no mesmo soquete)

O Modelo de Serviço do TCP

Todas as conexões TCP são *full-duplex* e ponto a ponto

- Tráfego pode ser feito em ambas as direções ao mesmo tempo
- Cada conexão possui exatamente dois pontos terminais
- TCP não admite multidifusão e difusão

O Segmento TCP



contagem por bytes de dados (não segmentos!)

20 bytes

URG: dados urgentes (pouco usado)

ACK: campo de ACK é válido

PSH: produz envio de dados (pouco usado)

RST, SYN, FIN: estabelecimento de conexão (comandos de criação e término)

Total de verificação do segmento

número de bytes que o receptor está pronto para aceitar

Tarefa 2 – postar no Canvas até 07/03/2021

1. Os programas a seguir estão disponíveis no endereço [www.cdk5.net/ipc]. Utilize-os para fazer uma série de testes para determinar as condições nas quais os datagramas, às vezes, são descartados. Dica: o programa cliente deve ser capaz de variar o número de mensagens enviadas e seus tamanhos; o servidor deve detectar quando uma mensagem de um cliente específico é perdida.

Tarefa 2 – postar no Canvas até 07/03/2021

O cliente UDP envia uma mensagem para o servidor e obtém uma resposta

```
import java.net.*;
import java.io.*;
public class UDPClient{
   public static void main(String args[ ]){
      // args fornece o conteúdo da mensagem e o nome de host do servidor
      DatagramSocket aSocket = null;
      try {
            aSocket = new DatagramSocket( );
            byte [ ] m = args[0].getBytes( );
            InetAddress aHost = InetAddress.getByName(args[1]);
            int serverPort = 6789;
            DatagramPacket request =
                  new DatagramPacket(m, m.length( ), aHost, serverPort);
            aSocket.send(request);
            byte[ ] buffer = new byte[1000];
            DatagramPacket reply = new DatagramPacket(buffer, buffer.length);
            aSocket.receive(reply);
            System.out.println("Reply: " + new String(reply.getData( )));
      } catch (SocketException e){System.out.println("Socket: " + e.getMessage( ));
      } catch (IOException e){System.out.println("IO: "+ e.getMessage( ));
      } finally { if(aSocket!= null) aSocket.close( );}
   }
}
```

Tarefa 2 – postar no Canvas até 07/03/2021

O servidor UDP recebe uma mensagem e a envia de volta para o cliente.

```
import java.net.*;
import java.io.*;
public class UDPServer{
public static void main(String args[ ]){
        DatagramSocket aSocket = null;
        try{
                aSocket = new DatagramSocket(6789);
                byte[ ] buffer = new byte[1000];
               while(true){
                  DatagramPacket request = new DatagramPacket(buffer, buffer.length);
                    aSocket.receive(request);
                    DatagramPacket reply = new DatagramPacket(request.getData( ),
                        request.getLength( ), request.getAddress( ), request.getPort( ));
                 aSocket.send(reply);
                 }
        } catch (SocketException e){System.out.println("Socket: " + e.getMessage( ));
        } catch (IOException e) {System.out.println("IO: " + e.getMessage( ));
        } finally {if (aSocket!= null) aSocket.close( );}
        }
}
```

Tarefa 2 – postar no Canvas até 07/03/2021

2. Use o programa cliente e servidor UDP do exercício anterior para fazer um programa cliente que leia repetidamente uma linha de entrada do usuário, a envie para o servidor em uma mensagem datagrama UDP e receba uma mensagem do servidor. O cliente estabelece um tempo limite em seu soquete para que possa informar o usuário quando o servidor não responder.

Tarefa 2 – postar no Canvas até 07/03/2021

3. Os programas a seguir estão disponíveis no endereço [www.cdk5.net/ipc]. Modifique-os de modo que o cliente leia repetidamente uma linha de entrada do usuário e a escreva no fluxo. O servidor deve ler repetidamente o fluxo, imprimindo o resultado de cada leitura. Faça uma comparação entre o envio de dados em mensagens de datagrama UDP e por meio de um fluxo.

Tarefa 2 – postar no Canvas até 07/03/2021

O cliente TCP estabelece uma conexão com o servidor, envia uma requisição e recebe uma resposta.

```
import java.net.*;
import java.io.*;
public class TCPClient {
        public static void main (String args[ ]) {
        // os argumentos fornecem a mensagem e o nome de host de destino
           Socket s = null;
           try{
               int serverPort = 7896;
               s = new Socket(args[1], serverPort);
               DataInputStream in = new DataInputStream(s.getInputStream( ));
               DataOutputStream out =
                       new DataOutputStream( s.getOutputStream( ));
               out.writeUTF(args[0]); // UTF é uma codificação de string; veja a Seção 4.3
                String data = in readUTF( );
               System.out.println("Received: "+ data);
              }catch (UnknownHostException e){
               System.out.println("Sock:"+e.getMessage( ));
           } catch (EOFException e){System.out.println("EOF:"+e.getMessage( ));
           } catch (IOException e){System.out.println("IO:"+e.getMessage( ));
        } finally {if(s!=null) try {s.close( );}catch (IOException e){/*close falhou*/}}
        }
}
```

Tarefa 2 – postar no Canvas até 07/03/2021

O servidor TCP estabelece uma conexão para cada cliente e, em seguida, ecoa o pedido do cliente

```
import java.net.*;
import java.io.*;
public class TCPServer {
    public static void main (String args[ ]) {
      try{
        int serverPort = 7896;
        ServerSocket listenSocket = new ServerSocket(serverPort);
        while(true) {
          Socket clientSocket = listen Socket.accept( );
          Connection c = new Connection(clientSocket);
        }
      } catch(IOException e) {System.out.println("Listen :"+e.getMessage( ));}
    }
 }
class Connection extends Thread {
    DataInputStream in;
    DataOutputStream out;
    Socket clientSocket;
    public Connection (Socket aClientSocket) {
      try {
        clientSocket = aClientSocket;
        in = new DataInputStream(clientSocket.getInputStream( ));
        out =new DataOutputStream(clientSocket.getOutputStream( ));
        this.start( );
      } catch(IOException e) {System.out.println("Connection:"+e.getMessage( ));}
    }
    public void run( ){
      try {              // an echo server
        String data = in.readUTF( );
        out.writeUTF(data);
      } catch(EOFException e) {System.out.println("EOF:"+e.getMessage( ));
      } catch(IOException e) {System.out.println("IO:"+e.getMessage( ));
      } finally { try {clientSocket.close( );}catch (IOException e){/*close falhou*/ }}
    }
}
```

Tarefa 2 – postar no Canvas até 07/03/2021

4. Use os programas desenvolvidos no exercício anterior para testar o efeito sobre o remetente quando o receptor falha e vice-versa.