Sistemas Distribuídos

4 Comunicação entre Processos

- **Comunicação Síncrona e Assíncrona**
- **Comunicação UDP**
- **Comunicação TCP**
- **Comunicação *Multicast***

Profª Ana Cristina B. Kochem Vendramin
DAINF / UTFPR

Sistemas Distribuídos Síncronos

- Tempo de execução de cada passo computacional de um processo possui um limite de tempo superior e inferior.
- Cada mensagem transmitida deve ser recebida dentro de um tempo limitado.
- Cada processo tem um relógio local cuja taxa de deriva de relógio (*clock drift rate*) tem um limite conhecido.
- É possível utilizar *timeouts* para detectar falha de um processo ou link de comunicação.
- A determinação de limites realistas e a garantia de tais limites pode ser árdua ou impossível.

Sistemas Distribuídos Assíncronos

- Não permite suposições sobre os intervalos de tempo envolvidos em qualquer execução.
- Não há limites quanto ao tempo de execução de processos, atraso na entrega de mensagens e taxas de deriva de relógio.
- Dificuldade em detectar falhas de tempo.
  - Exemplo:
    - Internet → serviços best-effort.

Características da Comunicação entre Processos

- Comunicação Síncrona
  - Processos emissor e receptor sincronizam-se em todas as mensagens.
  - Operações send e receive são BLOQUEANTES
    - Quando um send é enviado, o processo (ou thread) emissor é bloqueado até a resposta chegar.
    - Quando um receive é emitido, o processo (ou thread) fica bloqueado até uma mensagem chegar.

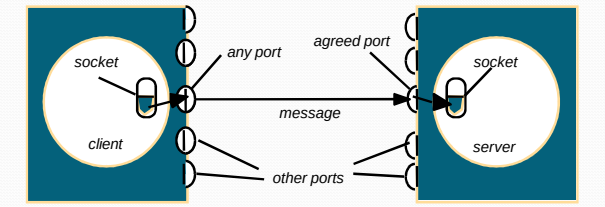
Características da Comunicação entre Processos

- Comunicação Assíncrona
  - Operação send é não-bloqueante
    - Processo emissor tem a permissão de prosseguir tão logo que a mensagem tenha sido copiada para um *buffer* local.
  - Operação receive pode ser bloqueante ou não

Características da Comunicação entre Processos

- Comunicação Assíncrona
  - Operação receive pode ser bloqueante ou não
    - Receive bloqueante
      - Não é necessário receber notificação, fora do seu fluxo normal de execução, para avisar que seu *buffer* possui dados a serem lidos
      - Em um ambiente como Java que suporta várias *threads* em um único processo, o *receive* bloqueante não tem desvantagem.
      - Pode ser emitido por uma *thread* enquanto outras *threads* no processo permanecem ativas.
      - Sistemas atuais costumam prover *receive* bloqueante.

Características da Comunicação entre Processos



[CDK 01]

Comunicação UDP

- Protocolo não orientado a conexão.
- Transporte de dados não confiável.
- Emprega send não-bloqueante e receive bloqueante.
- Mensagem de entrada é inserida em uma fila.
- Mensagens são retiradas da fila por invocações de receive no socket.
- Método receive bloqueia até que um datagrama seja recebido, a menos que um timeout seja configurado no socket.
- Método receive retorna o endereço IP e a porta do processo emissor junto com a mensagem.

Java API para Comunicação UDP

- **Classe InetAddress**
  - Obtém o endereço IP invocando o método getByName() e fornecendo o hostname.
    - Exceção: UnknownHostException.
- InetAddress host = InetAddress.getByName("localhost");

Java API para Comunicação UDP

- Duas classes: DatagramPacket, DatagramSocket
- **Classe DatagramPacket**
  - Fornece dois construtores:
    - Enviando: cria instância com um array de bytes contendo a mensagem, o comprimento da mensagem, o endereço IP e o número da porta destino do socket.
    - Recebendo: dois argumentos que especificam um array onde a mensagem e o seu tamanho em bytes serão armazenados.

Java API para Comunicação UDP

- **Classe DatagramPacket**
  - Mensagem pode ser recuperada do DatagramPacket através do método getData.
  - Porta - método getPort.
  - Endereço IP - método getAddress.

Java API para Comunicação UDP

- **Classe DatagramSocket**
  - Suporta sockets para enviar e receber datagramas.
  - Construtores:
    - Argumento: número da porta a ser utilizada pelo processo.
    - Um construtor sem argumentos permite ao sistema escolher qualquer porta local disponível.
  - Exceção SocketException
    - Se a porta já estiver em uso ou se uma porta menor que 1024 for especificada.

Java API para Comunicação UDP

- **Classe DatagramSocket**
  - Métodos:
    - **send**:
      - Argumento: instância da classe DatagramPacket contendo uma mensagem e seu destino.
    - **receive**:
      - Argumento: DatagramPacket vazio no qual serão inseridos a mensagem, seu tamanho e sua origem.
        - Exceção: IOExceptions

Java API para Comunicação UDP

- **Classe DatagramSocket**
  - Métodos (cont.):
    - **setSoTimeout**
      - Configuração de timeout.
      - Método receive ficará bloqueado durante o tempo especificado e depois lançará uma exceção InterruptedIOException.

Java API para Comunicação UDP

- Programa cliente
  - Cria um socket.
  - Envia uma mensagem a um servidor na porta 6789.
  - Espera resposta.
  - Os argumentos de entrada de main são o hostname do servidor e a mensagem.
    - Mensagem convertida em um array de bytes;
    - Hostname convertido em endereço IP.

[CDK01]

```
import java.net.*;
import java.io.*;
public class UDPClient{
    public static void main(String args[]){
      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,  args[0].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();}
  } }
```


Java API para Comunicação UDP

- Programa servidor
  - Cria um socket associado a porta de servidor 6789.
  - Repetidamente espera por requisições de clientes.
  - Responde enviando de volta a mesma mensagem.

[CDK01]

```
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();}
    }
}
```

Comunicação TCP

- Características:
  - Acknowledgement e retransmissões de mensagens.
  - Controle de fluxo.
  - Ordenação de mensagens.
  - Verifica mensagens duplicadas.
  - Estabelece a conexão antes de enviar o stream.

Comunicação TCP

- Processo de conexão
  - Cliente: cria um socket para o stream e o associa a uma porta qualquer, então envia um connect request ao servidor indicando a porta.
  - Servidor: cria um listening socket, o associa a uma porta local conhecida e espera pedidos de conexão de clientes.
  - O **listening socket** possui uma fila para colocar pedidos de conexão.
  - Servidor aceita uma conexão e então um socket stream é criado para se comunicar com o cliente.

Java API para Comunicação TCP

- Duas classes: ServerSocket, Socket
- **Classe ServerSocket**
  - Cria um socket na porta do servidor para escutar pedidos de conexão de clientes.
  - **Método accept**
    - Retira um pedido de conexão da fila ou bloqueia até que uma requisição chegue.
    - Resultado do accept: instância da classe Socket.
      - Fornece acesso aos streams de comunicação com o cliente.

Java API para Comunicação TCP

- **Classe Socket**
  - Usada pelos dois processos em comunicação TCP.
  - Cliente: usa um construtor para criar um socket associado a uma porta local e conecta o processo à porta do computador remoto.
  - Argumentos: hostname e porta do processo servidor.
  - Exceções:
    - UnknownHostException: hostname errado.
    - IOException: erro de I/O.
  - Métodos para acessar os dois fluxos associados ao socket:
    - getInputStream().
    - getOutputStream().

[CDK01]

```
import java.net.*;
import java.io.*;
public class TCPClient {
    public static void main (String args[]) {
    // arguments supply message and hostname of destination
    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 is a string encoding
          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)
    {System.out.println("close:"+e.getMessage());}}
    } }
```


[CDK01]

```
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 = listenSocket.accept();
               Connection c = new Connection(clientSocket);
          }
     } catch(IOException e) {System.out.println("Listen
     :"+e.getMessage());}
     }
}
```

```
// Servidor TCP (cont.)
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 failed*/}}
    }}
```

[CDK01]

Comunicação Multicast

- Construído sobre o protocolo IP.
- A nível de programa de aplicação, o IP multicast está disponível somente via UDP.
- Permite ao emissor transmitir um único pacote IP para um conjunto de computadores que formam o grupo multicast.
- Emissor não toma conhecimento das identidades dos receptores individuais e do tamanho do grupo.
- Especificado por um IP da Classe D
  - 1º octeto = 1110xxxx em IPv4;
  - 224.0.0.0 a 239.255.255.255.

Java API para Comunicação Multicast

- **Classe MulticastSocket**
  - Subclasse da DatagramSocket com habilidade adicional de juntar-se a grupos multicast.
    - Dois construtores - permite que sockets sejam criados para usar um porta local específica ou uma disponível.
    - Métodos:
      - **joinGroup;**
      - **leaveGroup;**
      - **setTimeToLive.**


```
import java.net.*;
import java.io.*;
public class MulticastPeer{
  public static void main(String args[]){
    MulticastSocket s =null;
    try {
     InetAddress group = InetAddress.getByName(args[1]);
     s = new MulticastSocket(6789);
     s.joinGroup(group);
    byte[] m = args[0].getBytes();
   DatagramPacket messageOut =
            new DatagramPacket(m, m.length, group, 6789);
   s.send(messageOut);
   byte[] buffer = new byte[1000];
  for(int i=0; i< 3; i++) {
      DatagramPacket messageIn = new DatagramPacket(buffer,buffer.length);
    s.receive(messageIn);
    System.out.println("Received:" + new String(messageIn.getData()));
}
s.leaveGroup(group);
}catch (SocketException e){System.out.println("Socket: " + e.getMessage());
}catch (IOException e){System.out.println("IO: " + e.getMessage());}
}finally {if(s != null) s.close();}}}
```

[CDK01]

Referências Bibliográficas

- Coulouris, George; Dollimore, Jean; Kindberg, Tim. Distributed Systems: concepts and design. Third Edition. Addison-Wesley 2001.
- Coulouris, George; Dollimore, Jean; Kindberg, Tim; tradução João Tortello. Sistemas Distribuídos: conceitos e projeto. 4. ed. Bookman 2007.