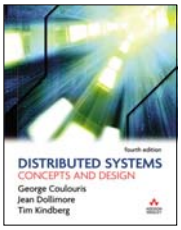
Material baseado no livro *Distributed Systems: Concepts and Design*, 4th Edition, Addison-Wesley, 2005.



4 – Comunicação entre Processos

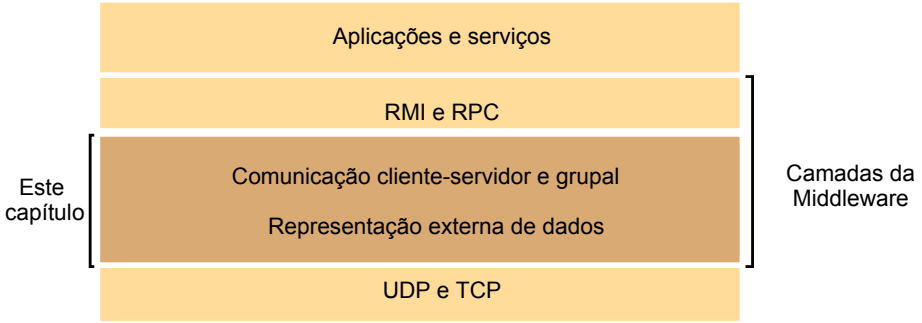
Agenda:

- **API para comunicação entre processos**
- **Representação de dados externos**
- **Comunicação cliente-servidor**
- **Comunicação grupal**

API para comunicação entre processos

- Interface de programação para utilizar os serviços básicos oferecidos pelo subsistema de comunicação subjacente
- Tópicos abordados:
  - Características da comunicação entre processos
  - Resolução de nomes
  - Comunicação usando *sockets*

Características de comunicação – contexto



Características de comunicação – operações

- Duas operações básicas: ***send*** e ***receive***
  - Definidas em termos de destinos e mensagens
  - Implementadas através de "filas" (*buffers*) de mensagens em cada lado da comunicação
  - Podem envolver a sincronização entre o processo que envia mensagem e o processo que a recebe

Características de comunicação – sincronismo

- Comunicação síncrona
  - Envio e recebimento sincronizados para cada mensagem (operações "bloqueantes")
    - *send* bloqueia o processo remetente até que o *receive* correspondente tenha sido executado pelo processo de destino
    - *receive* bloqueia o processo de destino até a chegada de uma mensagem
- Comunicação assíncrona
  - Envio e recebimento sem sincronização (operações "não-bloqueantes")
    - *send* libera o processo remetente assim que a mensagem tiver sido copiada para a fila local; execução e transmissão ocorrem em paralelo
    - *receive* libera o processo destino antes da chegada da mensagem, a qual deve ser retirada da fila posteriormente, através de um mecanismo de *pooling* ou de notificação (*callback*)

Características de comunicação – sincronismo

- Comunicação síncrona é a mais usada na prática!
  - Operações não-bloqueantes ainda podem exigir alguma forma de sincronização no nível da aplicação
  - Operações bloqueantes podem ser utilizadas sem interromper totalmente o processo que as invocou quando executadas em fluxos (*threads*) separados de execução
  - A maioria dos sistemas de comunicação atuais não oferece variações não-bloqueantes da operação *receive* (Por quê?)

Características de comunicação – destinos

- Destino = (endereço de rede + porta)
  - Endereço de rede corresponde ao IP do computador onde é executado o processo que receberá a mensagem
  - Porta corresponde ao destino de uma mensagem dentro de um mesmo computador (especificado como um inteiro entre 0 e 65535)
    - A cada porta é associado um único processo destino (exceto no caso de portas usadas para difusão seletiva de mensagens)
    - Um mesmo processo pode ter múltiplas portas associadas a ele
    - Qualquer processo que conhece o no. da porta de outro processo pode enviar mensagens para ele através dessa porta
    - Servidores costumam divulgar o no. de suas portas para os seus clientes ou usam portas padrão (ex.: HTTP = 80, FTP = 21, etc.)
- Não há transparência de localização!

Características de comunicação – destinos

- Destinos transparentes quanto a sua localização podem ser obtidos de duas maneiras:
  - Utilizando nomes simbólicos para referenciar endereços de rede e um serviço de nomes para traduzir nomes em endereços em tempo de execução
    - Forma mais comum de implementar transparência de localização na Internet!
    - Suporta re-alocação dos serviços mas não migração (Por que não?)
  - Utilizando identificadores independentes de localização para referenciar processos remotos oferecidos pelo próprio S.O.
    - Disponível em alguns sistemas operacionais distribuídos (ex.: Mach OS)
- Destino = (endereço de rede + PId)?
  - Em princípio, o próprio identificador (*PId*) do processo, criado pelo S.O, poderia ser usado como destino, no lugar do número de porta
  - Mas essa solução é ainda pior do ponto de vista de flexibilidade e transparência (Por quê?)

Características de comunicação – confiabilidade

- Definida em termos da *validade* e da *integridade* do serviço de comunicação
  - *Validade:* garantia de que as mensagens serão entregues mesmo diante de um número "razoável" de falhas de transmissão (perdas de pacotes)
  - *Integridade:* garantia de que as mensagens serão entregues sem serem corrompidas ou duplicadas

Características de comunicação – ordenação

- Serviço de comunicação deve garantir que as mensagens serão entregues na ordem em que são enviadas pelo processo remetente
- Entrega fora da ordem de envio pode ser considerada falha de transmissão por aplicações que dependem da ordem das mensagens (Exemplos?)

Resolução de nomes

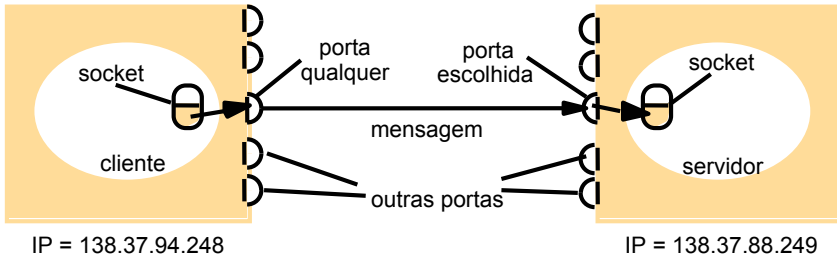
- API de Java para resolução de nomes de domínio
- Implementada através da classe *InetAddress*
  - Criação somente através de métodos estáticos
  - Métodos encapsulam localização e consulta a servidor DNS
  - Independente do formato da representação interna (IPv4, IPv6, ...)
- Exemplo de uso:

```
import java.net.*;
...
try {
   InetAddress ip = InetAddress.getByName("www.unifor.br");
Catch (UnknownHostException e){
   System.out.println("Endereço desconhecido!");
}
```


Comunicação usando *sockets* (I)

- *Socket:* abstração para cada um dos extremos (*end-points*) da comunicação entre dois processos
  - Originário do BSD UNIX e posteriormente incorporado em virtualmente todos os outros sistemas operacionais
  - Usado tanto para comunicação assíncrona (protocolo UDP) quanto para comunicação síncrona (protocolo TCP)
- Comunicação exige a criação de um *socket* pelo processo remetente e de um outro pelo processo destinatário
  - *socket* do destinatário (servidor) deve estar exclusivamente acoplado a uma porta do computador local
  - Um mesmo *socket* pode ser usado tanto para enviar quanto para receber mensagens (comunicação bidirecional)
  - Um processo pode usar múltiplas portas, mas não pode compartilhar portas com outros processos do mesmo computador (exceto no caso de *sockets* associados a endereços de difusão seletiva)

Comunicação usando *sockets* (II)



Comunicação usando *sockets* sobre UDP (I)

- Transmissão não confiável de "pacotes" (*datagrams*) entre um processo cliente e um processo servidor
  - Não garante entrega nem ordem de recebimento das mensagens
  - Exige que clientes e servidores criem um *socket* acoplado ao IP e a uma porta da máquina local
    - Servidor escolhe uma porta específica (conhecida pelos clientes)
    - Cliente utiliza uma porta qualquer dentre as portas disponíveis
- Comunicação sem conexão prévia
  - No cliente: endereço e porta do servidor passados como parâmetros de entrada para cada operação *send*
  - No servidor: endereço e porta do cliente recebidos como parâmetros de saída de cada operação *receive*

Comunicação usando *sockets* sobre UDP (II)

- Operações bloqueantes e não bloqueantes
  - *send* retorna logo após repassar a mensagem para os protocolos subjacentes (UDP/IP)
    - Mensagem é descartada na chegada se não há um *socket* acoplado ao endereço e à porta de destino
  - *receive* bloqueia processo se não houver mensagens na fila
    - Variação não bloqueante disponível em algumas implementações
    - Bloqueio pode ser controlado através de temporizadores (*timeouts*)
    - Pode receber mensagens de diferentes endereços de origem
- Modelo de falha
  - *Checksums* para detectar e rejeitar mensagens corrompidas
  - Garantia de entrega e ordem de chegada a cargo da aplicação

Comunicação usando *sockets* sobre UDP (III)

- Uso de UDP
  - Em algumas aplicações pode ser aceitável utilizar um serviço sujeito a falhas de omissão ocasionais (ex.: DNS, VoIP)
  - Uso de UDP evita custos normalmente associados com a garantia de entrega das mensagens:
    - Necessidade de armazenar informações de estado na origem e no destino
    - Transmissão de mensagens extras (além dos dados da aplicação)
    - Latência no envio

Exemplos de uso das API's em Java e C/Unix

- Em Java:
  - API implementada em algumas classes do pacote *java.net*
    - *DatagramPacket*
    - *DatagramSocket*
    - *Socket*
    - *ServerSocket*
- Em C/Unix:
  - API padrão (biblioteca de funções) para manipulação de *sockets*
    - Definições básicas no arquivo de cabeçalho *socket.h*

API de Java para comunicação com *sockets* sobre UDP

- Classes principais:
  - *DatagramPacket*
    - Usada pelo cliente para construir um *datagram* a partir de um array de bytes e do endereço+porta de um servidor
    - Usada pelo servidor para armazenar o *datagram* recebido juntamente com o endereço+porta do socket cliente
  - *DatagramSocket*
    - Usada para criar sockets para envio e recebimento de datagramas
    - Construtores com ou sem a opção de especificar a porta desejada (se o programador não especificar a porta, o sistema escolhe a primeira porta disponível)
    - Lança exceção *SocketException* se porta especificada já estiver em uso

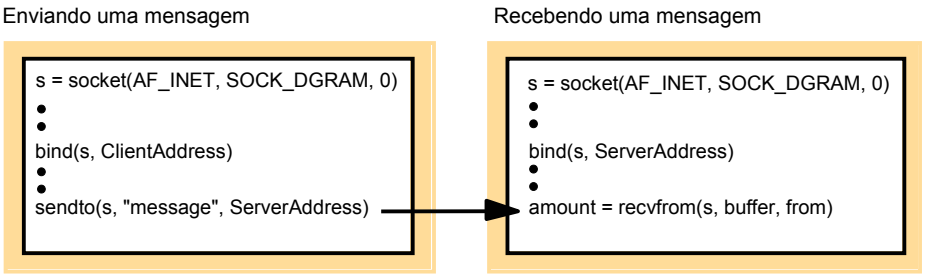
Exemplo de uso no lado do cliente

```
import java.net.*;
import java.io.*;
public class UDPClient{
   public static void main(String args[]){
            // args give message contents and server hostname
            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();}
   }
}
```


Exemplo de uso no lado do servidor

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

Exemplo de uso em C/Unix



ServerAddress e *ClientAddress* representam o endereço (IP + porta) do socket

Comunicação usando *sockets* sobre TCP (I)

- Transmissão confiável de mensagens entre um processo cliente e um processo servidor
  - Cliente solicita a conexão de um *socket* acoplado a uma porta local qualquer para o endereço e a porta do *socket* no servidor
  - Servidor aceita conexões entre seu *socket* local e os *sockets* dos clientes
  - Conexão feita através de um par de "fluxos" (*streams*)
    - Um fluxo de saída no cliente é conectado a um fluxo de entrada no servidor, e vice-versa
    - Permite a comunicação bi-lateral e concorrente entre clientes e servidores

Comunicação usando *sockets* sobre TCP (II)

- Noção de fluxos de dados abstrai das seguintes características da rede:
  - A aplicação pode escolher a quantidade de dados que vai ler ou escrever em um fluxo; cabe à implementação do TCP o quanto desses dados serão transmitidos em cada pacote IP
  - Um esquema de confirmações e temporizadores permite a detecção e retransmissão de mensagens extraviadas
  - Um esquema de controle de fluxo permite sincronizar a velocidade de envio na origem com a velocidade de recebimento no destino
    - Se o remetente estiver enviando dados muito rapidamente, ele é bloqueado até que o destinatário tenha recebido uma quantidade de dados suficiente
  - Identificadores de mensagens são associados a cada pacote IP, o que permite ao destinatário detectar e rejeitar mensagens duplicadas, ou re-ordenar eventuais mensagens recebidas fora de ordem

Comunicação usando *sockets* sobre TCP (III)

- Protocolos de interação
  - Clientes e servidores devem concordar sobre uma seqüência para a troca de mensagens e sobre como interpretar seu conteúdo
  - Falta de entendimento pode causar erros de interpretação (ex.: receber um inteiro esperando uma string) e bloqueios indevidos (ex.: tentativa de receber mais dados do que os que foram de fato enviados)
- Operações bloqueantes
  - *send* bloqueia o processo até que haja espaço disponível na fila correspondente ao *stream* de entrada do *socket* de destino
  - *receive* bloqueia o processo até que haja uma mensagem disponível na fila correspondente ao *stream* de entrada do *socket* local

Comunicação usando *sockets* sobre TCP (IV)

- Modelo de falha
  - *Checksums* para detectar e rejeitar mensagens corrompidas
  - Números seqüenciais para detectar e rejeitar mensagens duplicadas
  - Temporizadores e retransmissões para lidar com perdas de pacotes
    - Podem não ser suficientes para lidar com falhas de omissão decorrentes da sobrecarga da rede
    - Processos podem não conseguir distinguir se as suas mensagens foram de fato recebidas (qual o problema?)
- Uso de TCP
  - Muitas aplicações da Internet rodam sobre conexões TCP, em portas reservadas, incluindo:
    - HTTP (porta 80)
    - FTP (porta 20/21)
    - Telnet (porta 23)
    - SMTP (porta 25)

API de Java para comunicação com *sockets* sobre TCP

- Classes principais:
  - *Socket*
    - Usada pelo cliente para criar um *socket* (acoplado a uma porta local qualquer) para ser conectado ao *socket* de um servidor remoto
    - Lança exceção *UnkownHostException* se houver problemas com o endereço do servidor, ou *IOException*, se houver erros de E/S
    - Métodos *getInputStream* e *getOutputStream* permitem acesso ao *stream* de entrada e ao *stream* de saída, respectivamente, associados ao *socket*
  - *ServerSocket*
    - Usada pelo servidor para criar um *socket* (acoplado a uma determinada porta local) para aceitar conexões dos clientes
    - Método *accept* devolve um objeto da classe *Socket,* já conectado ao socket de um cliente que tenha solicitado conexão, ou bloqueia o servidor até que um novo pedido de conexão seja recebido

Exemplo de uso no lado do cliente

```
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 see Sn 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){System.out.println("close:"+e.getMessage());}}
        }
}
```


Exemplo no lado do servidor

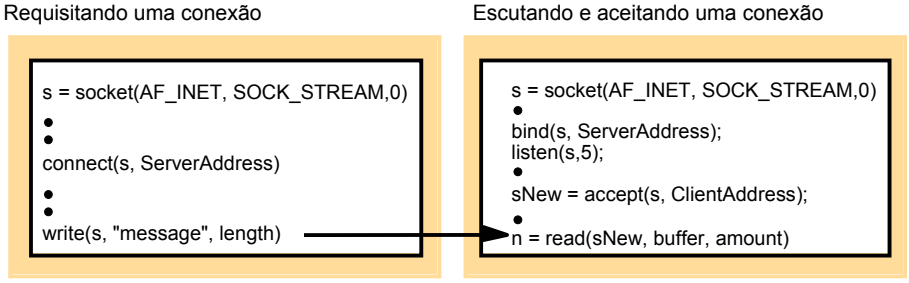
```
import java.net.*;
import java.io.*;
public class TCPServer {
  public static void main (String args[]) {
        try{
                int serverPort = 7896;
                ServerSocket listenSocket = new ServerSocket(serverPort);
                while(true) {
                        Socket clientSocket = listenSocket.accept();
                        Connection c = new Connection(clientSocket);
                }
        } catch(IOException e) {System.out.println("Listen :"+e.getMessage());}
  }
}

// continua no próximo slide
```

Exemplo no lado do servidor (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*/}}
        }
}
```


Exemplo de uso em C/Unix



ServerAddress e *ClientAddress* representam o endereço (IP + porta) do socket

Agenda

- API para comunicação entre processos
- Representação de dados externos
- Comunicação cliente-servidor
- Comunicação grupal

Representação de dados externos

- Motivação:
  - Programas e rede manipulam dados em diferentes níveis de abstração (abstrações de dados X seqüência de bytes)
    - Abstrações de dados devem ser “achatadas” (ou “serializadas”) pelo remetente, antes da transmissão, e então reconstruídas pelo destinatário, após o seu recebimento
  - Nem todos os programas utilizam a mesma forma de representação para os mesmos tipos de dados
    - Diferentes conjuntos de tipos primitivos
    - Diferentes tamanhos (em bytes) para os mesmos tipos
    - Diferentes ordem de bits para tipos do mesmo tamanho
    - Diferentes arquiteturas para números de ponto flutuante
    - Diferentes padrões de caracteres (EBCDIC, ASCII, Unicode, ...)
- Como fazer para que programas em diferentes computadores troquem dados de forma consistente?

Representação de dados externos

- Duas abordagens:
  1. Remetente envia os dados no seu formato original, junto com uma indicação do formato utilizado, e destinatário os converte novamente para o seu formato local, se necessário
     - A favor: evita conversões desnecessárias
     - Contra: destinatário precisa conhecer todos os outros formatos
  2. Remetente converte os dados para um formato externo (acordado previamente) antes da transmissão, e destinatário os converte novamente do formato externo para o seu formato local
     - A favor: remetente e destinatário só precisam conhecer um único formato externo
     - Contra: conversões desnecessárias quando os computadores envolvidos utilizam o mesmo formato (como otimizar?)
     - A mais utilizada na prática (ex.: RPC, CORBA, RMI, SOAP)

Empacotamento e desempacotamento de dados (I)

- Empacotamento (*marshalling*):
  - Agrupamento de um conjunto de itens de dados num formato adequado (representação externa) para transmissão via mensagens
- Desempacotamento (*unmarshalling*):
  - Re-agrupamento dos dados recebidos no formato externo para produzir um conjunto equivalente de itens de dados no formato local do destinatário
- Realizados automaticamente pelas camadas da middleware, sem intervenção do usuário ou programador
  - Necessitam da especificação detalhada dos tipos de dados a serem transmitidos em uma linguagem apropriada (IDL, WSDL, etc)
  - Evitam erros de programação
  - Podem ser usados para outros fins (ex.: persistência)

Empacotamento e desempacotamento de dados (II)

- Três abordagens discutidas no curso:
  - CDR (CORBA)
    - Representação de dados externos em formato binário, utilizada na invocação de objetos remotos com CORBA
    - Suporte para múltiplas linguagens de programação
  - Serialização de objetos (Java RMI)
    - Representação de dados externos em formato binário, utilizada, entre inúmeros outros usos, na invocação de objetos remotos com Java RMI
    - Apenas para uso com Java
  - XML (SOAP)
    - Representação de dados externos em formato textual (documentos XML), utilizada na invocação de serviços web
    - Suporte pra múltiplas linguagens de programação
- Representações textuais geralmente são bem maiores do que representações binárias equivalentes (Por quê? Vantagens?)

Representação de dados externos em CORBA

- Representação comum de dados (*Common Data Representation – CDR*) introduzida na versão CORBA 2.0
- Definição de formato para todos os tipos de dados que podem ser usados como argumentos e valores de retorno em invocações remotas para objetos CORBA
  - 15 tipos primitivos: *short* (16 bits), *long* (32 bits), *double* (64 bits), ...
  - 6 tipos compostos: *sequence*, *string*, *array*, *struct*, *enumerated*, *union*
- Ordem dos bits segue o estilo local do remetente e é indicada em cada mensagem (conversão a critério do destinatário)
- Padrão de caracteres acordado entre cliente e servidor
- Valores primitivos dos tipos compostos adicionados em seqüência, seguindo uma ordem específica para cada tipo

Representação de dados externos em CORBA: tipos compostos

Tipo	*Representação*
sequence	Tamanho (*unsigned long*) seguido pelos elementos em ordem
string	Tamanho (*unsigned long*) seguido pelos caracteres em ordem
array	Elementos em ordem (tamanho fixo)
struct	Na ordem da declaração dos componentes
enumerated	Valores (*unsigned long*) na ordem especificada
union	Marcador de tipo seguido pelo membro selecionado

Exemplo de representação em CORBA CDR

- Definição de um tipo *Pessoa* em C/C++:

  struct Pessoa { string nome; string lugar; unsigned long ano};

- Representação em CORBA CDR de um item de dado do tipo *Pessoa* com os atributos {'Smith', 'London', 1934}:

Índice (seq. de bytes) ← 4 bytes →		Comentário
0–3	5	tamanho da string
4–7	"Smit"	'Smith'
8–11	"h___"	
12–15	6	tamanho da string
16–19	"Lond"	'London'
20-23	"on__"	
24–27	1934	unsigned long

Serialização de objetos em Java

- Serialização: atividade de "achatar" o estado de um objeto (ou de uma hierarquia de objetos relacionados) para uma forma seqüencial mais adequada para armazenamento ou transmissão
  - Valores dos atributos primitivos escritos num formato binário portável
  - Atributos não primitivos serializados recursivamente
  - Não pressupõe conhecimento sobre o tipo dos objetos para o processo de reconstrução ("desachatamento")
    - Informações sobre a classe dos objetos, como nome e número de versão, são incluídas na forma serial
  - Implementada através dos métodos *writeObject* e *readObject* das classes *ObjectOutputStream* e *ObjectInputStream*, respectivamente
- Executada de forma transparente pelas camadas da middleware em Java (uso intensivo de Reflexão!)

Exemplo de serialização em Java

- Definição de um tipo *Pessoa* em Java:

```
public class Pessoa implements Serializable {
    private String nome; private String lugar; private int ano;
    public Pessoa(String nome, String lugar, int ano) {
            this.nome = nome; this.lugar = lugar; this.ano = ano;
    } // continua com a definição dos métodos...
}
```

- Exemplo (simplificado) de serialização de um objeto do tipo *Pessoa*: *Pessoa p = new Pessoa("Smith", "London", 1934);*

Valores serializados				Comentário
Pessoa	No. de versão (8 bytes)		h0	nome da classe, número de versão
3	int ano	java.lang.String nome:	java.lang.String lugar:	número, tipo e nome das variáveis de instância
1934	5 Smith	6 London	h1	valores das variáveis de instância

Representação de dados externos em XML (I)

- XML é uma linguagem de marcação de propósito geral para uso na Web, definida e mantida pelo W3C
  - Projetada para facilitar a codificação textual de documentos estruturados
  - Assim com HTML, XML é derivada de SGML (*Standard Generalized Markup Language*), que é bem mais complexa e, por isso, foi pouco utilizada na prática
- Itens de dados em XML são "marcados" com strings especiais chamadas marcadores (*tags*), que representam a estrutura lógica dos dados
  - Diferença para HTML?
- XML é extensível!
  - Usuários podem definir seus próprios marcadores
  - Extensões precisam ser acordadas no caso de documentos compartilhados por múltiplas aplicações

Representação de dados externos em XML (II)

- XML é a base para a representação de dados externos (padrão SOAP) e interfaces (padrão WSDL) em todas as middlewares baseadas na tecnologia de serviços web
  - Vantagens:
    - Permite "compreensão" e facilita monitoramento por humanos (útil em atividades de teste e depuração)
    - Independente de plataforma
  - Desvantagens:
    - Mensagens bem maiores que suas equivalentes em formato binário
    - Maior tempo de processamento para empacotamento/desempacotamento
  - Alternativas:
    - Compactação de mensagens
      - Reduz tamanho mas piora processamento
    - XML em formato binário
      - Reduz tamanho e processamento, mas impede compreensão

Representação de dados externos em XML (III)

- Representação em XML de um elemento do tipo Pessoa:

```
<person id="123456789">
                    <name>Smith</name>
                    <place>London</place>
                    <year>1934</year>
                    <!-- a comment -->
</person >
```

- Conteúdos originalmente em binário (imagens, arquivos criptografados, etc) podem ser representados utilizando a notação *base64*
  - Cada byte mapeado para um caractere alfanuméricos mais os símbolos de controle “+”, “/” e “=”
- Regras para validação de documentos bem formados
  - Ex.: completude de marcadores, correto aninhamento de elementos, possuir um único elemento raiz, etc

Representação de dados externos em XML (IV): Espaço de nomes

- Os escopo dos marcadores de um documento XML pode ser restringido através da definição de espaços de nomes (*namespaces*)
  - Um espaço de nomes corresponde a um conjunto de nomes para uma coleção de tipos e atributos de elementos
  - Referenciado por um URL (usado com valor do atributo *xmlns*)
  - O nome do espaço de nomes (sufixo do atributo *xmlns*) pode ser usado como prefixo para os nomes dos marcadores e atributos de uma espaço de nome particular
  - O exemplo ao lado ilustra um elemento do tipo *Pessoa* definido utilizando o espaço de nomes *pers*:

```
<person  pers:id="123456789“
                xmlns:pers = "http://www.cdk4.net/person">
                <pers:name> Smith </pers:name>
                <pers:place> London </pers:place >
                <pers:year> 1934 </pers:year>
</person>
```

Representação de dados externos em XML (V): Definição de esquema

- Um esquema XML define:
  - Os elementos e atributos (incluindo seus tipos e valores *default*) que podem aparecer em um documento
  - Como os elementos são aninhados e em qual ordem e quantidade
  - Se um elemento pode ser vazio
  - Se um elemento pode conter texto
- Um mesmo esquema pode ser compartilhado por múltiplos documentos XML
  - Permite a validação de documentos de acordo com o esquema
    - Ex.: uma mensagem SOAP pode ser validada pelo processo destinatário com base no esquema XML previamente definido para esse padrão

Representação de dados externos em XML (VI): Definição de esquema

- Duas alternativas de definição
  - DTD (*Document Type Definition*)
    - Definição similar ao formato BNF (gramática de linguagens)
  - XML Schema
    - Definição em XML, utilizado um esquema próprio
    - Exemplo de um esquema para o tipo *Pessoa* definido em XML Schema:

```
<xsd:schema  xmlns:xsd = URL of XML schema definitions >
   <xsd:element name= "person" type ="personType" />
      <xsd:complexType name="personType">
         <xsd:sequence>
          <xsd:element name = "name"  type="xs:string"/>
             <xsd:element name = "place"  type="xs:string"/>
             <xsd:element name = "year"  type="xs:positiveInteger"/>
         </xsd:sequence>
         <xsd:attribute name= "id"   type = "xs:positiveInteger"/>
      </xsd:complexType>
</xsd:schema>
```

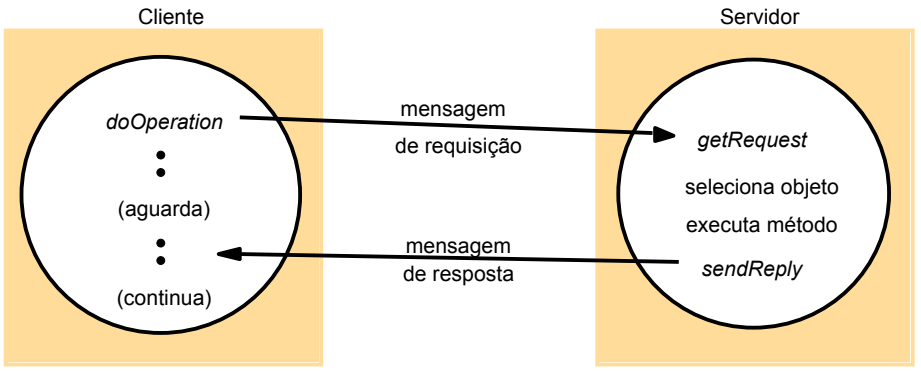
Agenda

- API para comunicação entre processos
- Representação de dados externos
- Comunicação cliente-servidor
- Comunicação grupal

Comunicação cliente-servidor

- Projetada para apoiar os papéis e as trocas de mensagens típicos do modelo cliente-servidor
  - Comunicação síncrona confiável do tipo requisição/resposta (resposta pode ser usada como confirmação para requisição)
  - Opção para comunicação assíncrona se clientes aceitarem (e tolerarem) receber a resposta *a posteriori*
- Protocolo baseado em três primitivas (exemplo em Java):
  - *public byte[] doOperation (RemoteObjectRef o, int methodId, byte[] arguments)*
    - Envia uma requisição para um objeto remoto e retorna a resposta obtida
  - *public byte[] getRequest ()*
    - Obtém uma requisição de um cliente através da porta do servidor
  - *public void sendReply (byte[] reply, InetAddress clientHost, int clientPort)*
    - Envia resposta para o cliente no dado endereço (IP + porta)

Protocolo requisição-resposta



Protocolo requisição-resposta (II)

- Estrutura das mensagens:

messageType	*int (0=Requisição, 1= Resposta)*
requestId	*int*
objectReference	*referência para objeto remoto*
methodId	*int ou instância de Method (em Java)*
arguments	*array de bytes*

  - Identificador da requisição (*requestId*) gerado a partir de um contador de requisições mantido pelo processo remetente (contador é zerado ao atingir um valor máximo pré-estabelecido)
  - Identificador da mensagem composto por pelo identificador da requisição mais endereço (IP + porta) do processo remetente
    - Útil para implementar comunicação confiável

Protocolo requisição-resposta (III)

- Modelo de falha
  - Pressupõe apenas falhas por omissão (sem comportamento arbitrário)
  - Reenvio da mensagem quando o processo remetente detecta uma possível falho do processo de destino
    - Detecção de falhas através do uso de temporizadores
  - Destinatário verifica os identificadores das mensagem recebidas para descartar eventuais duplicações
  - Duas soluções para o servidor tratar a perda da mensagem de resposta:
    - Re-execução da operação pelo servidor
      - Permitida apenas para operações idempotentes (exemplos?)
    - Manutenção de um histórico (*log*) de respostas
      - Maior custo de memória (uma otimização é manter apenas a última resposta de cada cliente no *log*)
      - Nova requisição pode ser interpretada como confirmação da resposta anterior
- Tratamento de falhas pode ser desnecessário se o protocolo for implementado sobre um mecanismo de comunicação confiável (ex.: TCP)

Variações do protocolo requisição-resposta

- Protocolo (só de) *requisição* (R)
  - Usado quando não há valor de retorno nem necessidade de confirmação
  - Cliente continua execução imediatamente após o envio da requisição
- Protocolo *requisição-resposta* (RR)
  - Baseado no protocolo requisição-resposta visto antes
  - Sem necessidade de confirmação explícita (resposta do servidor serve como confirmação para requisição do cliente, e nova requisição do cliente serve como confirmação para resposta anterior do servidor)
- Protocolo *requisição-resposta-confirmação* (RRC)
  - Inclui confirmação explícita do cliente para cada resposta do servidor (mensagem de confirmação contém o identificador da requisição original)
  - Usada no servidor para descartar entradas do *log* de respostas
    - Identificadores das mensagens interpretados como confirmações para todas as requisições anteriores recebidas de um mesmo cliente
    - Tolera a perda das mensagens de confirmação

Troca de mensagens nos protocolos R, RR, e RRC

Protocolo	*Mensagem enviada por*		
	Cliente	*Servidor*	*Cliente*
R	*Requisição*		
RR	*Requisição*	*Resposta*	
RRC	*Requisição*	*Resposta*	*Confirmação da resposta*

Protocolo HTTP: um estudo de caso

- Usado inicialmente para troca de informações entre navegadores (clientes) e servidores da Web
  - Protocolo de transporte para invocações de serviços web
- Interação iniciada pelo cliente, que fornece uma URL indicando o endereço (IP + porta) do servidor e o identificador do recurso requisitado
- Suporte para um conjunto fixo de métodos (*GET*, *PUT*, *POST*, etc) aplicáveis a todos os recursos
- Suporte para negociação de conteúdo e autenticação
  - Negociação de conteúdo: cliente informa quais tipos de representação de dados ele suporta; servidor escolhe a representação mais apropriada para cada cliente
  - Autenticação: servidor envia um “desafio” para o cliente no primeiro acesso a uma área protegida; cliente obtém nome e senha do usuário e os envia junto com cada requisição subseqüente como “credenciais” para acesso aos recursos do servidor

Protocolo HTTP: seqüência de interações

- No protocolo original:
  - Um cliente requisita conexão a um servidor, que pode ou não aceitá-la
  - Se a conexão for aceita, o cliente então envia uma mensagem de requisição ao servidor
  - Servidor envia uma mensagem de reposta ao cliente
  - A conexão é fechada em ambos os lados
- Vantagem
  - Servidor não precisa manter informação sobre o estado das conexões
- Desvantagem
  - Risco de sobrecarga do servidor e/ou da rede sob alta demanda
- Versões mais recentes do protocolo (HTTP1.1+) permitem estabelecer *conexões persistentes*
  - Conexão permanece aberta ao longo de várias interações
  - Reenvio automático de mensagens para operações idempotentes

Protocolo HTTP: representação de dados

- Mensagens de requisição e resposta empacotadas em formato textual (padrão ASCII)
  - Garante a fácil utilização do protocolo e de suas mensagens por programadores
- Recursos não textuais empacotados como seqüências de bytes, possivelmente compactadas
- Formato padrão para representação de dados de tipos conhecidos
  - Estruturas *MIME – Mutipurpose Internet Mail Extensions* (ex.: *text/plain*, *text/html*, *image/jpeg*, *application/postscript*, ...)
  - Clientes podem especificar quais estruturas são conhecidas/aceitas

Protocolo HTTP: métodos (I)

- *GET*: requisita o recurso identificado pela URL
  - Se recurso referencia dados, o servidor devolve os dados identificados
  - Se recurso referencia um programa, o servidor executa o programa e devolve os resultados da execução
  - Argumentos podem ser adicionados à URL (por exemplo, para enviar os conteúdo de um formulário como argumento para um programa *GCI*)
  - Pode ser configurado para obter apenas parte do recurso
- *HEAD*: idêntico ao *GET*, mas no lugar dos dados o servidor apenas devolve meta-informações sobre os mesmos (ex.: data da última alteração, tipo, tamanho, etc)
- *POST*: especifica a URL de um recurso do servidor (por exemplo, um programa) capaz de tratar dos dados enviados como argumentos

Protocolo HTTP: métodos (II)

- *PUT*: requisita que os dados enviados sejam armazenados como o identificador da URL, seja através da modificação de um recurso existente ou através da criação de um novo
- *DELETE*: requisita a exclusão do recurso identificado pela URL
  - Nem sempre permito pelo servidor
- *OPTIONS*: requisita uma lista contendo os métodos que podem ser aplicados a uma dada URL
- *TRACE*: servidor simplesmente devolve a requisição recebida
  - Útil para depuração e diagnóstico
- Todas as requisições podem ser interceptadas por servidores proxy
  - Respostas anteriores para *GET* e *HEAD* podem ser mantidas em *cache*

Protocolo HTTP: conteúdo das mensagens (I)

- Mensagem de requisição:

Método	*URL ou Caminho*	*Versão*	*Cabeçalhos*	*Corpo*
GET	//www.dcs.qmw.ac.uk/index.html	HTTP/ 1.1		

  - Servidores proxy exigem a URL completa para o recurso
  - Parte do cabeçalho é usada para impor condições sobre o recurso (estado, tipo, etc) ou prover informações sobre o cliente (credenciais, direitos de acesso, etc)
  - O corpo da mensagem não é necessário quando o recurso apenas referencia dados a serem obtidos do servidor

Protocolo HTTP: conteúdo das mensagens (II)

- Mensagem de resposta:

Versão	*Código de status*	*Motivo*	*Cabeçalhos*	*Corpo*
HTTP/ 1.1	200	OK		dados do recurso

  - Código de status e motivo reportam sobre o sucesso (ou fracasso) decorrente da requisição
    - Código é um inteiro de três dígitos (interpretado por programas)
    - Motivo é uma frase textual (interpretada por humanos)
  - Cabeçalhos são usados para passar informações adicionais sobre o servidor, ou sobre o próprio recurso, para o cliente (necessidade de credenciais de acesso, algoritmo de compactação, URL para re-direcionamento, etc)
  - Corpo contém os dados associados com a URL especificada
    - Inclui seus próprios cabeçalhos indicando tipo, formato de compressão, etc

Agenda

- API para comunicação entre processos
- Representação de dados externos
- Comunicação cliente-servidor
- Comunicação grupal

Comunicação grupal

- A comunicação par-a-par pode não ser o melhor modelo de comunicação quando a implementação de um serviço envolve múltiplos processos, possivelmente executando em computadores distintos
  - Maior tolerância a falhas e/ou maior disponibilidade
  - Necessidade de comunicação com um grupo de processos
- Nesses casos, o mais apropriado é utilizar um mecanismo de difusão seletiva (*multicast*) de mensagens
  - Envio de uma única mensagem de um processo para cada um dos membros de um grupo de processos de destino
  - Membros do grupo podem não ter conhecimento uns dos outros nem ser conhecidos pelo remetente
  - Várias possibilidades quanto ao comportamento do mecanismo de difusão:
    - Garantia de entrega
    - Ordem de recebimento
    - Atomicidade

Cenários de usos para comunicação grupal

- Recursos replicados
  - Solicitação de recursos
    - Cliente envia requisição de serviços para todos os membros do grupo de servidores
  - Garantia de consistência
    - Atualizações feitas em uma cópia do recurso devem ser difundidas para todas as outras réplicas
- Descoberta automática de serviços em redes móveis *ad hoc*
  - Mensagens de difusão podem ser enviadas por servidores e clientes para tentar localizar serviços de busca e de registro de serviços
- Propagação de notificações de eventos
  - Difusão de eventos do sistema para um grupo de clientes ou servidores interessados

Comunicação grupal com IP Multicast (I)

- Variação do protocolo IP para difusão seletiva
- Implementado sobre o protocolo IP tradicional
  - Grupos de difusão seletiva especificados por endereços IP da classe D (início “1110” no IPv4)
- Processos podem se juntar a (ou sair de) um número arbitrário de grupos a qualquer momento
- Mensagens enviadas para um grupo são automaticamente repassadas a todos os seus membros
  - Tamanho e identidade dos membros do grupo não são conhecidos pelo remetente
  - Remetente não precisa ser membro
- Implementado exclusivamente sobre UDP

Comunicação grupal com IP Multicast (II)

- Aspecto de roteamento
  - Em redes locais
    - Utiliza o mecanismo de difusão seletiva nativo da própria rede
  - Em redes de maior escala
    - Utiliza "roteadores de difusão seletiva" (*multicast routers*), que repassam os pacotes individualmente para roteadores de outras redes que contenham membros do grupo de destino
    - Parâmetro TTL (*time to live*) pode ser utilizado para restringir o número máximo de roteadores envolvidos na propagação dos pacotes
  - Roteamento de pacotes para o grupo feito em nível de rede pode ser muito mais eficiente do que se feito individualmente, para cada pacote, em nível de aplicação (*anycast*)
    - Exemplos?

Comunicação grupal com IP Multicast (III)

- Alocação de endereços:
  - Alguns endereços são alocados permanentemente pelas autoridades da Internet (faixa de 224.0.0.1 a 224.0.0.255)
    - Indisponíveis mesmo quando não possuem nenhum membro associado
  - Os outros endereços são alocados temporariamente e devem ser criados explicitamente antes de serem usados
    - Voltam a ficar disponíveis quando não há mais membros associados
    - Possibilidade de conflitos de endereços! (Como evitar?)
- Modelo de falha
  - Sujeito às mesmas limitações do protocolo UDP
    - Sem garantias quanto à entrega ou à ordem de chegada das mensagens
    - Possibilidade de entregas parciais (apenas parte dos membros do grupo poderá receber as mensagens)

API de Java para comunicação grupal com IP Multicast

- Classe *MulticastSocket*
  - Subclasse de *DatagramSocket*
- Principais métodos:
  - *joinGroup* - usado para incluir o *DatagramSocket* como membro de um determinado grupo de difusão (endereço IP Multicast + porta)
  - *leaveGroup* - usado para retirar o *DatagramSocket* de seu atual grupo de difusão
  - *setTimeToLive* – usado para ajustar o tempo de sobrevida para efeito de propagação dos pacotes

Exemplo de código (I)

```
import java.net.*;
import java.io.*;
public class MulticastPeer{
        public static void main(String args[]){
         // args contém o conteúdo da mensagem e o IP do grupo de destino (ex.: "228.5.6.7")
         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);


         // continua no próximo slide
```

Exemplo de código (II)

```
            // obtém mensagens de três outros membros do grupo
                    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();}
    }
}
```


Limitações da comunicação grupal com IP Multicast (I)

- Um ou mais membros podem descartar pacotes enviados ao grupo devido a estouro dos seus *buffers* locais
- Pacotes podem ser perdidos entre dois roteadores, fazendo com que todos os membros localizados além do segundo roteador não os recebam
  - O mesmo vale se algum dos dois roteadores falhar
- Alguns membros do grupo podem receber mensagens de um mesmo remetente em uma ordem diferente daquela recebida pelos outros membros
- Mensagens enviadas por dois destinatários diferentes podem não chegar exatamente na mesma ordem a todos os membros do grupo

Limitações da comunicação grupal com IP Multicast (II)

- Limitações ressaltam a necessidade de mecanismos de difusão de mensagens mais confiáveis
  - Exemplos:
    - Difusão confiável (*reliable multicast*)
    - Difusão totalmente ordenada (*totally ordered multicast*)
    - Difusão parcialmente ordenada (*partially ordered multicast*)
    - Difusão atômica (*atomic multicast*)
- Vários mecanismo de difusão confiáveis estão disponíveis na forma de *middleware*
  - Exemplos: ISIS, JGroups, Appia, Spread, JMS, WS-Notification, etc.
- Discutidos em mais detalhes no Capítulo 12 do livro

Exercícios recomendados

- No livro: 4.1-2, 4.7-8, 4.12, 4.18-23, 4.25-2