UNIVERSIDADE DE ÉVORA

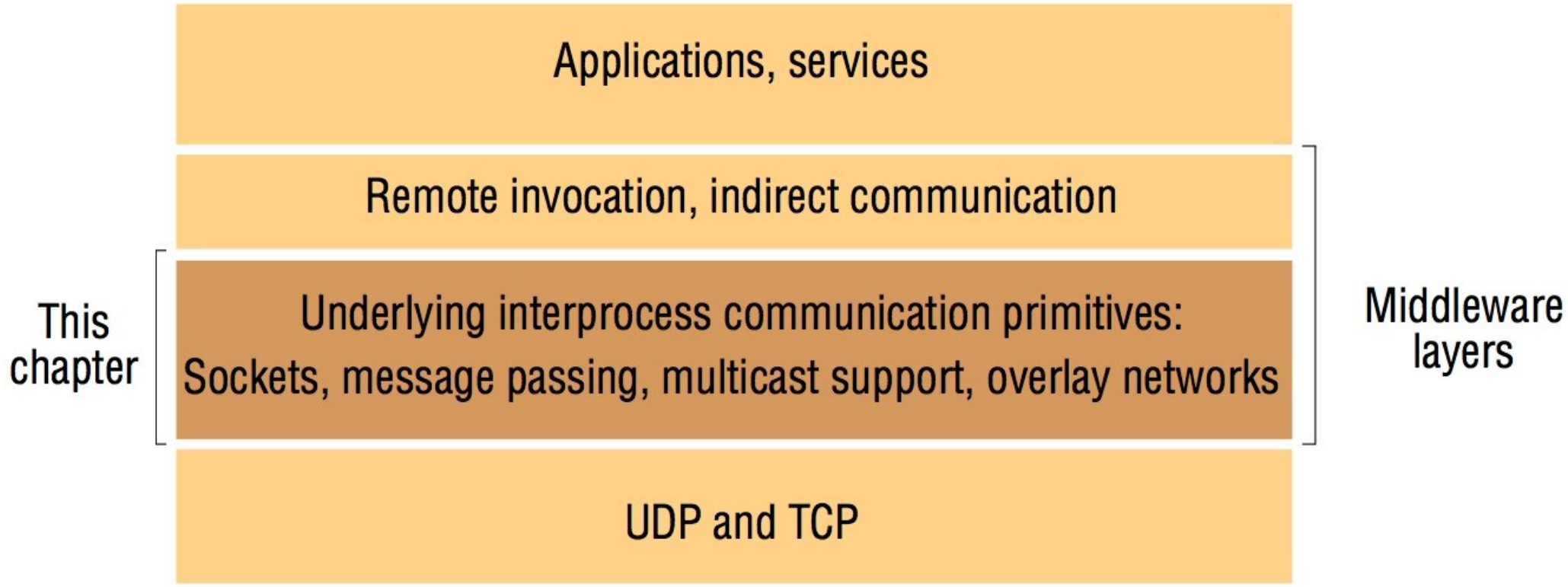
Sistemas Operativos II

Comunicação em Sistemas Distribuídos
camada inferior de *middleware*

Introdução

- Camada “inferior” do *Middleware*, relacionada com detalhes de
  - comunicação
  - representação externa de dados
  - *marshalling*
- Características de protocolos de comunicação entre processos num SD

Middleware: camada inferior



Protocolos de Internet

- recordar...
  - UDP
    - abstração para comunicação entre processos
    - permite o envio de mensagens isoladas, através de pacotes chamados datagramas
  - TCP
    - abstração para comunicação entre processos
    - fornece uma ligação (*stream*, canal) bidirecional entre dois processos
- Comunicação: **Operações** envolvidas uma mensagem:
  - send
  - receive

Protocolos de Internet

- Existe uma fila-de-espera (*buffer*) associada ao processo destinatário
  - envio: a mensagem é adicionada à fila remota
  - recepção: a mensagem é retirada da fila local

- Comunicação síncrona: os processos emissor e receptor sincronizam-se a cada mensagem
  - *send* e *receive* são operações bloqueantes
    - o emissor fica parado no *send* até que o *receive* seja efetuado
    - ao efetuar um *receive*, o receptor fica bloqueado até a mensagem chegar
- Comunicação assíncrona: não há sincronização
  - *send* não é bloqueante
    - o emissor prossegue assim que mensagem passa ao *buffer* local de saída
  - *receive* pode ser bloqueante ou não bloqueante*
    - * - o processo prossegue com um *buffer* que será preenchido *em background*, havendo uma notificação quando isso acontecer

Protocolos de Internet

- Destino da mensagem: (endereço, porto)
- Porto:
  - inteiro, identifica o destinatário da mensagem numa máquina
  - associado a um só destinatário mas usado por vários emissores
  - um processo pode receber mensagens de vários portos
  - um processo não partilha portos com outros processos na mesma máquina
  - uma máquina tem 2^{16} números para portos disponíveis
- Preocupações com o envio de mensagens:
  - fiabilidade: garantia de entrega das mensagens, ainda que alguns pacotes sejam perdidos
  - ordenação correta: garantia de que as mensagens são entregues pela ordem com que foram enviadas (e não há trocas)

Protocolos de Internet

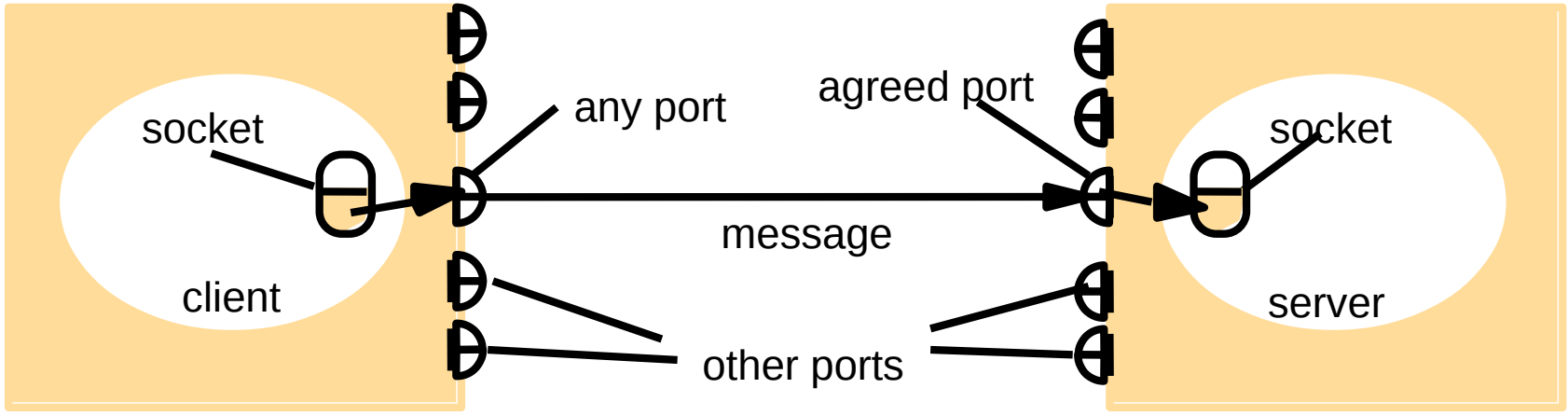
- O endereço do destinatário ou de um serviço:
  - endereço IP definido
  - obtido de um servidor de nomes (*binder*)
    - alguma transparência de localização

- UDP e TCP usam Sockets

(**socket**: abstração para o acesso ao canal de comunicação)

  - Comunicação entre processos
    - transmitir mensagens entre o *socket* de um processo e o *socket* de outro processo
  - Para um processo receber mensagens:
    - o *socket* tem de estar *bound* a um porto local e um endereço da máquina onde é executado

Sockets e Portos



UDP

- não há *acknowledgement* (confirmação) ou reenvios
- em caso de falha a mensagem não é recebida
- tamanho da mensagem:
  - IP: até 2^16 bytes (headers + mensagem) (muitos ambientes limitam 8K)
  - para mensagens maiores, as aplicações devem dividir a mensagem em vários pedaços (*chunks*)
- Para enviar ou receber
  - é necessário um socket associado (*bound)* a um porto local e endereço IP
- *send* não bloqueante; *receive*: bloqueante (com possível timeout)

- Modo *receive from any*
  - a origem da mensagem é desconhecida
  - o endereço e porto do emissor é detectado na operação *receive*

UDP

- Modelo de Falhas
  - falhas de omissão: algumas mensagens podem eventualmente perder-se (não chegam, ou têm erros de *checksum* ou falta de espaço no *buffer* de chegada)
  - ordenação: não há garantia de entrega pela ordem de envio
- Caberá às aplicações assegurar algumas garantias de fiabilidade, por exemplo através de confirmações (ack.)
- Utilização de UDP
  - por vezes o risco destas falhas pontuais pode ser tolerável/aceitável
  - vantagem: menos *overheads* relacionados
    - estado da informação na origem e destino
    - latência
  - exemplo de utilização: DNS

Cliente UDP

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

Servidor UDP em ciclo a atender clientes

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

TCP

- tamanho da mensagem: à responsabilidade da aplicação, tamanho arbitrário. O TCP envia (de modo transparente) os dados em blocos, desde o buffer de saída
- usa *acknowledgement* e *checksums*
  - garante a entrega desses blocos reenviando aqueles cuja confirmação não chega antes de um *timeout*
- controlo do fluxo de dados: se o emissor é mais rápido, então é bloqueado até que o receptor tenha consumido alguns dados
- elimina duplicados e garante a ordem correta na recepção (através de números de sequência em cada pacote IP)
- com conexão
- bloqueante para leitura; bloqueante no envio apenas se o controlo de fluxo atuar

TCP

- Modelo de Falhas
  - não é *reliable*/fiável, na medida em que não garante a entrega em todos os cenários (quando há dificuldades na rede, a ligação pode perder-se)
  - um processo não distingue entre falha na rede e falha no processo do outro lado da ligação
  - um processo não sabe se uma mensagem (recente) já foi ou não recebida (pode estar ainda a ser reenviada)

- Utilizações de TCP
  - HTTP, FTP, Telnet, SMTP

Cliente TCP: *connect, send, receive*

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

Servidor TCP: em ciclo, aceita ligações

```
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());}
    }
}

// this figure continues on the next slide
```

Servidor TCP: concorrente, serviço *echo*

```
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*/}}
        }
}
```

Repr. Externa de Dados e *Marshalling*

- Processos: estruturas de dados
- Mensagens na Comunicação: sequências de bytes
  - é necessário converter no envio e recepção
    - inteiros: *big-endian* (byte mais significativo primeiro, como *network byte order*), *little-endian*
- para uma estrutura de dados poder usar-se em RPC ou RMI
  - serializável, passível de ser representado de modo *flattened* e os tipos primitivos num formato acordado
- Representação Externa de Dados
  - formato usado para representação de estruturas e tipos primitivos
- *Marshalling*
  - tradução das estruturas e tipos primitivos para uma RED adequada para a transmissão
  - *unmarshalling*: processo inverso para reconstruir os dados à chegada

Repr. Externa de Dados e Marshalling

- Alternativas para RED:
  - CORBA CDR *(Common Object Request Broker Architecture CDR)*
  - Serialização Java – usada em RMI
    - https://docs.oracle.com/javase/tutorial/jndi/objects/serial.html
  - Sun XDR
  - XML
  - JavaScript Object Notation (JSON)
    - http://www.json.org/
  - *Protocol Buffers* (Google)
    - https://developers.google.com/protocol-buffers/

CORBA CDR (*Common Data Representation*)

- tipos primitivos (short, long, float... char, boolean, bit) #15 tipos
  - bytes transmitidos de acordo com o emissor (*big-endian* / *little-endian*), que é também especificada na mensagem

- tipos compostos:

Type	*Representation*
sequence	length (unsigned long) followed by elements in order
string	length (unsigned long) followed by characters in order (can also can have wide characters)
array	array elements in order (no length specified because it is fixed)
struct	in the order of declaration of the components
enumerated	unsigned long (the values are specified by the order declared)
union	type tag followed by the selected member

Mensagem CORBA CDR

- não é possível passar uma estrutura com ponteiros

- exemplo com uma Struct:

index in sequence of bytes	← 4 bytes →	*notes on representation*
0–3	`5`	*length of string*
4–7	`"Smit"`	*'Smith'*
8–11	`"h___"`	"padding"
12–15	`6`	*length of string*
16–19	`"Lond"`	*'London'*
20-23	`"on__"`	
24–27	`1934`	*unsigned long*

Representa uma Struct *Person* com os valores: {'Smith', 'London', 1934}

Marshalling CORBA

- Operações de Marshalling
  - geradas automaticamente a partir da especificação dos tipos de dados a transmitir na mensagem (argumentos, retorno)
- Tipos descritos com CORBA IDL (interface definition language)
- Assume-se que o emissor e receptor conhecem os tipos de cada elemento da mensagem, por isso o tipo não é passado (apenas o valor)

Java RMI

- objetos e tipos primitivos podem ser transmitidos
  - estes dados são transmitidos como argumento ou resultado de invocação remota de métodos
- formalmente, podem ser transmitidos (argumentos, retorno):
  - tipos primitivos
  - instâncias de classes que implementem a interface java.io.Serializable
- a aplicação que recebe a mensagem pode não conhecer o tipo dos dados
  - a representação serializada inclui informação sobre a classe do objeto (nome, versão/hash)

Serialização em Java

- Handle: referência para um objeto incluído na representação serializada

Serialized values			*Explanation*
Person	8-byte version number		h0	*class name, version number*
3	int year	java.lang.String name:	java.lang.String place:	*number, type and name of instance variables*
1934	5 Smith	6 London	h1	*values of instance variables*

h0 e h1 são *handles;* O valor das Strings é precedido pelo comprimento

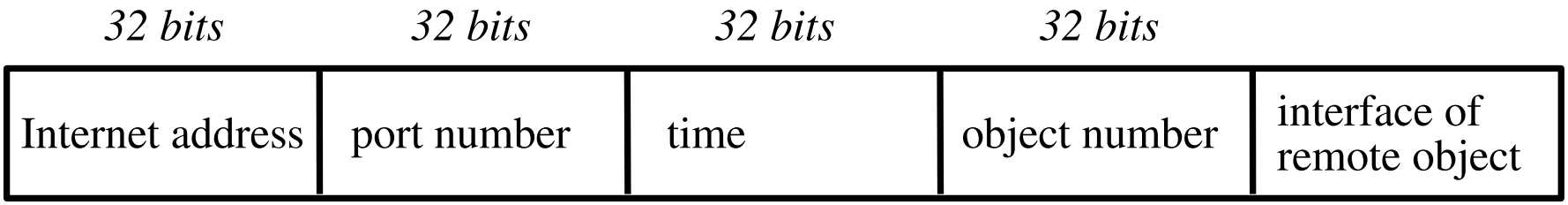
Cada objeto é escrito uma só vez, depois pode ser referido pelo seu Handle

Java Reflection

- Java Reflection
  - capacidade de descobrir propriedades de uma classe, como o nome e tipo de variáveis de instância ou métodos.
- Permite criar, obter e descobrir classes até então desconhecidas a partir do nome
- **`java.lang.reflect`**

Representação da Referência Remota de um Objeto

- Identificador para um objeto remoto válido no SD
- Cada objeto remoto tem uma única Ref. Remota
  - passada na mensagem do cliente que pede a invocação remota
- É importante garantir que a referência é única universalmente
  - junção de diversos elementos (data de criação...)



Serialização em XML

```
<person id="123456789">
        <name>Smith</name>
        <place>London</place>
        <year>1984</year>
        <!-- a comment -->
</person >
```

Serialização em XML: esquema XML com tipologia da estrutura

```
<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>
```

Serialização em JSON

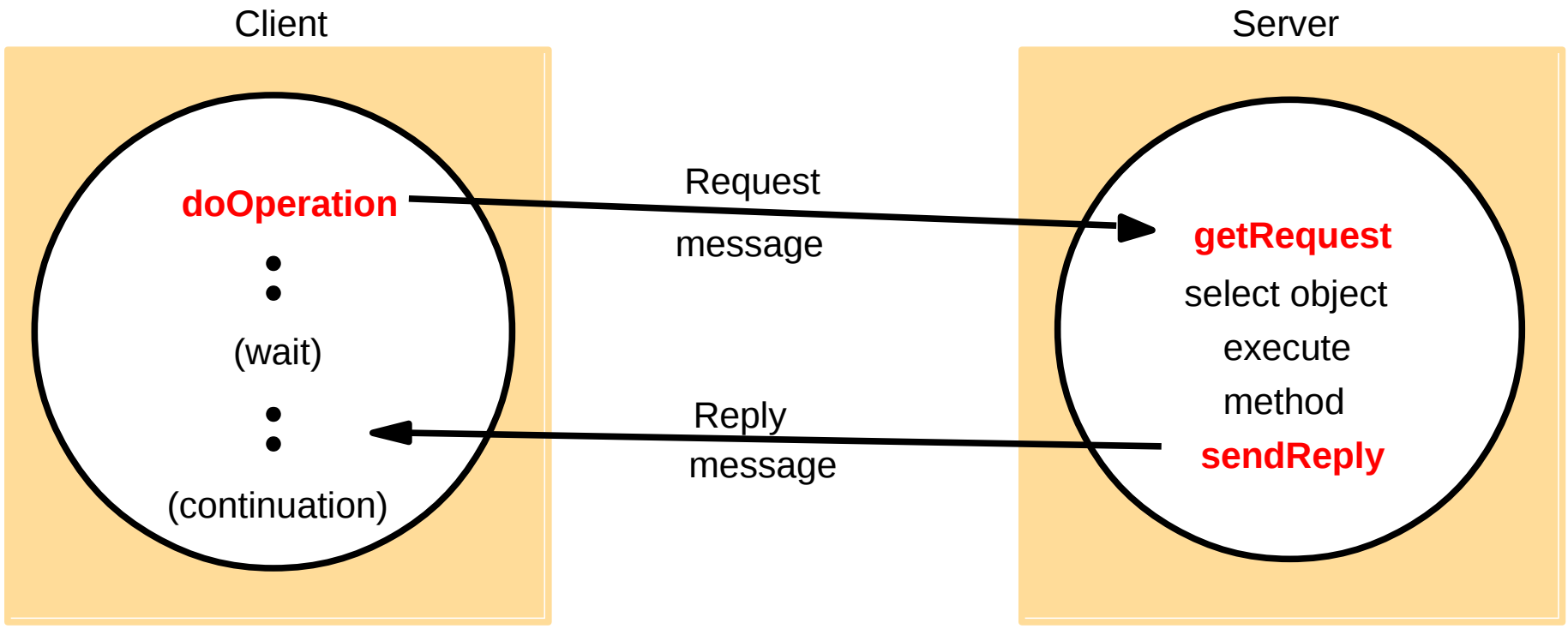
```
{"id": 123456789,
 "name": "Smith",
 "place": "London",
 "year": 1984 }
```

Comunicação Cliente-Servidor

- usualmente baseado em comunicação Request-Reply síncrona
  - pois a resposta do servidor comprova a recepção da mensagem
  - mas também pode ser assíncrona (se o cliente recolhe as respostas mais tarde)
- UDP ou TCP
- UDP evita *overhead*:
  - *ackowledgement* – aqui são redundantes (já temos o reply)
  - estabelecimento da conexão (mais 2 pares de mensagens)
  - controlo de fluxo é desnecessário na maioria dos casos (são passados argumentos e resultados de tamanho reduzido)

Protocolo *Request-Reply*

- assenta em 3 primitivas:



Operações no Protocolo *Request-Reply*

- Primitivas usadas em *Request-Reply*:

public byte[] doOperation (RemoteObjectRef o, int methodId, byte[] arguments)

envia a mensagem com um pedido ao objeto remoto e devolve a resposta.
Argumentos: o objeto remoto, o método a invocar e respetivos argumentos.

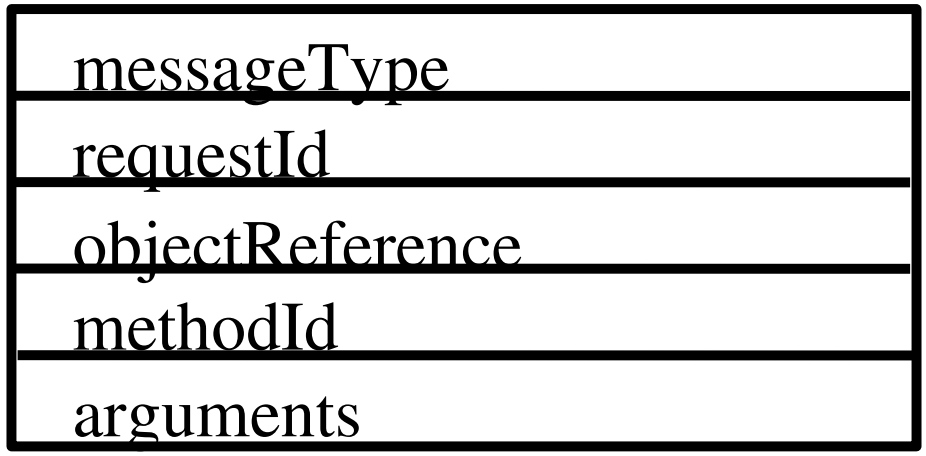
public byte[] getRequest ();

através de um porto no servidor, recebe o pedido do cliente

public void sendReply (byte[] reply, InetAddress clientHost, int clientPort);

envia a mensagem de resposta para o cliente, através do seu par (endereço,porto)

Estrutura da Mensagem - *Request-Reply*



Identificador da mensagem composto por:

- **requestId** (torna o ID único para o emissor)
- **identificador do processo emissor** (e.g. endereço e porto) (torna o ID único no sistema)

Modelo de Falhas para *Request-Reply*

- se implementado sobre UDP: falhas de comunicação
  - falhas de omissão (algumas mensagens podem perder-se)
  - não há garantia de entrega pela mesma ordem do envio
- Falhas relacionadas com os processos (paragem, crash)
- Como tornar o sistema com UDP + fiável (a nível da aplicação):
  - utilizar timeout na operação doOperation, para reenvio de pedido
  - descartar requests duplicados (servidor verifica o identificador)
  - tratar mensagens perdidas – caso o servidor já tenha enviado o *reply*, então o *request* duplicado é processado para enviar de novo a resposta (se a op. for idempotente, caso contrário deve usar-se uma tabela, designada Histórico – *ver adiante*)
  - uso de Histórico: tabela com os últimos resultados que permite o reenvio de respostas sem repetir processamento)
    - normalmente precisa apenas da última resposta para cada cliente

RPC

- protocolos usados em RPC:
  - request (R)
  - request-reply (RR)
  - request-reply-acknowledge reply (RRA)

- diferentes comportamentos perante falhas de comunicação
- a sua utilização depende do nº de mensagens necessárias

Name	*Messages sent by*		
	Client	*Server*	*Client*
R	*Request*		
RR	*Request*	*Reply*	
RRA	*Request*	*Reply*	*Acknowledge reply*

Request-Reply sobre TCP

- facilita a transmissão de argumentos e resultado de tamanho arbitrário
- mais fiável
- evita a necessidade de filtragem de duplicados e reenvios a nível do protocolo *request-reply*

- TCP facilita a implementação do protocolo RR

- Exemplo de protocolo RR:
  - HTTP

HTTP: request-reply

- Request

method	*URL or pathname*	*HTTP version*	*headers*	*message body*
GET	//www.dcs.qmw.ac.uk/index.html	HTTP/ 1.1		

- Reply

HTTP version	*status code*	*reason*	*headers*	*message body*
HTTP/1.1	200	OK		resource data

HTTP 1.1: ligações persistentes

•perduram durante várias séries de mensagens request-reply, para evitar o *overhead* do estabelecimento de uma nova ligação

Sockets para Datagramas e *Streams*

Sending a message

```
s = socket(AF_INET, SOCK_DGRAM, 0)
•
•
bind(s, ClientAddress)
•
•
sendto(s, "message", ServerAddress)
```

Receiving a message

```
s = socket(AF_INET, SOCK_DGRAM, 0)
•
•
bind(s, ServerAddress)
•
•
amount = recvfrom(s, buffer, from)
```

UDP

Requesting a connection

```
s = socket(AF_INET, SOCK_STREAM,0)
•
•
connect(s, ServerAddress)
•
•
write(s, "message", length)
```

Listening and accepting a connection

```
s = socket(AF_INET, SOCK_STREAM,0)
•
bind(s, ServerAddress);
listen(s,5);
•
sNew = accept(s, ClientAddress);
•
n = read(sNew, buffer, amount)
```

TCP

stream

Comunicação em Grupo

- *multicast operation*: operação que envia uma mensagem de um processo para cada um dos membros de um grupo de processos

- usualmente, a constituição do grupo é transparente para o emissor e não há garantias de entrega ou ordenação

- **Multicast** fornece uma útil infraestrutura para (construir) SD que incluam:
  - deteção de servidores em *spontaneous networking*
  - replicação de dados para melhor desempenho
  - tolerância a falhas baseada na replicação dos serviços
  - propagação de notificações de eventos

IP Multicast

- Multicast sobre Internet Protocol (IP)
  - pacotes IP são destinados a endereços (máquinas – sem portos (estes dizem respeito à camada de transporte))
- multicast group: definido por um endereço IPv4 classe D
  - Em particular na gama 224.0.0.0 a 239.255.255.255
    - Endereços geridos pela *Internet Assigned Numbers Authority*
- é possível enviar datagramas para um grupo sem ser membro
- a nível da programação, o IP Multicast pode usar-se apenas com UDP
  - os dados circulam como datagramas

- ao nível de IP, um computador pertence ao grupo multicast se um ou mais processos tem sockets associados ao grupo

IP Multicast

- Time To Live (TTL)
  - limita a distância de propagação de um datagrama *multicast*
  - indica o nº de *multicast routers* que o datagrama pode atravessar

- Endereços *Multicast*:
  - permanentes
    - existem mesmo quando não há membros no grupo
  - temporários
    - cessam quando não há membros

IP Multicast

- Modelo de Falhas no *multicast* de datagramas
  - falhas de omissão (inerentes ao UDP)
    - alguns membros podem não receber algumas mensagens
  - *unreliable multicast*: não garante a entrega de uma mensagem para cada membro do grupo
  - ordenação:
    - duas mensagens enviadas por processos diferentes podem não chegar pela ordem de envio
    - os pacotes IP de uma mensagem podem não chegar pela ordem de envio

Java API para IP Multicast: *join*, troca de datagramas com o grupo

```
import java.net.*;
import java.io.*;
public class MulticastPeer{
        public static void main(String args[]){
         // args give message contents & destination multicast group (e.g. "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);


         // this figure continued on the next slide
```

Java API para IP Multicast: *join*, troca de datagramas com o grupo

```
            // get messages from others in group
                    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();}
        }
    }
```

Comunicação entre Processos - UNIX

- as primitivas para IPC (comunicação entre processos) estão disponíveis em *system calls* em cima de UDP ou TCP
- destino da mensagem: socket address (internet address, port)
- um processo pode criar um socket para comunicar com outro processo, invocando a *system call* socket
- Comunicação via:
  - datagramas - UDP
  - *streams* - TCP

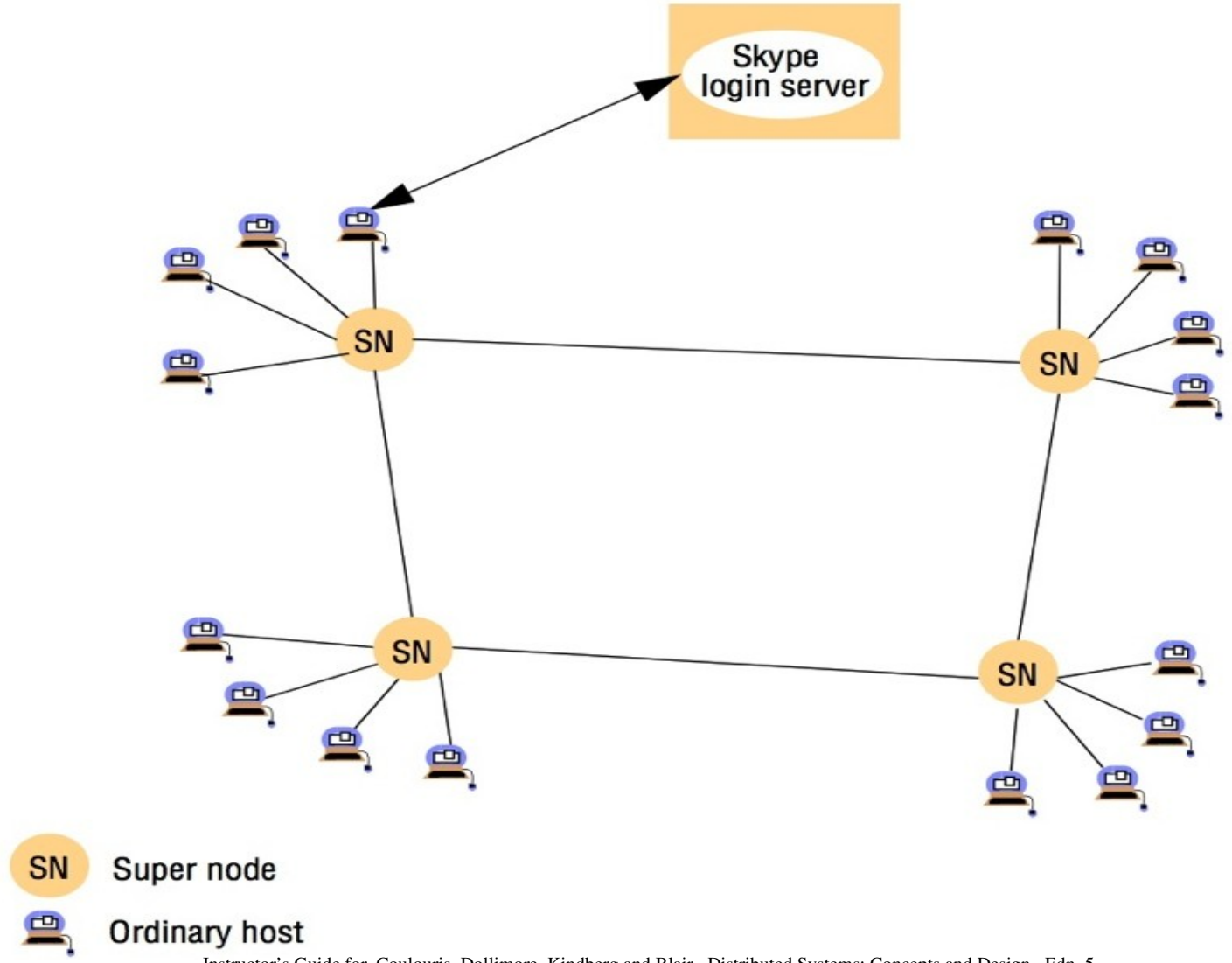
Redes Virtuais

- *Network Overlay / Virtual Network*
  - Rede virtual formada por um conjunto de nós e ligações lógicas/virtuais entre esses nós
    - Assenta numa rede convencional (como uma rede IP)
    - Esconde aspetos particulares dos segmentos em que assenta... fornece uma plataforma uniforme
    - Oferece serviços adicionais
      - Garantias adicionais de qualidade do serviço
      - Forma de encaminhamento (routing) própria
      - Segurança
      - Outros, vocacionados para uma aplicação ou serviço específico
  - Possível Desvantagem
    - Eventual perda de desempenho por existir mais uma camada

Caso de estudo: Skype

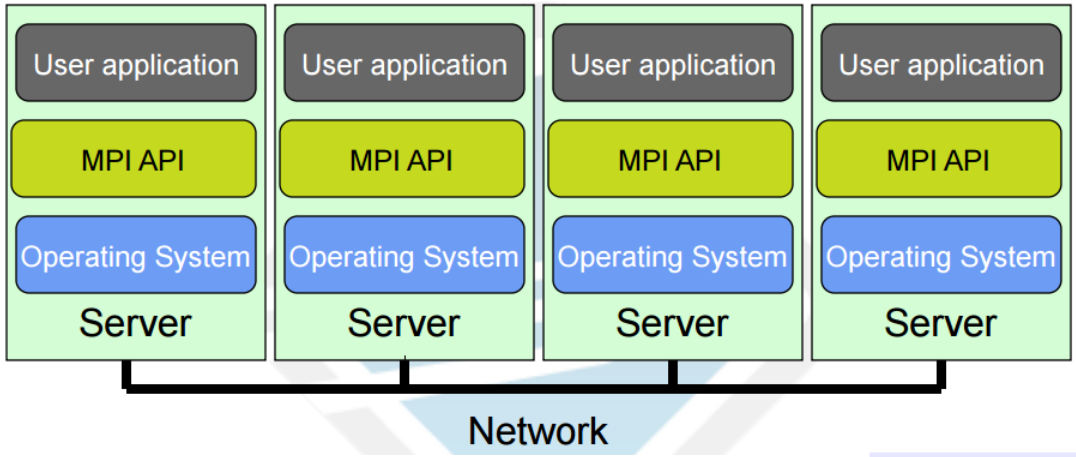
- Skype: aplicação Peer-to-Peer
  - de VoIP, IM, interfaces para serviço telefónico normal
  - Network overlay system com perto de 400 milhões de utilizadores
- Arquitetura: *Peer-to-peer...* com
  - Hosts cliente
  - Super nodes: hosts com capacidade suficiente
    - largura de banda, acessibilidade e/ou tempo de ligação ativa
    - endereço público de acessibilidade global
    - capacidade de processamento
- Funcionalidades importantes
  - Pesquisa de utilizadores
    - Envolve a consulta de 8 super nodes, em média
  - Chamadas de voz (TCP para negociar; UDP ou TCP para streaming)

Skype Network Overlay



MPI

- *Message Passing Interface* (MPI)
  - Mecanismo para comunicação entre processos baseado nas primitivas send/receive, usado em programação distribuída, para computação de alto desempenho



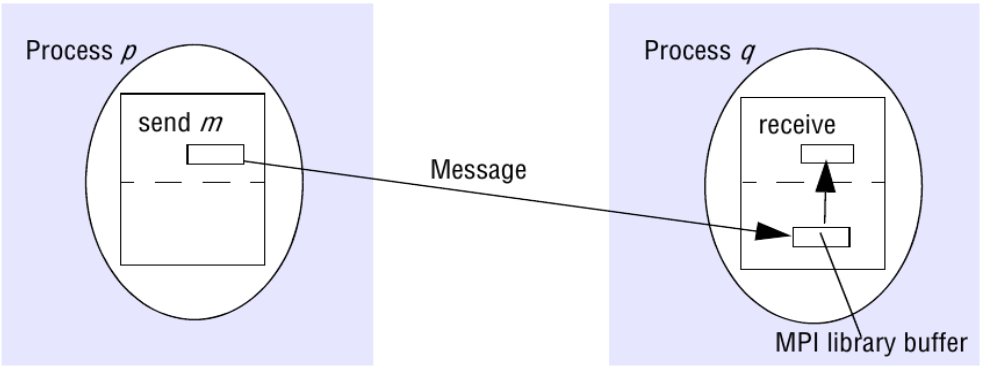


Imagem:
https://www.open-mpi.org/video/general/what-is-[open]-mpi-1up.pdf

MPI

Send operations	*Blocking*	*Non-blocking*
Generic	*MPI_Send*: the sender blocks until it is safe to return – that is, until the message is in transit or delivered and the sender's application buffer can therefore be reused.	*MPI_Isend*: the call returns immediately and the programmer is given a communication request handle, which can then be used to check the progress of the call via *MPI_Wait* or *MPI_Test*.
Synchronous	*MPI_Ssend*: the sender and receiver synchronize and the call only returns when the message has been delivered at the receiving end.	*MPI_Issend*: as with *MPI_Isend*, but with *MPI_Wait* and *MPI_Test* indicating whether the message has been delivered at the receive end.
Buffered	*MPI_Bsend*: the sender explicitly allocates an MPI buffer library (using a separate *MPI_Buffer_attach* call) and the call returns when the data is successfully copied into this buffer.	*MPI_Ibsend*: as with *MPI_Isend* but with *MPI_Wait* and *MPI_Test* indicating whether the message has been copied into the sender's MPI buffer and hence is in transit.
Ready	*MPI_Rsend*: the call returns when the sender's application buffer can be reused (as with *MPI_Send*), but the programmer is also indicating to the library that the receiver is ready to receive the message, resulting in potential optimization of the underlying implementation.	*MPI_Irsend*: the effect is as with *MPI_Isend*, but as with *MPI_Rsend*, the programmer is indicating to the underlying implementation that the receiver is guaranteed to be ready to receive (resulting in the same optimizations),