Capítulo III – Comunicação entre processos

From: Coulouris, Dollimore and Kindberg
Distributed Systems: Concepts and Design
Edition 4, © Addison-Wesley 2005

. Sockets UDP e TCP

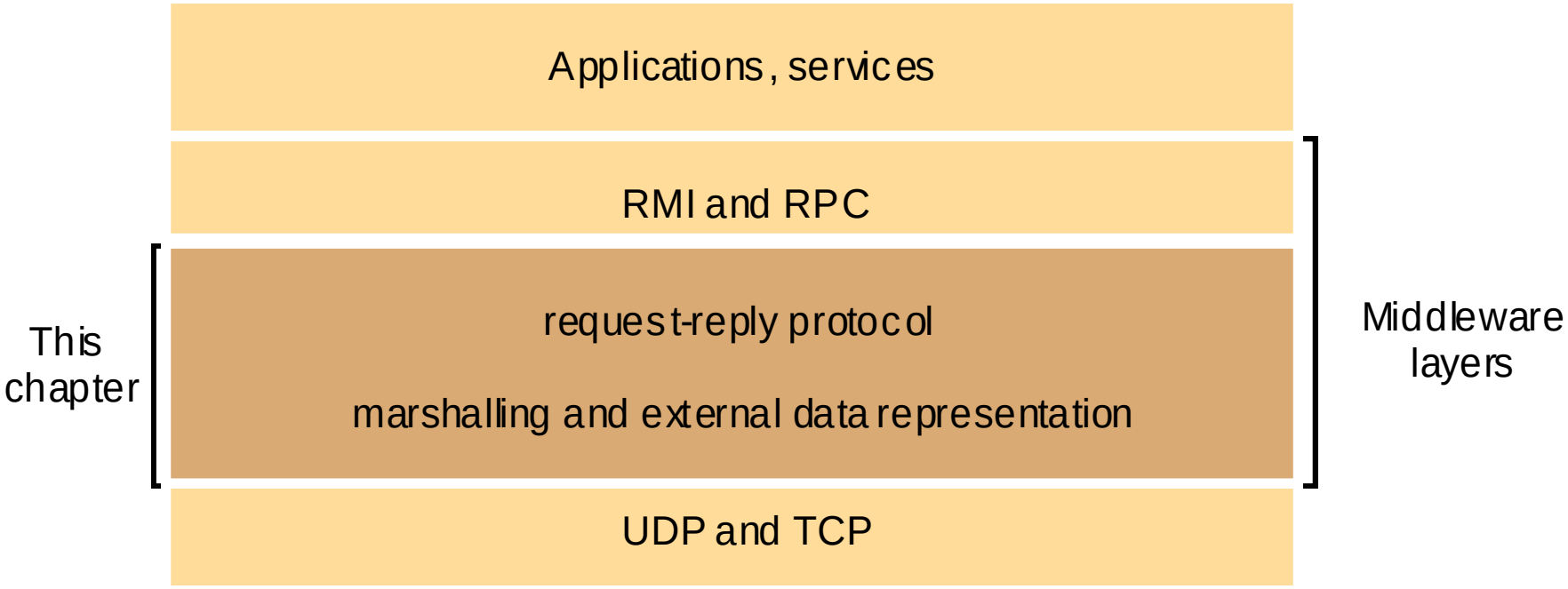
. A serialização de estruturas de dados

. Comunicação cliente-servidor

O protocolo pedido-resposta

Semântica perante falhas

Paula Prata,
Departamento de Informática da UBI
http://www.di.ubi.pt/~pprata

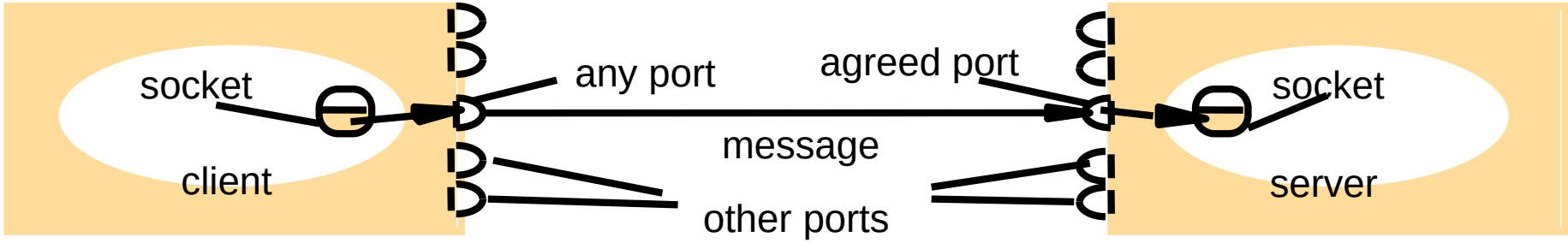

Applications, services
RMI and RPC
request-reply protocol
marshalling and external data representation
UDP and TCP
This chapter
Middleware layers

1 – Sockets UDP e TCP

(ideia surgida com o sistema UNIX de Berkeley -BSD Unix)

- Abstracção para representar a comunicação entre processos:

-a comunicação entre dois processos consiste na transmissão de uma mensagem de um socket num processo para um socket noutro processo.



. Nos protocolos internet, as mensagens são enviadas para um par:

- endereço internet
- nº de um porto

. O socket de um processo tem que ser conectado a um porto local para que possa começar a receber mensagens.

. Um vez criado tanto serve para receber como para enviar mensagens.

. O número de portos disponíveis por computador é 2^{16} (= 65536)

. Para receber mensagens, um processo pode usar vários portos simultaneamente, mas não pode partilhar um porto com outro processo diferente no mesmo computador.

(Excepção: processos que usem IP multicast)

. Cada socket é associado a um determinado protocolo, UDP ou TCP.

```
  public Cliente(){
   try {
     Socket sc = new Socket("127.0.0.1", 2222);
     System.out.println("Cliente cria socket    " + sc );
     …
Cliente cria socket    Socket[addr=/127.0.0.1,port=2222,localport=1533]

…

public Servidor() {
 try {
   ServerSocket ss = new ServerSocket(2222);
   while (true) {
     Socket s = ss.accept();
     System.out.println("Servidor - accept executado   " + s);
     System.out.println(s.getOutputStream());
     …
Processo servidor - accept executado  Socket[addr=/127.0.0.1,port=1533,localport=2222]
java.net.SocketOutputStream@45a877
```

Obter o endereço internet de uma máquina, em Java:

Classe InetAddress – representa os endereços IP

```
InetAddress    umaMaquina =
                        InetAddress.getByName ("penhas.di.ubi.pt");
System.out.println( umaMaquina.getHostAddress() );
```

193.136.66.27

```
InetAddress    umaMaquina =
                        InetAddress.getByName ("193.136.66.27");
System.out.println( umaMaquina.getHostName() );
```

penhas.di.ubi.pt

Principais protocolos de rede actuais:

UDP – User Datagram Protocol

. protocolo sem conexão

. comunicação por “datagrams”

TCP – Transmission Control Protocol

. protocolo com conexão

. comunicação por streams

Comunicação através do protocolo UDP

A comunicação entre dois processos é feita através dos métodos

send e *receive*.

-Um item de dados (“datagram”) é enviado por UDP sem confirmação (“acknowledgment”) nem reenvio.

- Qualquer processo que queira enviar ou receber uma mensagem tem que criar um socket com o IP da máquina local e o número de um porto local.

- O porto do servidor terá que ser conhecido pelos processos clientes.

- O cliente pode usar qualquer porto local para conectar o seu socket.

Comunicação através do protocolo UDP (cont ...)

- O processo que invocar o método *receive* (cliente ou servidor) recebe o IP e o porto do processo que enviou a mensagem, juntamente com os dados da mensagem.

Tamanho da mensagem

. O receptor da mensagem, tem que definir um array (buffer) com dimensão suficiente para os dados da mensagem

. O protocolo permite mensagens até 2^{16} bytes.

. Mensagens maiores que o buffer definido, serão truncadas

Comunicação através do protocolo UDP (cont ...)

Operações bloqueáveis

send – não bloqueável

O processo retorna do send assim que a mensagem é enviada.

No destino, a mensagem é colocada na fila do socket respetivo.

Se nenhum processo estiver ligado ao socket, a mensagem é descartada.

Comunicação através do protocolo UDP (cont ...)

Operações bloqueáveis

receive – bloqueável

O processo que executa o receive, bloqueia até que consiga ler a mensagem para o buffer do processo.

Enquanto espera por uma mensagem o processo pode criar uma nova thread para executar outras tarefas.

Ao socket pode ser associado um timeout, findo o qual o receive desbloqueia.

Comunicação através do protocolo UDP (cont ...)

Modelo de Avarias

Tipo de avarias que podem ocorrer:

Avaria por omissão – a mensagem não chega porque,

. Buffer cheio local ou remotamente

. Erro de conteúdo - checksum error

Avaria de ordenamento – as mensagens chegam fora de ordem

Utilização do protocolo UDP:

. Aplicações onde são aceitáveis avarias de omissão

. Domain Naming Service - DNS

. Transmissão de imagem

...

Classe DatagramSocket em Java

. permite criar um socket na máquina local para o processo corrente

. construtor sem argumentos, usa o primeiro porto disponível

. construtor com argumentos especifica-se o nº do porto

. se o porto está a ser usado é gerada a excepção SocketException

...

Classe DatagramPacket em Java

Ao instanciar um DatagramPacket para enviar uma mensagem, usar o construtor com os parâmetros:

- Um array de bytes que contém a mensagem,

- O comprimento da mensagem

- O endereço Internet do socket destino

(objecto do tipo InetAddress)

- Nº do porto do socket destino

Classe DatagramPacket em Java

Ao instanciar um DatagramPacket para receber uma mensagem,
usar o construtor com os parâmetros:

- Referência de um buffer de memória para onde a mensagem será transferida,
- O comprimento desse buffer

A mensagem é colocada neste objecto do tipo DatagramPacket.

Para extrair os dados da mensagem usa-se o método

getData() da classe DatagramPacket

- Os métodos *getPort()* e *getAddress()* devolvem o nº do porto e o IP do processo emissor, respectivamente.

Exemplo de utilização de sockets UDP em Java:

. Um processo cliente envia uma mensagem para um nó remoto e recebe em resposta a mesma mensagem. A mensagem e o nome da máquina remota são passados como parâmetros do programa.

. O processo servidor fica à espera de mensagens no porto 6789.
Ao receber uma mensagem, extrai a mensagem e envia-a de volta para o cliente, para o IP e o porto recebidos.

Processo Cliente

```
import java.net.*;
import java.io.*;
public class UDPClient{

    public static void main(String args[]){
    // args vai conter o conteúdo da mensagem e o nome do servidor
            DatagramSocket aSocket = null;
            try {
                // cria um socket para o processo cliente ligando-o a um porto disponível
                aSocket = new DatagramSocket();
                byte [] m = args[0].getBytes();
                InetAddress aHost =  InetAddress.getByName(args[1]);
                int serverPort = 6789;

                //criar o datagrama para envio
                 DatagramPacket request =
                        new  DatagramPacket(m,  args[0].length(), aHost, serverPort);

                // envia a mensagem
                aSocket.send(request);
```

Processo Cliente (cont ...)

```
// prepara o cliente para receber resposta do servidor
byte[] buffer = new   byte[1000];
DatagramPacket reply = new DatagramPacket(buffer, buffer.length);

// recebe resposta
aSocket.receive(reply);

System.out.println("Reply: " + 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();}

} }
```

Processo Servidor

```
import java.net.*;
import java.io.*;
public class UDPServer{
        public static void main(String args[]){
        DatagramSocket aSocket = null;
           try{

             //cria um objecto do tipo socket e liga-o ao porto 6789
                aSocket = new DatagramSocket(6789);

                // buffer de recepção vazio
                byte[] buffer = new byte[1000];

                while(true){
                   // instancia o objecto onde vai receber a msg
                   DatagramPacket request =
                                new DatagramPacket(buffer, buffer.length);

                   // bloqueia até receber a mensagem
                  aSocket.receive(request);
```

Processo Servidor (cont ...)

```
            // instancia o objecto para enviar a mensagem
            DatagramPacket reply =
                    new DatagramPacket(request.getData(), request.getLength(),
                                       request.getAddress(), request.getPort());

           // envia a resposta ao cliente
                 aSocket.send(reply);

           } // fim do while

        }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 através do protocolo TCP

Utilização da abstracção stream *para ler/escrever dados.*

Tamanho das mensagens:

- A aplicação é que decide quantos bytes devem ser enviados ou lidos da stream, sem a preocupação do tamanho máximo de pacotes

Perda de Mensagens:

- O protocolo TPC usa um esquema de confirmação de recepção das mensagens. Se necessário retransmite a mensagem.

Comunicação através do protocolo TCP (cont ...)

Controlo do fluxo de execução:

- O TPC tenta uniformizar as velocidades dos processos que lêem e escrevem de/numa stream.
Se “quem” escreve é muito mais rápido do que “quem” lê, então o processo que escreve é bloqueado até que o outro processo leia dados suficientes

Ordenação e duplicação de mensagens:

- Identificadores de mensagens são associados com cada pacote de dados, permitindo ao receptor detectar e rejeitar mensagens duplicadas ou reordenar mensagens fora de ordem.

Comunicação através do protocolo TCP (cont ...)

Destino das mensagens:

- Um par de processos estabelece uma conexão antes de poderem comunicar por uma stream. A partir dessa ligação, podem comunicar sem terem de indicar o endereço IP nem o nº de porto.

Modelo de comunicação:

Quando dois processos tentam estabelecer uma ligação através de Sockets TCP, um dos processos desempenha o papel de cliente e outro de servidor. Depois de estabelecida a ligação podem comportar-se como processos pares.

Cliente:

Cria um objecto do tipo Socket que tenta estabelecer uma ligação com um porto de um servidor, numa máquina remota. Para estabelecer esta ligação é necessário indicar o endereço IP e o porto da máquina remota.

Comunicação através do protocolo TCP (cont ...)

Servidor:

Cria um objecto do tipo "listening" socket associado ao porto servidor. Este socket possui um método que fica bloqueado até que receba um pedido de ligação ao porto correspondente.

Quando chega o pedido de ligação, o servidor aceita-a instanciando um novo socket que, tal como o socket do cliente, tem duas streams associadas, uma para saída outra para entrada de dados.

Classes em Java

Socket, ServerSocket - ver capítulo 2 (ponto 2)

Comunicação através do protocolo TCP (cont ...)

Modelo de avarias

Streams TCP usam

- checksums para detectar e rejeitar pacotes corrompidos;

- timeouts e retransmissão para lidar com pacotes perdidos;

- número de sequência para detectar e rejeitar pacotes duplicados;

Comunicação através do protocolo TCP (cont ...)

Modelo de avarias

Se uma mensagem não chega porque o sistema está congestionado,

o sistema não recebe a confirmação da recepção da mensagem, reenvia sucessivamente a mensagem até que a conexão é cancelada após um certo tempo.

A mensagem não é transmitida, os processos participantes ficam sem saber o que aconteceu!

falha na rede? falha do outro processo?

Utilização do protocolo TCP

. Os serviços HTTP, FTP, Telnet, SMTP, ...

2 - A serialização de estruturas de dados

Tanto o processo local como o processo remoto manipulam estruturas de dados locais.

2 - A serialização de estruturas de dados

Para a transmissão de dados numa mensagem, é necessário serializar esses dados em sequências de bytes.

Do outro lado, os dados têm que ser reestruturados de forma a representarem a informação original mesmo que a arquitectura da máquina do processo receptor seja diferente da arquitectura do emissor.

Serialização (cont ...)

Exemplos de diferença de formatos consoante a arquitectura

- Valores inteiros podem ser representados com o bit mais significativo em primeiro lugar, i.é, endereço mais baixo, (big-endian) – mainframe IBM

ou com

o bit mais significativo no fim (little-endian) – processadores intel

- Valores reais, formato IEEE574 – processadores intel
formato BDC – processador mainframe da IBM

- Valores carácter, um char, 1 byte – Unix

um char, 2 bytes - Unicode

A heterogeneidade do hardware obriga à utilização de formatos neutros de serialização.

Serialização (cont ...)

Duas formas de permitir que quaisquer computadores diferentes troquem valores:

1 - Ter uma representação externa comum para os dois.

Os valores são convertidos para a representação externa, e depois, no receptor, são convertidos para o formato do receptor.
Se os dois computadores são iguais poderá omitir-se a conversão.

2 - Não ter a representação externa, mas junto com os dados é enviada informação sobre o formato usado, de forma a que o receptor possa converter os valores se necessário

Serialização (cont ...)

- Na implementação de RPC ("Remote Procedure Calling") e de RMI ("Remote Method Invocation") qualquer tipo de dados que possa ser passado como argumento ou devolvido como resultado tem que poder ser serializado.

- Um standard definido para a representação de estruturas de dados e dos tipos primitivos de dados denomina-se uma representação externa de dados ("External Data Representation"):

Os formatos podem ser binários (e.g. Sun XDR - **RFC 1832** , CDR - Corba)

- são compactos e eficientes em termos de processamento

Ou podem ser baseados em texto (ex. HTTP protocol)

- podem ser custosos pelo parsing, e pelo par de conversões nativo-texto, texto-nativo

Serialização (cont ...)

- O processo de transformar os dados do seu formato interno para uma representação externa que possa ser transmitida numa mensagem denomina-se "marshalling" (serialização).

- O processo inverso, de converter os dados da representação externa para o formato interno do receptor, reconstruindo as estruturas de dados, denomina-se "unmarshalling" (desserialização)

O middleware é que realiza o processo da marshalling e unmarshalling

Exemplos: ➔

Serialização (cont ...)

a) CORBA – Common Data Representation (CDR)

definido pela especificação CORBA 2.0, 1998

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

Serialização da estrutura Person {"Smith", "Londom", 1934 }

Serialização (cont ...)

OBS: O tipo de dados não foi transmitido?!!

- Não é necessário, porque tanto o emissor como o receptor já conhecem o tipo e a ordem porque os dados são enviados.
- Os argumentos dos métodos e o resultado são de um tipo conhecido à priori.

Marshalling em CORBA – é feito automaticamente quando se usa o compilador de IDL.

O programador especifica os serviços de um sistema distribuído através de uma linguagem de definição de interfaces

(IDL- interface definition language)

Serialização (cont ...)

IDL- interface definition language

Uma IDL descreve operações, com os respectivos parâmetros e resultado

Pode ser independente das linguagens que são utilizadas no código cliente e servidor

Nesse caso, são definidos mapeamentos da IDL em linguagens de programação passíveis de serem usadas.

Um compilador de IDL processa o ficheiro IDL e gera ficheiros a juntar ao código escrito pelo programador:

- Ficheiro com o stub do cliente

- Ficheiro com o skeleton do servidor ...

Serialização exemplo (cont ...)

b) Serialização de objectos em Java

Para que uma classe possa ser serializada é necessário que implemente
a interface Serializable

Os objectos dessa classe poderão ser utilizados para comunicação entre processos ou para serem armazenados por exemplo em ficheiros

Ao desserializar é suposto o processo não saber a que classe pertence o objecto que está a ser desserializado.

O nome da classe e um número de versão são adicionados na serialização.

Serialização exemplo Java (cont ...)

Objectos que contenham referências para outros objectos:

- o objecto referenciado é também serializado
- a cada referência é associado um número (**handle)**
- em posteriores serializações do mesmo objecto é usado o seu handle (economiza-se tempo e espaço)

Seja a classe,

```
public class Person implements Serializable {
    private String name;
    private String place;
    private int year;
   public Person (String aName, String aPlace, int aYear ){
        name = aName;
        place = aPlace;
        year = aYear;
    }  ...
```

Serialização exemplo Java (cont ...)

Seja o objecto

Person p = new Person ("Smith", "Londom", 1934);

Serialized values

Person	8-byte version number		h0
3	int year	java.lang.String name:	java.lang.String place:
1934	5 Smith	6 London	h1

Explanation

class name, v. number

number, type and name of instance variables

values of instance variables

h0 and h1 are handles

Para serializar o objecto p, usa-se o método

void writeObject(Object), da classe ObjectOutputStream

Serialização exemplo Java (cont ...)

Para desserializar,

Object readObject() da classe ObjectInputStream

Para escrever /ler num ficheiro

Usam-se as streams FileOutputStream e FileInputStream

Para escrever/ler de um Socket

Usam-se as streams de output e input associadas ao socket

Com o RMI a serialização e a desserialização são feitas pelo middleware

Referências para objectos remotos

Uma referência para um objecto remoto é um identificador de um objecto válido em todo o âmbito do sistema distribuído.

Seja um objecto remoto a que queremos aceder:

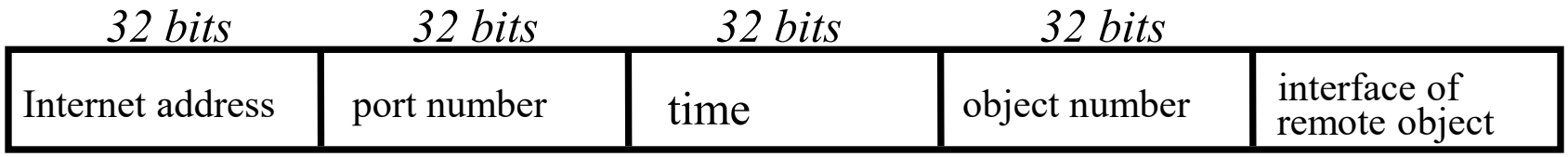
- a sua referência deverá existir no processo local,
na mensagem que enviamos ao objecto e
no processo remoto que possui a instância do objecto
cujo método estamos a invocar.

Referências para objectos remotos (cont ...)

- Referências remotas devem ser geradas de forma a garantir unicidade no espaço e no tempo:

Concatenando o endereço IP do computador com o nº do porto do processo que contém o objecto, com a data da sua criação e ainda com um nº sequencial para o objecto em questão.

Formato de uma referência para um objecto remoto:



3 – Comunicação cliente-servidor

É necessário um protocolo que, utilizando um mecanismo de transporte (e.g. TCP ou UDP), permita a conversação entre cliente e servidor

O protocolo pedido-resposta (request-reply protocol)

Usado pela maioria dos sistemas que suportam RPC e RMI

O protocolo para RMI é implementado usando três operações base:

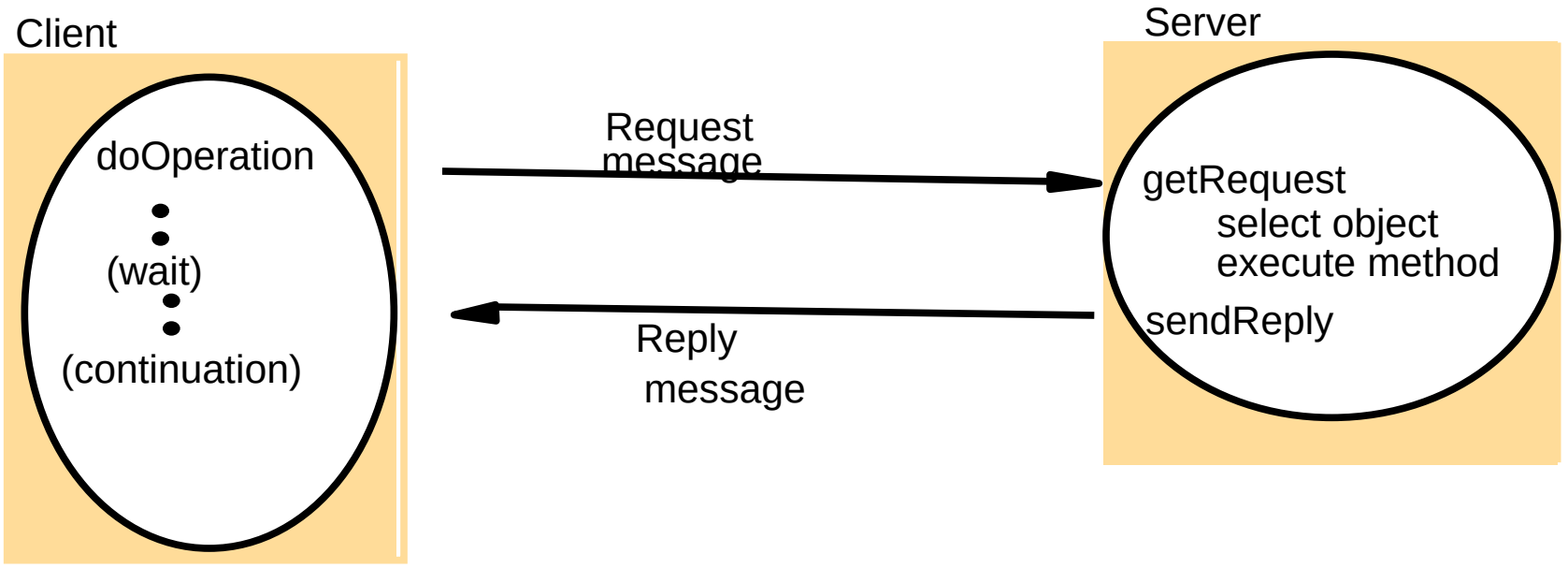
doOperation - activa um método remoto

getRequest - espera por pedidos de clientes

sendReplay - envia a resposta ao cliente

3 – Comunicação cliente-servidor

O protocolo pedido-resposta (cont ...)



Operações do protocolo pedido-resposta:

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

- Envia uma mensagem de pedido (request) a um objecto remoto e retorna uma resposta (reply),

- Os argumentos especificam o objecto remoto, o método a ser invocado e os argumentos desse método

Depois de enviar a mensagem o processo invoca uma operação de receive ficando bloqueado até à chegada da resposta

Operações do protocolo pedido-resposta:

public byte[] **getRequest** ();

Espera por pedidos de clientes.

- O processo servidor quando recebe um pedido de um cliente,

executa o método solicitado

e

envia a resposta correspondente activando o método sendReply

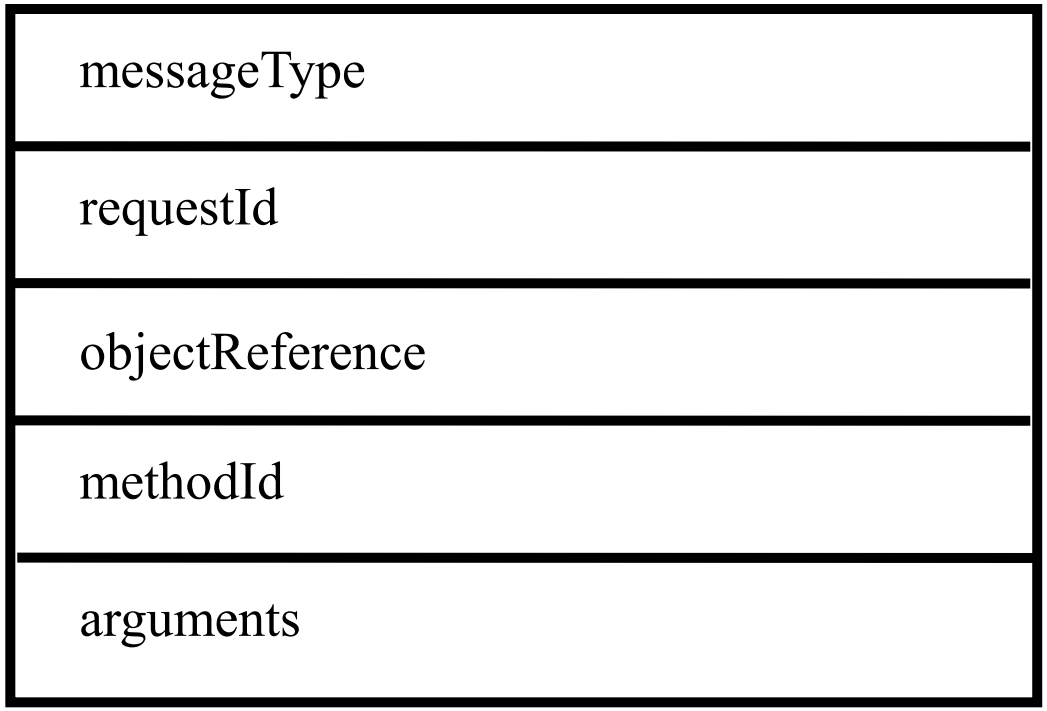
Operações do protocolo pedido-resposta:

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

- Envia a resposta ao cliente usando o seu endereço internet e o nº do porto.

- Quando o cliente recebe a resposta a operação doOperation é desbloqueada.

Estrutura das mensagens de pedido e resposta:



Estrutura das mensagens de pedido e resposta:

1º campo:

Tipo da mensagem: 0 se pedido, 1 se resposta

2ºcampo

Identificador único para cada mensagem do cliente,
o servidor copia-o e reenvia-o na resposta, permite ao cliente
verificar se a resposta diz respeito ao pedido que fez.

3º campo

Referência do objecto remoto (no formato do acetato 42)

4º campo

Identificador de qual dos métodos é invocado

5º campo

Argumentos do método invocado devidamente serializados

Para ser possível um sistema fiável de entrega de mensagens é necessário que cada mensagem tenha um identificador único.

Identificador de uma mensagem

requestId + número do porto (do emissor) + endereço IP (do emissor)

. requestId - inteiro, incrementado pelo cliente sempre que envia uma mensagem (quando atinge o valor inteiro máximo volta a zero) *(é único para o emissor)*

. o porto e o IP do emissor tornam o identificador da mensagem único no sistema distribuído.

(podem ser retirados da mensagens recebida no caso de a implementação ser em UDP)

Modelo de Avarias do protocolo pedido-resposta

Se as três operações são implementadas sobre UDP, então sofrem do mesmo tipo de avarias de comunicação que aquele protocolo:

. Avarias de omissão

. Mensagens não são garantidamente entregues por ordem.

Além disso, podem ocorrer avarias no processo:

- assumem-se “crash failures”, se o processo parar e permanece parado não produzindo valores arbitrários

Timeouts

- Para resolver o problema das mensagens perdidas,
a doOperation permite definir um serviço de tempo, timeout.

Após esgotar o timeout, a operação pode retornar com uma indicação de erro.

Geralmente, em vez de retornar, reenvia a mensagem várias vezes para ter a certeza que o problema foi “o fim” do servidor e não mensagens perdidas.

O que fazer com as mensagens de **pedido** repetidas !?

- O servidor, se estiver a funcionar, pode detectar repetições através do requestId.

Perda da mensagem de **resposta**

- Se a resposta se perdeu, o servidor ao receber novo pedido pode processar o método novamente (caso seja idempotente !)
e reenviar os resultados.

Operações Idempotentes – têm o mesmo efeito se executadas uma ou mais vezes.

Ex. Operação que adiciona um elemento a um conjunto

Contra-exemplo: adicionar um montante a uma conta bancária

Em vez de reexecutar o método, pode reenviar a mensagem, desde que esta fique armazenada num ficheiro que irá conter o registo do histórico do servidor. ➔

. Problema – consumo de memória

. Solução – o servidor ser capaz de identificar que o registo pode ser apagado do histórico, i.é, quando a resposta tiver sido recebida.

Como o cliente só pode enviar um pedido de cada vez, pode considerar-se que a recepção de um novo pedido é a confirmação da última resposta. !!

Dependendo do protocolo de transporte podem ser oferecidas diferentes garantias quanto ao número de execuções de um pedido ➔

Possíveis semânticas perante falhas

- Maybe (talvez) – perante possíveis perdas, não é repetido o pedido de invocação (em geral não é aceitável).

- At-least-once (pelo-menos-uma-vez) – perante perdas ou atrasos na resposta, são reenviados pedidos de execução que não são reconhecidos como duplicados pelo servidor
  (só deve ser usado em operações idempotentes).

- At-most-once (no máximo-uma-vez) – os possíveis reenvios de pedidos são reconhecidos como duplicados pelo servidor
  (é a semântica habitual dos sistemas de invocação remota)