O que é um sistema distribuído?

Conjunto de computadores ligados em rede, com software que permita a partilha de recursos e a coordenação de actividades, oferecendo idealmente um sistema integrado.

Características:
• Comunicação através de mensagens
• Concorrência
• Partilha de recursos
• Sistema Assíncrono
• Falhas Independentes
• Heterogeneidade

Características:
- Os componentes do sistema comunicam através de mensagens (não existem variáveis globais partilhadas) modelos de programação: cliente/servidor, modelo baseado em objectos

- Concorrência: os vários utilizadores utilizam o sistema em simultâneo (é necessário coordenar o acesso aos recursos partilhados: hw, sw, dados)

- Partilha de recursos impressoras, discos, ferramentas para trabalho cooperativo, bases de dados. A partilha de recursos levanta questões de segurança
Gestores de recursos controlam o acesso a recursos partilhados

- Sistema Assíncrono: não existe um relógio global, diferentes velocidades de processamento, não existe um limite para o tempo de comunicação

- Falhas Independentes: falhas na rede (perda de mensagens, duplicação, reordenação), falhas em unidades de processamento
-> a falha de um componente não impede necessariamente os outros de funcionar

- Heterogeneidade
Um sistema distribuído pode possuir:
diferentes tipos de rede .
diferentes tipos de hardware (diferentes representações de dados, diferente código máquina)
diferentes sistemas operativos (diferentes interfaces para os protocolos de comunicação)
diferentes linguagens de programação (diferentes representações de estruturas de dados como arrays ou registos,...)

Para tentar resolver o problema da heterogeneidade define-se uma camada de software intermédia: ***middleware***

O desenho e construção dos mecanismos de comunicação da Internet (protocolos Internet) permitiu que um programa em execução num qualquer ponto da rede possa enviar mensagens a programas em qualquer outro lugar.

Computação Móvel e Ubíqua

-Computadores portáteis .
-Telefones móveis .
-PDA's - Personal Digital Assistants .
-Máquinas fotográficas digitais .
-Pagers, etc
-wearable devices", relógios com processador" .
- (Sistemas embebidos (= computadores dentro de um produto, electrodomésticos, carros, ...

=> O termo **computação Ubíqua** pretende designar sistemas cuja utilização está de tal forma integrada na funcionalidade do produto que é transparente para o utilizador.
=> Maiores restrições em termos de: custo, tamanho, potência/autonomia

Outros Sistemas Distribuídos:

-Correio electrónico .
-World wide web .
-Sistemas de ficheiros distribuídos .

Aplicações críticas:
Reserva de bilhetes em companhias de transportes .
Comércio electrónico .
Máquinas Multibanco .
=> **Exigem fiabilidade e segurança**

Desafios na implementação de sistemas distribuídos:

• Escalabilidade
• Abertura
• Tolerância a Falhas
• Segurança
• Transparência

Escalabilidade
Capacidade de o sistema se manter a funcionar de forma correcta e à velocidade desejada independentemente do número de utilizadores.
É necessário:
- desenhar o software de forma a que o aumento de utilizadores não exija
grandes alterações
- evitar algoritmos e estruturas de dados centralizadas
(replicação de dados se necessário)
- controlar o aumento de custos devido à disponibilização de mais recursos
- controlar a perda de performance (replicação de serviços)
- evitar o transbordo de certos limites de recursos (ex. Endereço IP com 32 bits, insuficiente)

Sistema Aberto ("Openness")
Capacidade de o sistema ser extensível, quer em hardware quer em software
- novos componentes devem poder ser adicionados sem por em causa o funcionamento dos já existentes, e poder comunicar com eles

Para isso é importante que:
- sejam conhecidas as interfaces dos novos componentes através da publicação da sua documentação
- utilizar protocolos e formatos standard
Exemplo de publicação de interfaces:
Request For Comment (RFCs) www.ietf.org
Contém as especificações dos protocolos internet desde o início dos anos 8

Tolerância a Falhas
Tolerar uma falha significa conter os seus efeitos de forma a que o sistema continue a funcionar
Detecção da falha
Dados corrompidos (mensagens ou ficheiros) podem ser detectados através de somas de verificação.
- Localização da falha
Se não houve resposta a um pedido, o que significa?
- falha na rede
- falha no nó destino
como distinguir
Mascarar / Tolerar a falha
Algumas falhas podem ser ocultadas do utilizador se se utilizar redundância suficiente:
- quando uma mensagem não chega, pode ser retransmitida
- um ficheiro pode ser escrito em duplicado (um em cada disco)
- entre cada dois "routers" da internet, devem sempre existir dois percursos
- uma base de dados pode ser replicada em vários servidores

Segurança
Manter recursos computacionais seguros significa:
- Manter o nível de confidencialidade exigido pelos utilizadores
(protecção contra acessos não autorizados)
- Garantir a integridade dos dados
(protecção contra alteração ou corrupção de dados ou programas)
- Manter a disponibilidade do sistema
(protecção contra interferências com os meios de acesso aos recursos)
Alguns problemas por resolver:
- Ataques do tipo "negação de serviço" (denial of service)
- Segurança do código móvel

Transparência
O sistema deve ser visto como um todo e não como uma colecção de componentes distribuídos.
No standard de "Open Distributed Processing (ODP) foram definidos os seguintes tipos de transparência:
- Acesso
- Localização
- Concorrência
- Replicação

- Falhas
- Migração
- Desempenho
- Escalabilidade

Mais Importantes:
Acesso e Localização em conjunto são referidos como transparência de rede

Transparência de acesso:
permite que o acesso a recursos locais e a recursos remotos seja feito através das mesmas operações (i.é, usando a mesma interface)(ex. Samba ou WinSCP vs FTP)
Transparência de localização:
permite que os recursos possam ser acedidos sem o conhecimento da sua localização. (ex. Programas de correio electrónico)
Transparência de concorrência:
permite que os vários clientes de um componente não necessitem de ter em conta o acesso concorrente ao componente
Transparência de replicação:
permite que os clientes de um componente não se apercebam se existe replicação e estão a usar uma réplica e não o original; a utilização de várias instâncias de um componente pode ocorrer por razões de desempenho ou de fiabilidade; os utilizadores do componente não necessitem de saber que o componente possa ser replicado.
Transparência de Falhas:
permite que o sistema funcione na presença de falhas de hardware ou software sem que utilizadores e programadores saibam como as falhas foram ultrapassadas. (por ex. um sistema de e-mail pode retransmitir uma mensagem até que a mesma seja entregue com sucesso)
Transparência de Migração:
permite que um recurso possa mudar de localização sem que isso afecte a sua utilização.(ex. Telemóveis em movimento)
Transparência de Desempenho:
permite que o sistema seja reconfigurado para melhorar o seu desempenho sem que os utilizadores se apercebam.
Transparência de Escalabilidade:
permite que o sistema seja expandido sem que os utilizadores se apercebam de como isso foi conseguido.

Modelos de comunicação por mensagens

Modelos clássicos de cooperação de processos:

Sistemas de Memória Partilhada
- os processos acedem a um único espaço de endereçamento
- comunicação através de variáveis partilhadas
- sincronização feita pelas técnicas clássicas da programação concorrente
(ex. Semáforos ou Monitores)
Sistemas de Memória Distribuída
- vários espaços de endereçamento disjuntos
(cada processador tem a sua própria memória local
- comunicação por mensagens
(através de uma canal de comunicação)

Comunicação por mensagens
- Comunicação e sincronização integradas num único conceito
O programador utiliza os mecanismos de comunicação por mensagens sem se preocupar com a forma como é feito o armazenamento e a transferência dos dados

As várias formas de comunicação por mensagens distinguem-se por:

- **tipo de sincronização**
  - comunicação síncrona
  - comunicação assíncrona
  - invocação remota de procedimentos
- **forma como são especificados os vários intervenientes no processo**
  - identificação dos processos
  - criação dos processos (dinâmica ou estática)
  - comunicação bi ou uni-direccional

Três tipos de interacção:

Comunicação síncrona
o envio de uma mensagem é uma operação atómica que requer a participação de dois processos (emissor e receptor)
- se o emissor está pronto a enviar a mensagem mas o receptor não a pode receber, o emissor bloqueia.
- se o receptor está pronto a receber a mensagem mas o emissor não a envia, o receptor bloqueia

Em resumo:
o acto de comunicação sincroniza as sequências de execução dos dois processos
o 1º processo a chegar ao ponto da comunicação espera pelo segundo

Comunicação assíncrona
- o emissor pode enviar a mensagem e continuar a executar sem bloquear, independentemente do estado do processo receptor
- o receptor pode estar a executar quaisquer outras instruções, e mais tarde testar se tem mensagens a receber; quando aceita a mensagem não sabe o que se passa no emissor (pode até já ter terminado)

Comunicação síncrona vs assíncrona
(telefonar vs enviar uma carta)

c. síncrona – conceito de mais baixo nível (mais eficiente)
c. assíncrona – permite um maior grau de concorrência
– exige que o sistema de execução faça a gestão e o armazenamento das mensagens (um buffer de memória tem que estar preparado para armazenar um número de mensagens potencialmente ilimitado)

Invocação remota de procedimentos RPC (Remote Procedure Calling)

A comunicação entre dois processos diz-se uma chamada de procedimento remoto quando,

1 – um processo (o emissor) envia um mensagem
2 – o processo receptor produz uma resposta à mensagem e
3 – o emissor permanece suspenso até à recepção da resposta

Do ponto de vista do processo emissor (cliente) este mecanismo funciona como a chamada de um procedimento, esse procedimento não vai ser executado no próprio processo mas sim no receptor (servidor).
Os dados comunicados na mensagem, funcionam como parâmetros do tipo valor
A resposta do receptor pode ter a forma de parâmetros resultado

Invocação remota de procedimentos Variante do RPC

Emissor:
- após o envio da mensagem, o emissor prossegue a execução até que precise do resultado (permite maior concorrência)
- se, nesse ponto, a resposta ainda não estiver disponível, o emissor é suspenso

Receptor:
- quando um processo executa uma instrução de aceitação de mensagem, é suspenso até à chegada da mesma.

Pode ocorrer que o receptor pretenda:
- escolher uma de entre um conjunto de mensagens possíveis
- estabelecer condições para a recepção de uma mensagem

Para isso é necessário que exista uma instrução em que o receptor,
- selecciona uma de um conjunto de mensagens alternativas
- cada uma das quais pode ter associada uma condição para a aceitação da mensagem

Identificação dos processos

1) Sistema em que todos os processos têm um nome único o comando de envio pode nomear directamente o processo receptor

ENVIA <mensagem> PARA <nome do processo>
simetricamente no receptor
ESPERA <mensagem> DE <nome do processo>
requer que o receptor saiba o nome de todos os processos passíveis de lhe enviar uma mensagem

Se o receptor apenas estiver interessado em receber determinada mensagem, não importando quem é o emissor:
ESPERA <mensagem> *// o emissor é anónimo o receptor não*

2) Quando não é apropriado um sistema de nomes únicos para todos os processos definem-se entidades intermédias,

caixas de correio (ou canais)
conhecidas por ambos os intervenientes na comunicação

ENVIA <mensagem> PARA <caixa de correio>
ESPERA <mensagem> DE <caixa de correio>

Uma caixa de correio pode ter várias formas. Pode ser usada por:

• por vários emissores e vários receptores
• um emissor e vários receptores (difusão ou "broadcasting")
• vários emissores e um receptor
• um receptor e um emissor

Pode ainda ser estruturada para enviar informação em ambas as direcções ou apenas numa.
Ex.los

OCCAM – comunicação síncrona através de canais dedicados entre pares de processos
Ada – comunicação síncrona (RPC) em que um processo comunica com outro conhecendo o seu nome mas não divulgando a própria identidade
Linda – Comunicação assíncrona com difusão de mensagens que não contêm a identificação dos processos

Formas de criação de processos
Os processos de um programa distribuído podem ser criados de forma estática ou dinâmica

Definição estática: - todos os processos são criados no início da execução
- a atribuição de recursos (memória, canais de comunicação, etc)
é feita em tempo de compilação)
• mais eficiente

Definição dinâmica:
Permite maior flexibilidade:
- o sistema ajusta-se às necessidades da aplicação ao longo do tempo
- permite mecanismos de balanceamento de carga ("load balancing")

A criação estática de processos é apropriada para sistemas dedicados onde a configuração do sistema é conhecida à partida
Ex.los: - "embedded systems" - sistemas de monitorização médica , ...

Socket – interface de um canal de comunicação entre dois processos
Um par de processos comunica através de uma par de sockets

Paradigma de comunicação (análogo a usar um descritor de um ficheiro)
- criação ("open") do socket
- ler e escrever ("send" , "receive")
- destruição ("close") do socket

O endereço de um socket é especificado por:
- endereço internet (IP da máquina onde está o processo com que queremos comunicar)
- nº de um porto (um porto é representado por um inteiro >1024 que identifica os vários serviços em execução numa mesma máquina,
0-1023 reservados a utilizadores com privilégios root ou superuser)

Ex. lo 146.75.4.30/1234

A classe Socket
- permite criar sockets que comunicam através do protocolo TCP (Transmission Control Protocol) usando uma stream de bytes
- um dos sockets (o servidor) aguarda por um pedido de ligação enquanto o outro (o cliente) solicita a ligação
- após ser estabelecida a ligação entre os dois sockets, estes podem ser usados para transmitir dados nos dois sentidos

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

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

-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 IP permite mensagens até 2^16 bytes, (tamanho padrão - 8KB)
-Mensagens maiores que o buffer definido, serão truncadas

Operações bloqueáveis
send – não bloqueável

O processo retorna do send assim que a mensagem é enviada
No destino, é colocada na fila do socket respectivo
Se nenhum processo estiver ligado ao socket, a mensagem é descartada

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.

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*

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

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.

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.

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.

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.

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.

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

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!

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.

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

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.

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

- 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

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

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)

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 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 des-serializar é suposto o processo não saber a que classe pertence o objecto que está a ser des-serializado.

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

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;
}
```

Seja o objecto :
Person p = new Person (“Smith”, “Londom”, 1934);

Para serializar o objecto p, usa-se o método

void writeObject(Object), da classe ObjectOutputStream

Para des-serializar,

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

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

Operações do protocolo pedido-resposta:

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

- Envia uma mensagem de pedido (request) a um objecto remotoe 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

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

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:

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 38)

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 re-executar o método, pode reenviar a mensagem, desde que fique armazenada num ficheiro que faz 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)