

SISTEMAS DISTRIBUÍDOS

Capítulo 1

Introdução

NOTA PRÉVIA

A apresentação utiliza algumas das figuras do livro de base do curso

- G. Coulouris, J. Dollimore and T. Kindberg,
- Distributed Systems - Concepts and Design,
- Addison-Wesley, 5th Edition, 2011

ORGANIZAÇÃO DO CAPÍTULO

Definição, exemplos

Características essenciais dos sistemas distribuídos

Desafios: heterogeneidade, abertura, segurança, escala, falhas, concorrência, transparência

SISTEMAS DISTRIBUÍDOS ?

Exemplos?

- Serviços web
- Email
- Multibanco
- Etc.

O que é importante?

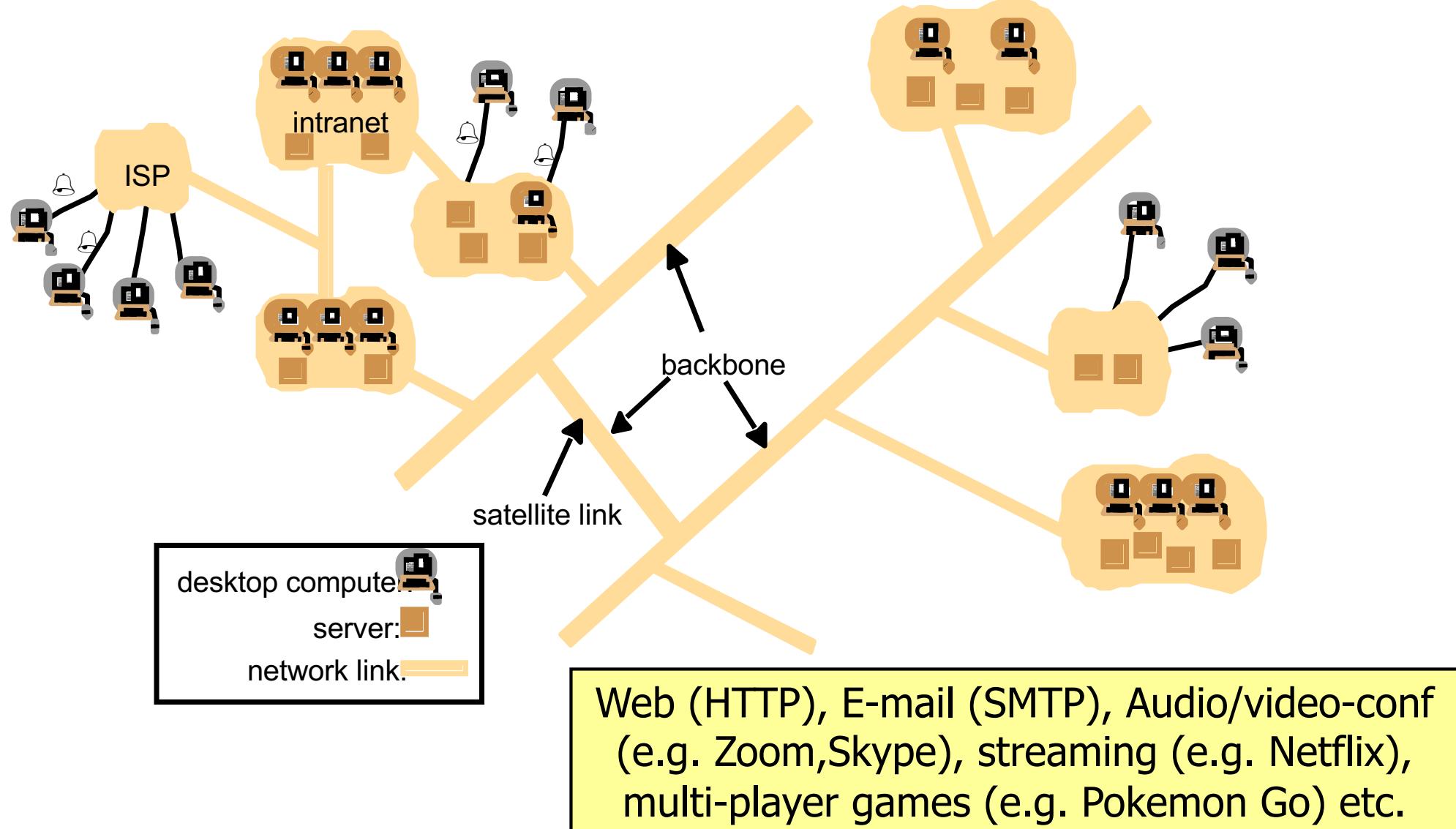
- Conjunto de nós / máquinas
- Utilização duma rede de comunicações para troca de mensagens

O QUE É UM SISTEMA DISTRIBUÍDO ?

Um sistema distribuído é um conjunto de componentes hardware e software interligados através de uma infra-estrutura de comunicações, que **cooperam e se coordenam entre si** apenas pela troca de mensagens, para execução de **aplicações distribuídas**

Assim, no âmbito desta cadeira, não estamos interessados nos sistemas que cooperam e se coordenam pela partilha de memória física comum (essa temática é abordada inicialmente na disciplina de Concorrência e Paralelismo)

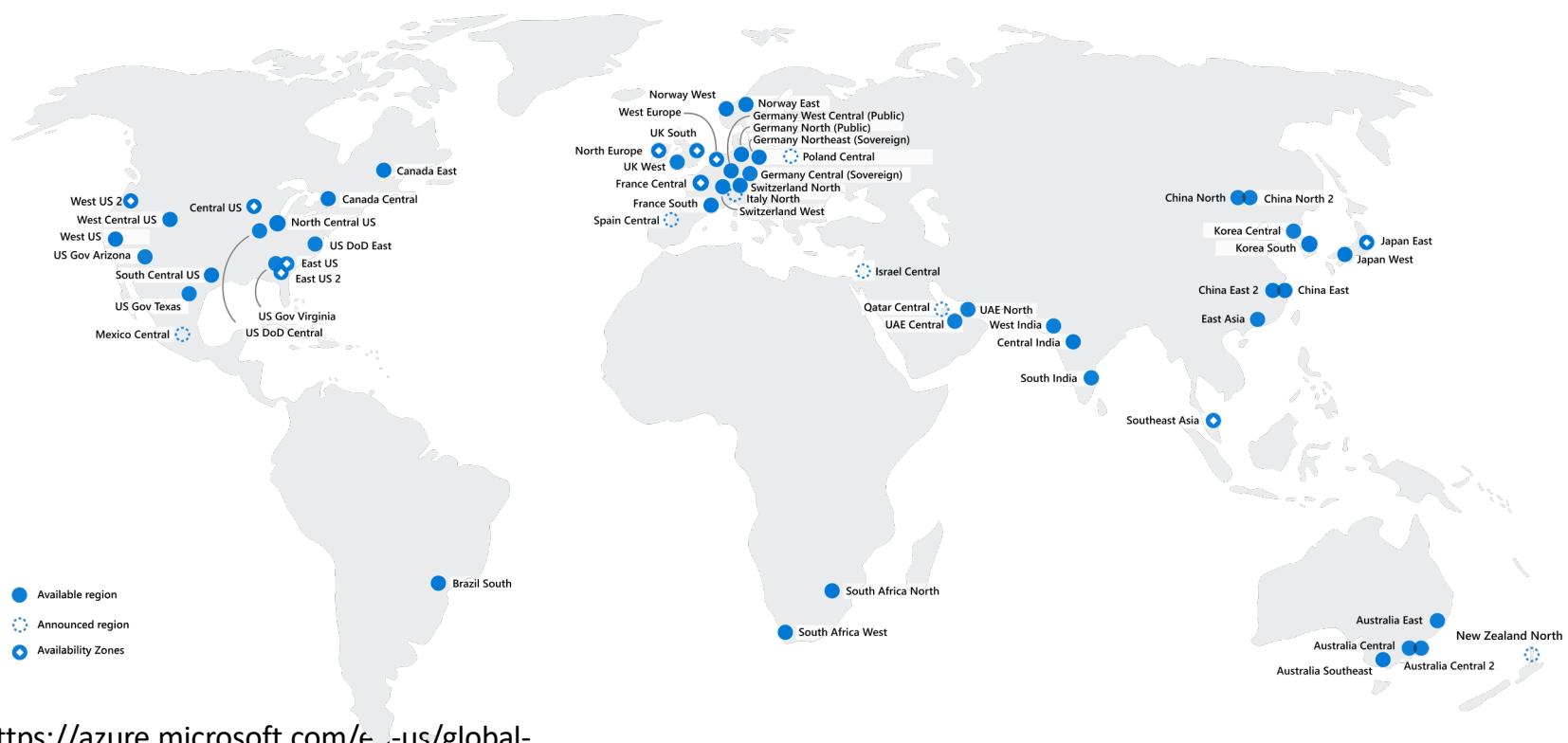
EXEMPLO: SERVIÇOS NA INTERNET



EXEMPLO: CLOUD COMPUTING

Diferentes tipos de serviço disponibilizados através duma infraestrutura distribuída globalmente.

- IaaS (Infrastructure as a Service) : e.g. AWS EC2 (máquinas virtuais)
- PaaS (Platform as a Service) : e.g. Google App Engine (app service)
- SaaS (Software as a Service) : e.g. Google Docs

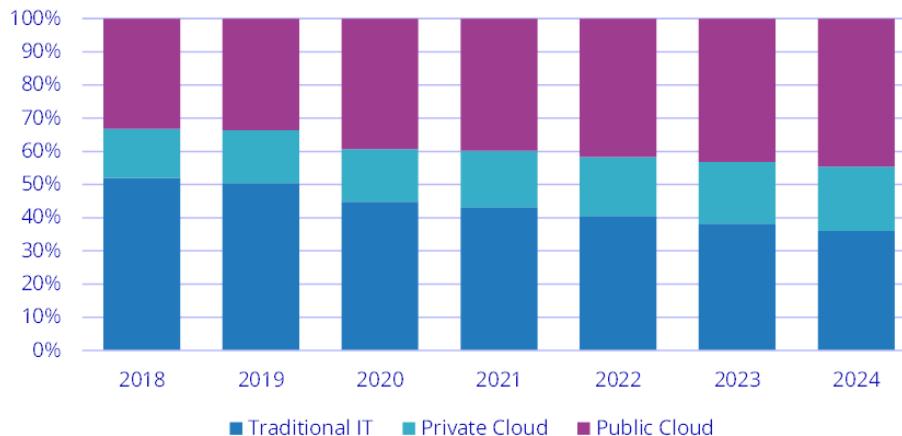


Azure locations: <https://azure.microsoft.com/en-us/global-infrastructure/geographies/> (accessed: Sep, 2020)

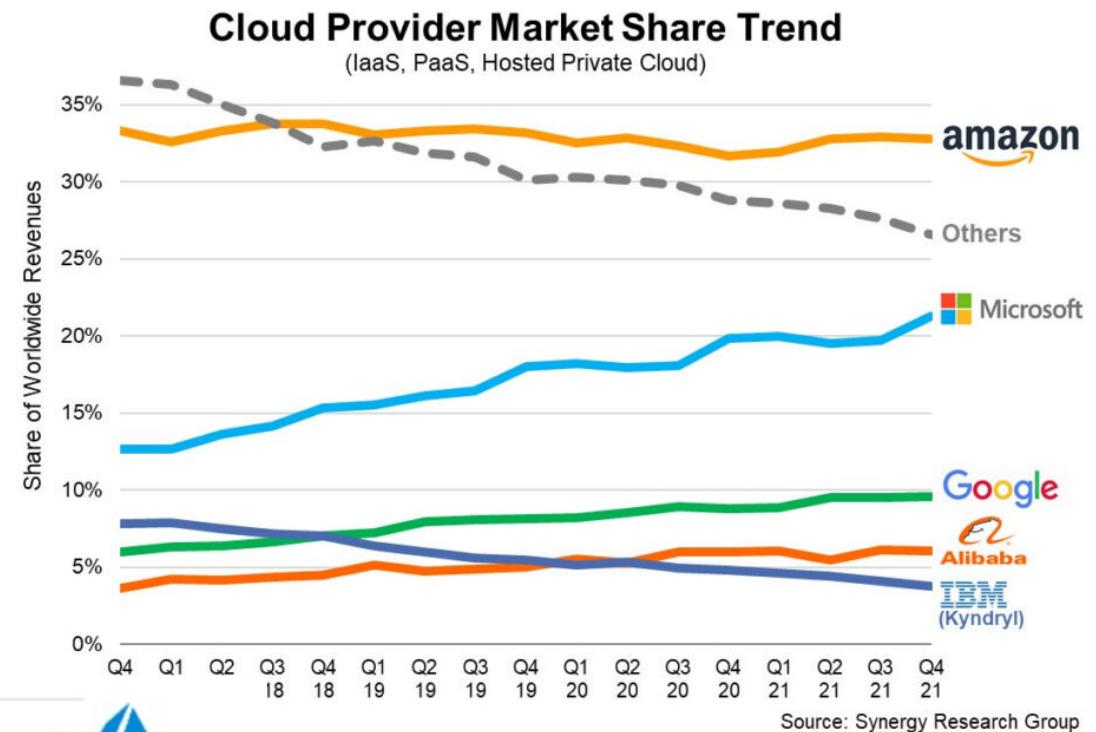
CLOUD COMPUTING: ALGUMAS TENDÊNCIAS



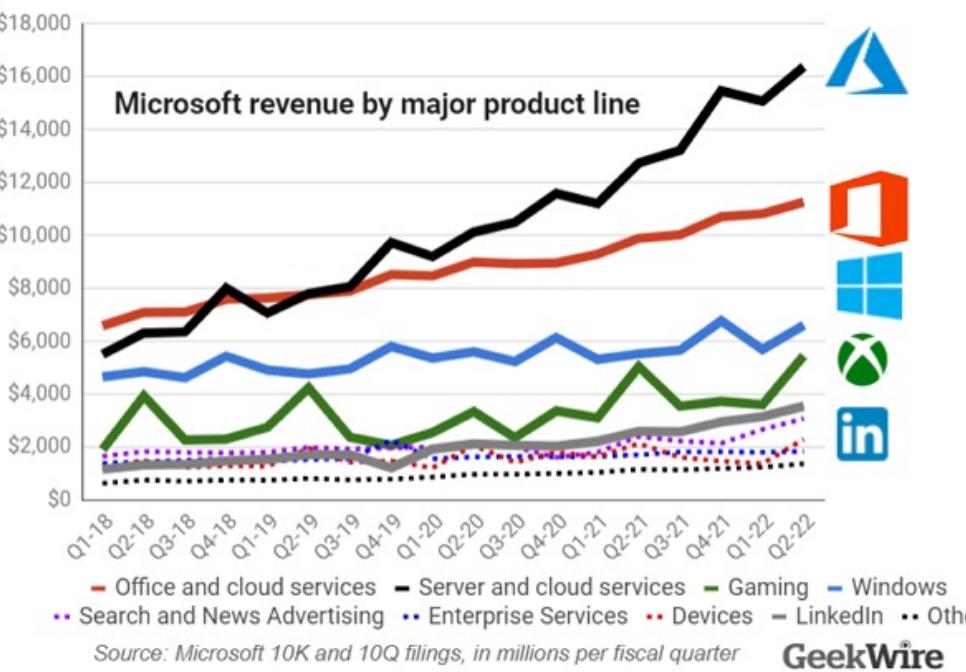
Worldwide Cloud IT Infrastructure Market Forecast by Deployment Type, 2018- 2024 (shares based on Value)



Source: IDC 2021



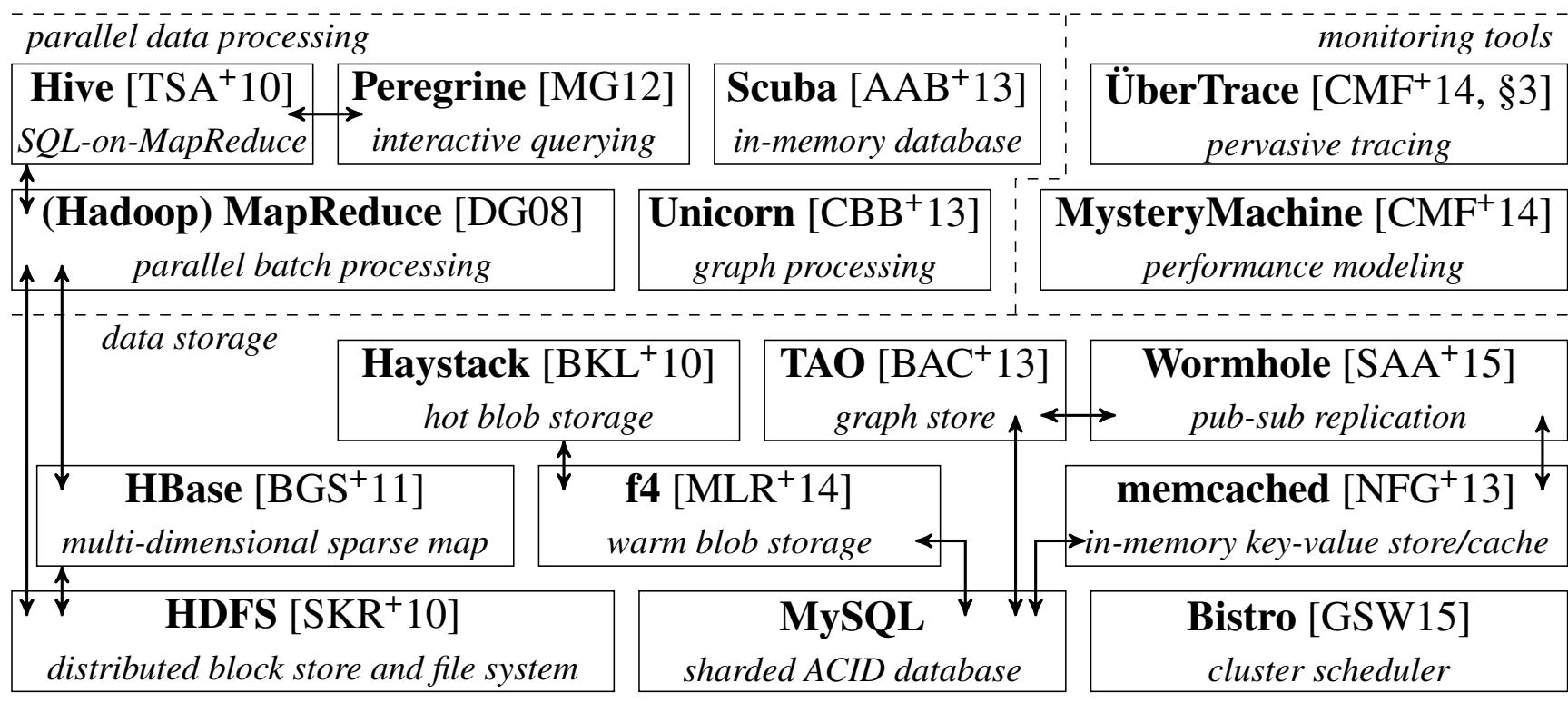
Source: Synergy Research Group



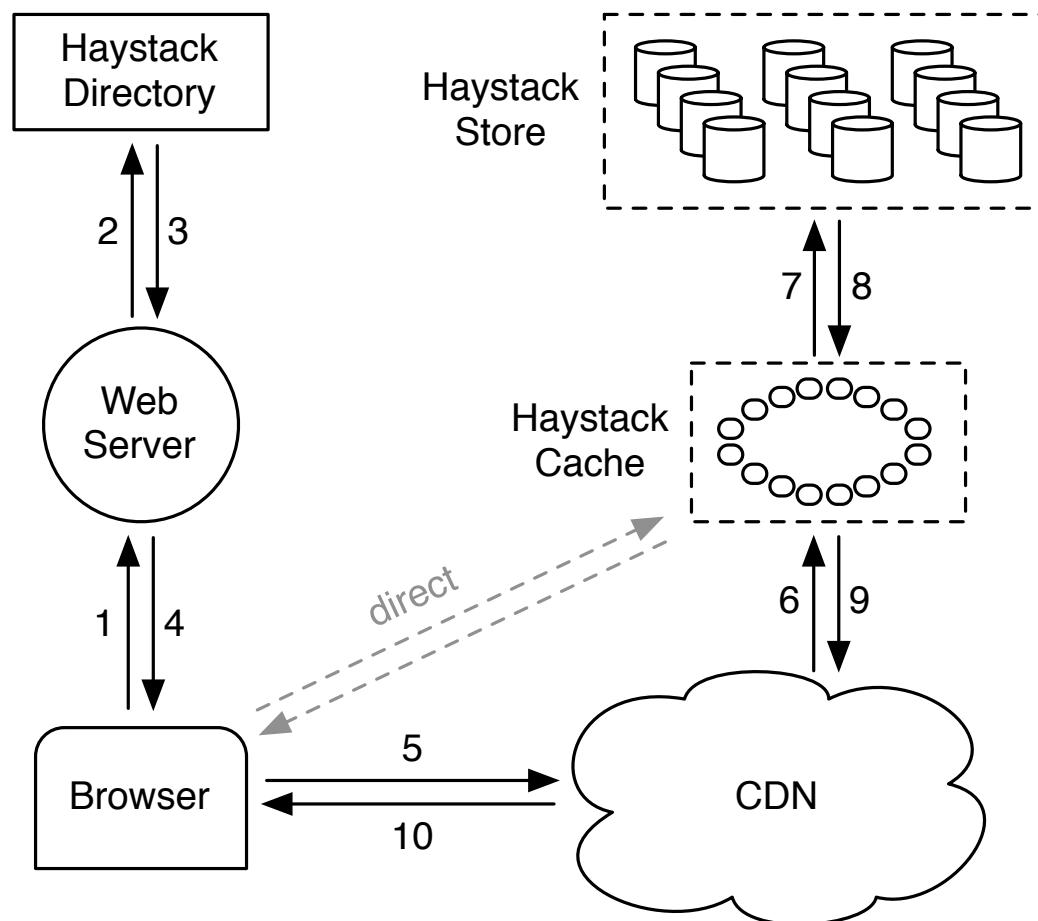
Source: Microsoft 10K and 10Q filings, in millions per fiscal quarter

GeekWire

EXEMPLO: FACEBOOK

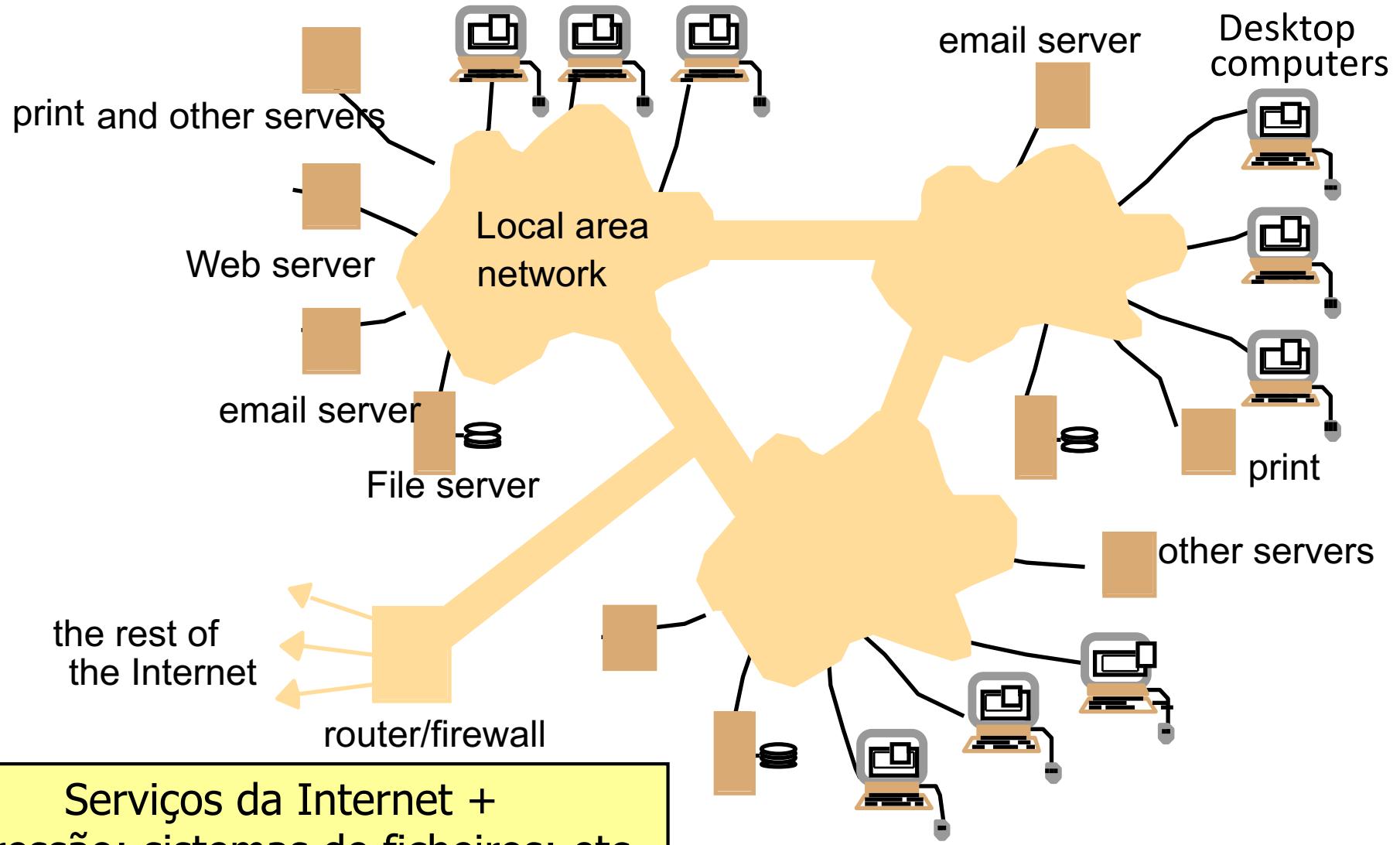


EXEMPLO: FACEBOOK HAYSTACK



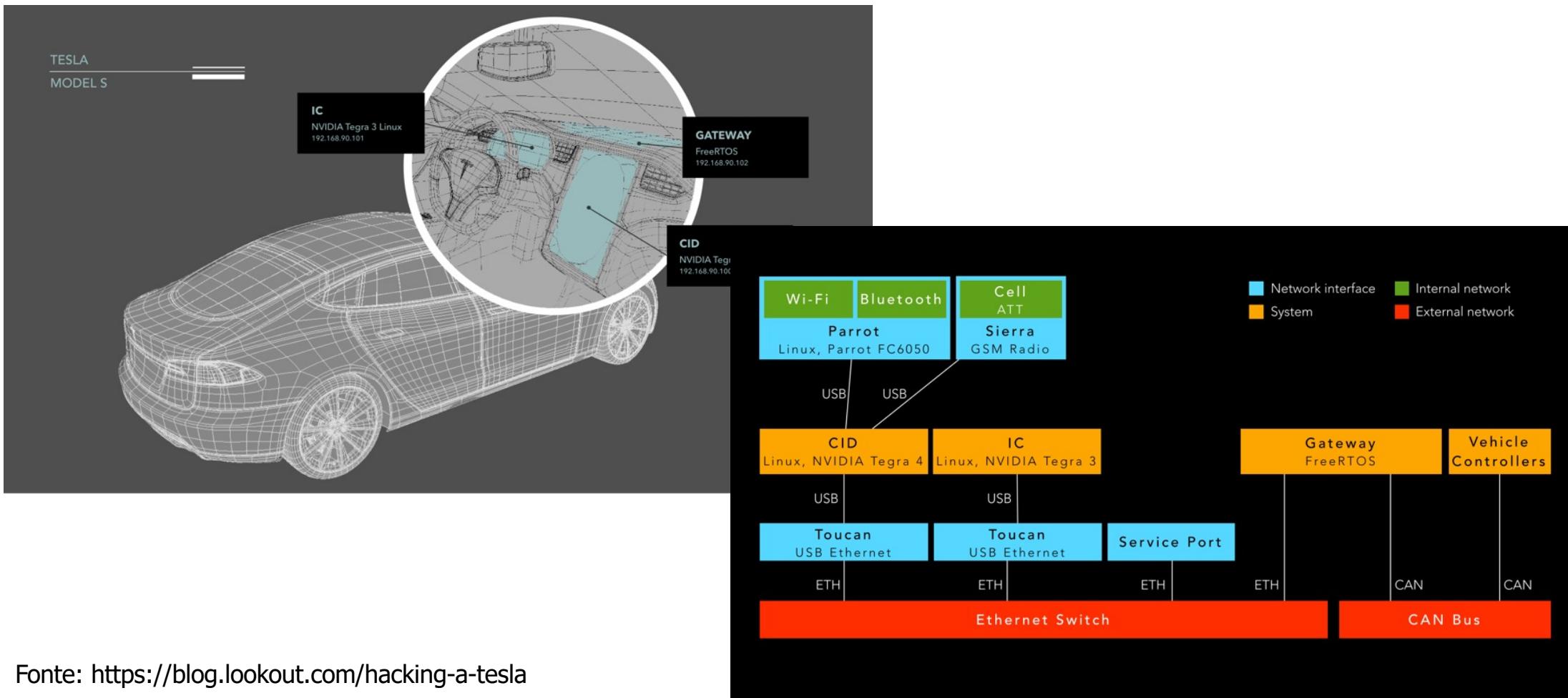
EXEMPLO: SERVIÇOS EM INTRANETS

Intranets: redes isoladas fisicamente, redes isoladas logicamente (private virtual network), ligação à Internet através de firewalls, etc.



OUTROS EXEMPLOS: CARROS

Dispositivos ou máquinas especiais controlados através de conjuntos de **computadores embebidos** (por exemplo: um avião ou um carro, uma fábrica)



OUTROS EXEMPLOS

Sistemas de controlo de processos industriais em fábricas (por exemplo, linhas de montagem)

Clusters de computadores interligados através de redes de alta velocidade para cálculo paralelo

MOTIVAÇÕES DOS SISTEMAS DISTRIBUÍDOS

Acesso generalizado sem restrições de localização

- Acessibilidade ubíqua (suporte para utilizadores fixos, móveis)

Partilha dos recursos distribuídos pelos diferentes utilizadores

- Exemplos: impressores, ficheiros

Distribuição da carga – melhoria do desempenho

Tolerância a falhas – melhoria da disponibilidade

Flexibilidade e adaptabilidade

- Decomposição de um sistema complexo num conjunto de sistemas mais simples

CARACTERÍSTICAS FUNDAMENTAIS

Componentes do sistema **executam de forma concorrente** – paralelismo real

- Necessidade de coordenação entre os vários componentes

Falhas independentes das componentes e das comunicações

- Impossível determinar se existe uma falha dum componente ou do sistema de comunicações
- Necessidade de tratar as falhas

Ausência de relógio global – existem limites para a precisão da sincronização dos relógios locais

- Impossível usar relógios locais para ordenar globalmente todos os eventos

IMPLICAÇÕES

Nenhum componente tem uma visão exacta instantânea do estado global de todo o sistema

Os componentes têm uma **visão parcial do estado global** do sistema

- Os componentes do sistemas estão distribuídos e só podem cooperar através da troca de mensagens, as quais levam um tempo não nulo a propagarem-se

Na presença de falhas, o estado global pode tornar-se incoerente, i.e., as visões parciais do estado global podem tornar-se incoerentes

- Por exemplo, réplicas de um objecto podem ficar incoerentes

AULA 2

... NA AULA 1

O que é um sistema distribuído

- Um sistema distribuído é um conjunto de componentes hardware e software interligados através de uma infra-estrutura de comunicações, que **cooperam e se coordenam entre si** apenas pela troca de mensagens, para execução de **aplicações distribuídas**

Exemplos de sistemas distribuídos

- Serviços na Internet
- Sistemas de controlo (e.g. fábricas)
- Sistemas embebidos e de tempo-real (e.g. carros, aviões)

... NA AULA 1

Características fundamentais

- Componentes do sistema **executam de forma concorrente** – paralelismo real
- **Falhas independentes** das componentes e das comunicações
- **Ausência de relógio global** – existem limites para a precisão da sincronização dos relógios locais

NESTA AULA

Desafios na conceção de sistemas distribuídos

DESAFIOS

Heterogeneidade

Abertura

Transparência

Segurança

Escala

Tratamento das falhas

HETERogeneidade

Hardware: smartphones, tablets, portáteis, servidores, clusters, ...

- Diferentes características dos processadores, da memória, da representação dos dados, dos códigos de caracteres,...

Redes de interligação e protocolos de transporte: Redes móveis (5G, 4G, 3G, GSM), WLANs, wired LANs,..., TCP/IP,

Sistema de operação: Windows, MacOS, iOS, Android,...

- Diferentes interfaces para as mesmas funcionalidades

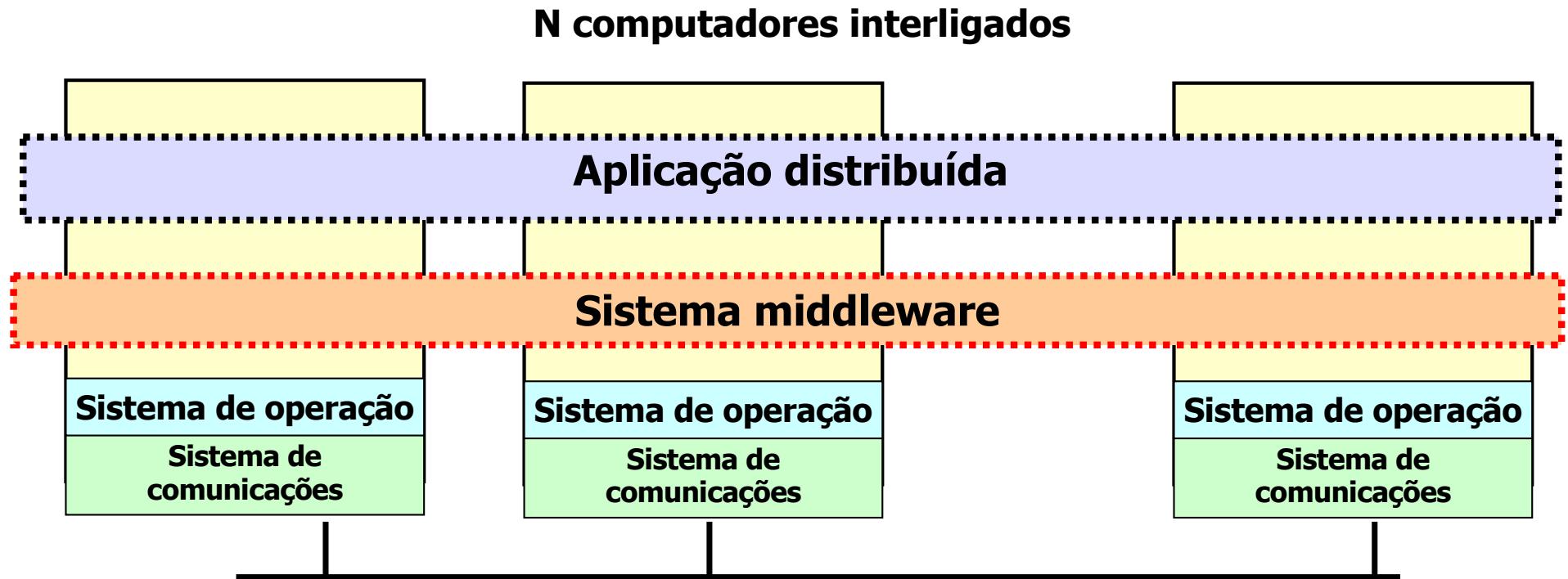
Linguagens de programação...

Lidar com esta heterogeneidade é muito complexo

As seguintes soluções podem ajudar:

- Sistemas de Middleware
- Máquinas Virtuais
- Containers

MIDDLEWARE



Sistemas operativos

Interfaces heterogénea

Serviços básicos

- Sistema *middleware*
 - Interface homogénea
 - Serviços mais complexos (invocação remota: Web-services; message-queue: *MQ, etc)
 - Verdadeira interoperabilidade requer idênticos interface e protocolos

MÁQUINA VIRTUAL

Objetivo: permitir executar os mesmos programas em máquinas com diferentes características

Máquina virtual aplicação: virtualiza ambiente de execução independentemente do sistema de operação

- E.g.: programas escritos numa única linguagem (JavaVM) ou em múltiplas linguagens (Microsoft CLR)

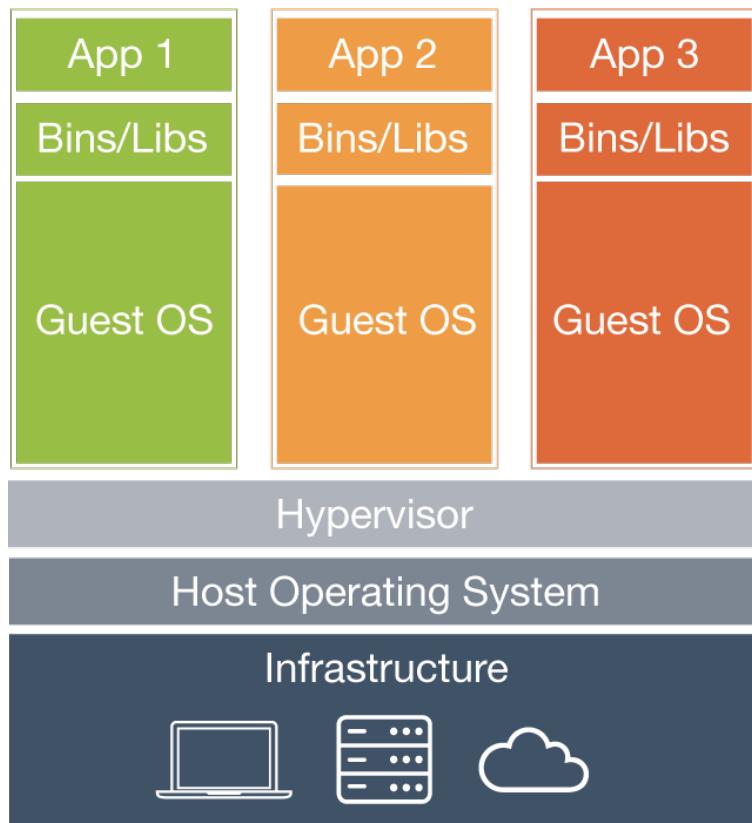
Máquina virtual sistema: virtualiza máquina física

- E.g.: VmWare, VirtualBox, etc.

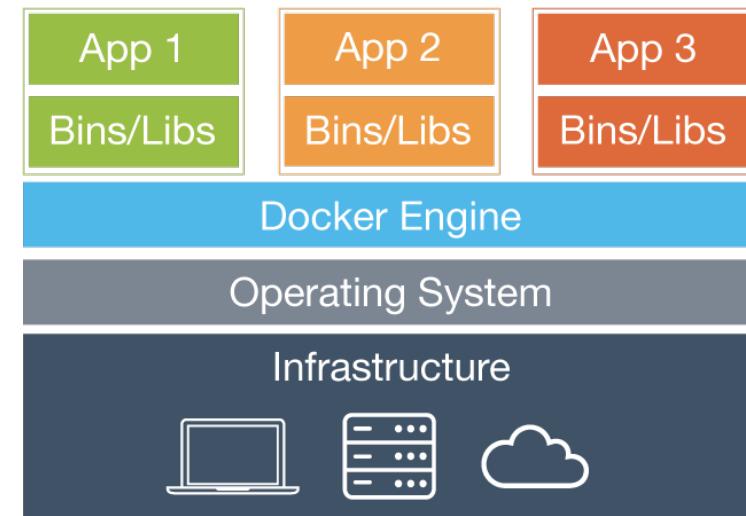
CONTAINER

Objetivo: permitir executar aplicações, incluindo todas as suas dependências

Máquina Virtual



Container



Images from [docker.com](https://www.docker.com)

CONTAINER (CONT.)

A container uses the underlying OS of the system (or VM) where it is running...

- The OS is shared among all containers running

.. But containers are isolated from each other with assigned exclusive resources, including CPU, memory, network, disk, etc.

This makes a container act as an independent machine, with its own IP.

ABERTURA

A abertura de um sistema determina o modo como pode ser estendido e re-implementado

Sistemas abertos

- Interfaces e modelo (incluindo protocolos de comunicação) conhecidos
- Evolução controlada por organismos de normalização independentes ou consórcios industriais
- Permite a interoperação de componentes com diferentes implementações

Sistemas proprietários

- Podem ser modificados pelo seu “dono”

Vantagens e desvantagens de cada aproximação?

ABERTURA E OPEN-SOURCE

Uma aplicação de código aberto (open source) é uma aplicação da qual o código fonte está disponível.

A utilização da aplicação é regida por uma licença que é tipicamente distribuída com o código. Existem vários tipos de licença:

Public domain – todos os direitos são transferidos.

Permissive licence (e.g. MIT, Apache) – oferece direito de utilização (incluindo relicenciar código derivado).

Copyleft (e.g. GPL) – oferece direito de utilização, proíbe tornar código proprietário (pode obrigar que todo o código que usa a aplicação tenha a mesma licença).

Noncommercial licence – oferece direito de utilização para utilização não comercial.

Proprietary licence – oferece direito de utilização em troca dum pagamento (utilização normal do copyright).

TRANSPARÊNCIA

A transparência (da distribuição) é a propriedade relativa a esconder ao utilizador e ao programador das aplicações a separação física dos elementos que compõem um sistema distribuído

- Simplicidade e flexibilidade são objectivos

Em certas circunstâncias, a transparência total é indesejável

- Que fazer em caso de falha?

SEGURANÇA

Necessidade de proteger os recursos e informação gerida num sistema distribuído

- Recursos têm valor para os seus utilizadores

Segurança tem três componentes:

- Confidencialidade: indivíduos não autorizados não podem obter informação
- Integridade: dados não podem ser alterados ou corrompidos
- Disponibilidade: acesso aos dados deve continuar disponível

Aspectos envolvidos

- Autenticação dos parceiros
- Canais seguros
- Prevenção de ataques de “negação de serviço” (denial of service attacks)

SEGURANÇA - RGPD

O Regulamento Geral sobre a Proteção de Dados (RGPD) (UE) 2016/679 regula a privacidade e proteção de dados pessoais, aplicável a todos os indivíduos na UE e Espaço Económico Europeu (EEU).

É aplicável a todas as empresas que operem no Espaço Económico Europeu, independentemente do seu país de origem.

Pretende garantir que:

- Não se podem guardar e disponibilizar dados sem consentimento explícito;
- Os dados guardados não podem ser utilizados sem que o proprietário tenha dado consentimento explícito;
- O proprietário tem o direito de revogar as permissões em qualquer momento.

ESCALA

A **escala de um sistema distribuído** é o âmbito que o mesmo abrange assim como o número de componentes.

A escala de um sistema tem várias facetas:

- recursos e utilizadores
- âmbito geográfico (rede local, país, mundo, ...)
- âmbito administrativo (uma organização, inter-organizações)

Um sistema capaz de escalar (escalável) é um sistema que continua eficaz quando há um aumento significativo do número de recursos e utilizadores

- i.e., em que não é necessário alterar a implementação dos componentes e da forma de interacção dos mesmos

COMO LIDAR COM A ESCALA ?

Para reduzir o número de pedidos tratados por cada componente

- Divisão de um componente em partes e sua distribuição
- Replicação e caching (problema da consistência entre réplicas e caches)

Para reduzir o tempo de acesso de clientes distribuídos geograficamente?

- Geo-replicação – replicar as aplicações (e dados) em diferentes locais geográficos
- Replicação na edge – replicar as aplicações ou dados na periferia da rede (e.g. CDN)

COMO LIDAR COM A ESCALA ?

Para reduzir dependências entre componentes

- Meios de comunicação assíncronos

Para simplificar o sistema

- Uniformidade de acesso aos recursos e dos mecanismos de cooperação, sincronização, etc.
- Meios de designação universais (independentes da localização e dos recursos)

AVARIAS, ERROS E FALHAS

Os componentes de um sistema podem **falhar**, i.e., comportar-se de forma não prevista e não de acordo com a especificação devido a **erros** (por exemplo a presença de ruído num canal de comunicação ou um erro de software) ou **avarias** (mecanismo que entra em mau funcionamento)

Num sistema distribuído, as **falhas são geralmente parciais** (num componente do sistema) **e independentes**

- Um componente em falha pode induzir uma mudança de estado incorrecta noutro componente, levando eventualmente o sistema a falhas, i.e., a ter um comportamento não de acordo com a sua especificação.

COMO LIDAR COM AS FALHAS

Detectar falhas

- Possível: e.g.: mensagens corrompidas através de checksums
- Pode ser impossível: Falha (crash) num computador remoto
 - Desafio: Funcionar através da suspeição das falhas

Mascarar falhas (após a sua detecção)

- Exemplos: retransmissão de mensagens, redundância

Tolerar falhas

- Definição do comportamento na presença de falhas
 - Parar até falhas serem resolvidas; recorrer a componentes redundantes para continuar a funcionar

Recuperação de falhas

- Mesmo num sistema que tolere falhas é necessário recuperar os componentes falhados. Porquê?
- Problema: recuperar estado do serviço

PARA SABER MAIS

G. Coulouris, J. Dollimore and T. Kindberg,
Distributed Systems – Concepts and Design,
Addison-Wesley, 5th Edition, 2011

Capítulo 1.

SISTEMAS DISTRIBUÍDOS

Capítulo 2

Arquiteturas e Modelos de Sistemas Distribuídos

NOTA PRÉVIA

A apresentação utiliza algumas das figuras do livro de base do curso

G. Coulouris, J. Dollimore and T. Kindberg,
Distributed Systems - Concepts and Design,
Addison-Wesley, 5th Edition, 2005

ORGANIZAÇÃO DO CAPÍTULO

Modelos arquiteturais

- Arquitetura/camadas de software
- Cliente/servidor, peer-to-peer, variantes

Modelos fundamentais – usados para descrever propriedades parciais, comuns a todas as arquiteturas

- Modelo de interação
- Modelo de falhas
- Modelo de segurança

CONTEXTOS - ARQUITETURA

Camadas de software

- Reparte a complexidade de um sistema, em várias camadas, com interfaces bem definidas entre si. Cada camada pode usar os serviços da camada abaixo, sem conhecimento dos detalhes de implementação.

Arquitetura (distribuída) multinível/camada

- as camadas do sistema são atribuídas a processos/máquinas diferentes

Arquitetura distribuída

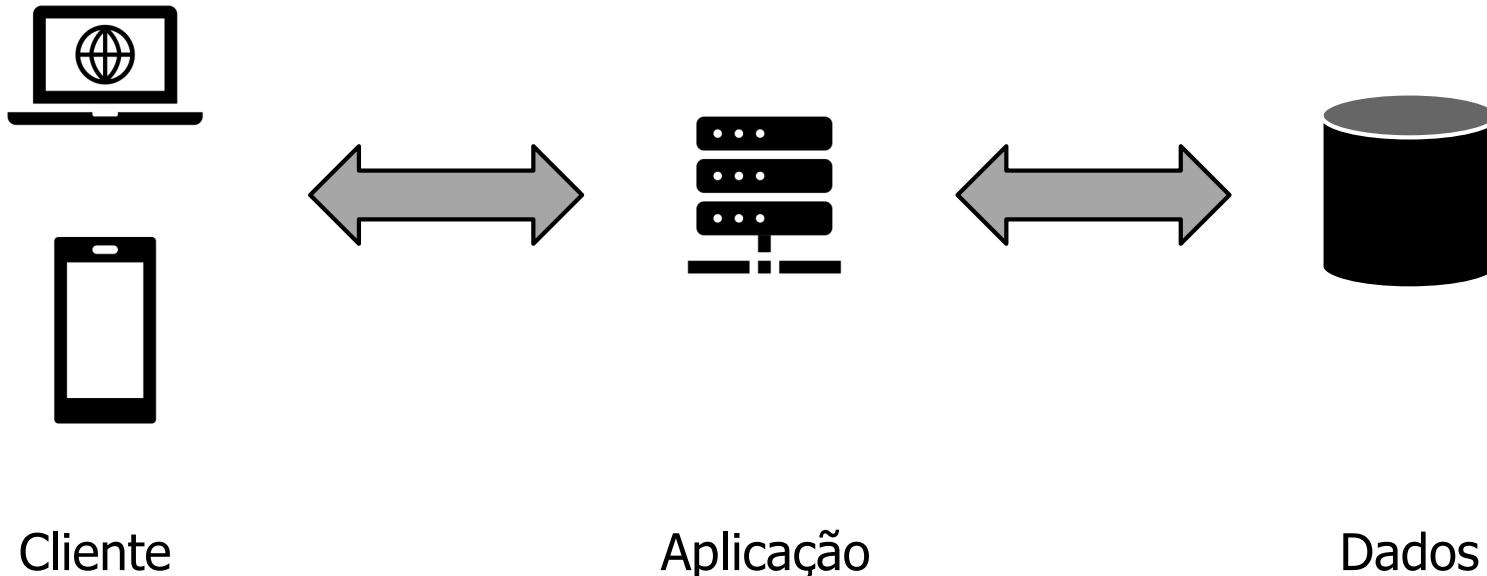
- Especifica como se organizam e quais as interações entre os vários **componentes** de **um sistema distribuído**
- Em todos os casos há implicações no desempenho, fiabilidade e segurança do sistema



ARQUITETURA EM CAMADAS: MODELO THREE-TIER

As aplicações Web e móveis são frequentemente implementadas usando uma arquitetura de três níveis:

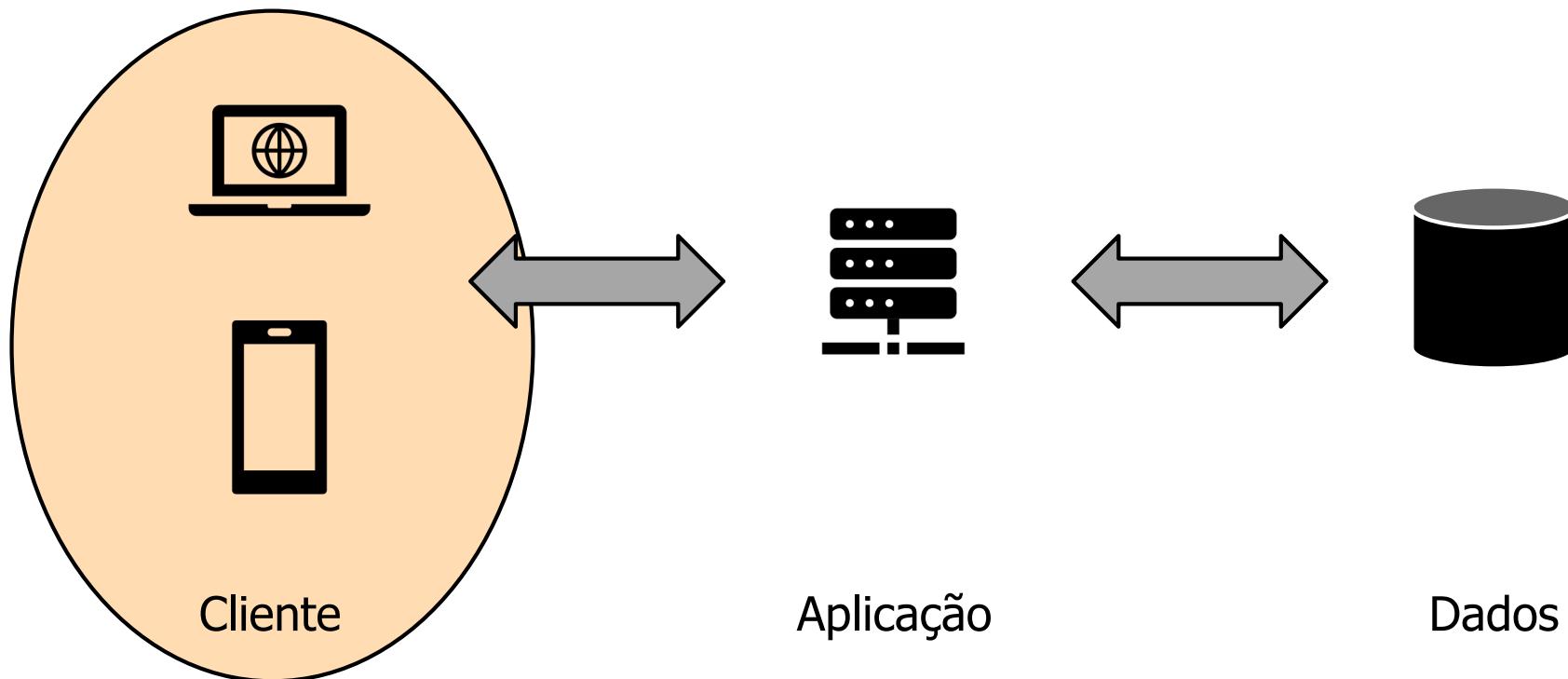
- Cliente (ou apresentação)
- Aplicação (ou lógica)
- Dados (ou armazenamento)



ARQUITETURA EM CAMADAS: MODELO THREE-TIER

O nível do cliente é composto:

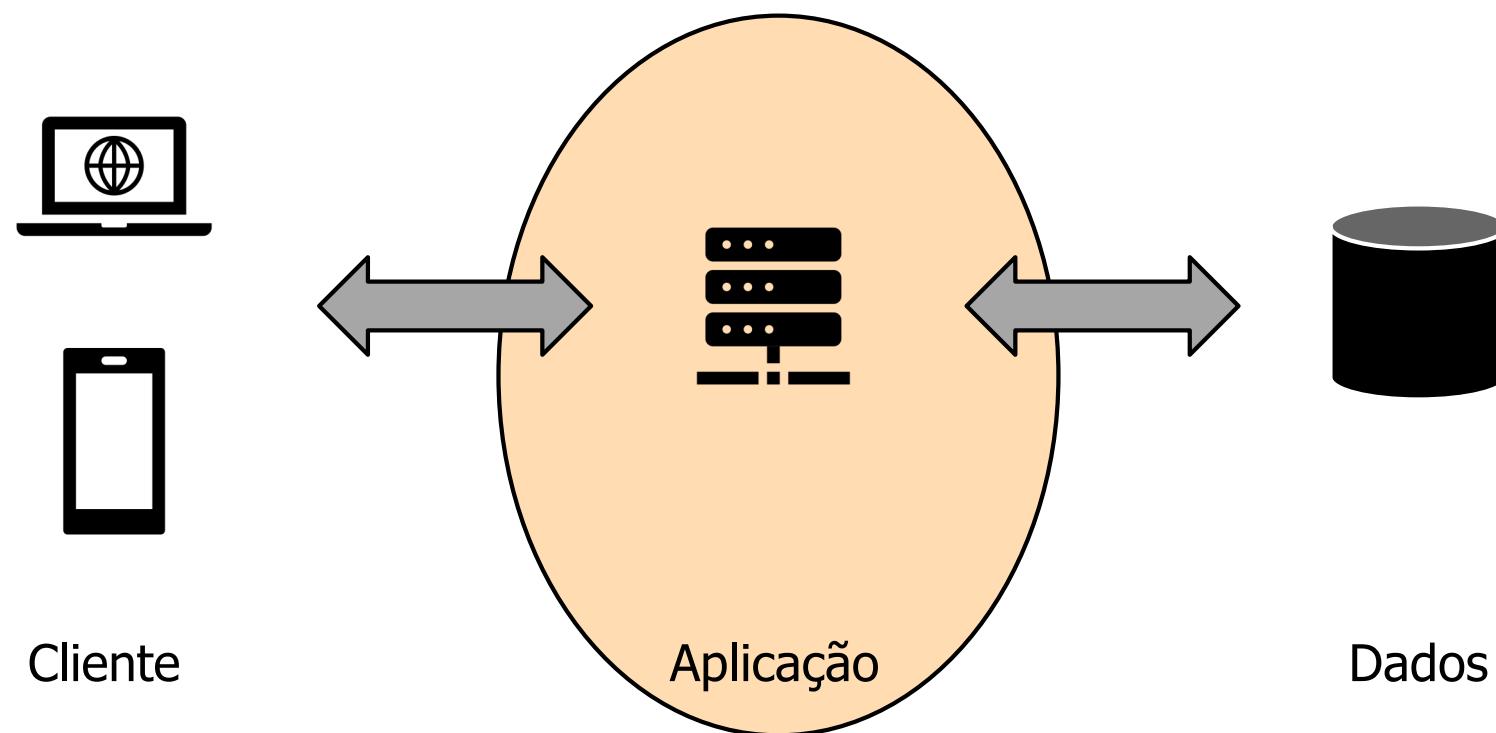
- Clientes Web: HTML + Javascript
- Aplicações móveis: Android, iOS, etc.



ARQUITETURA EM CAMADAS: MODELO THREE-TIER

Nível da aplicação é composto por:

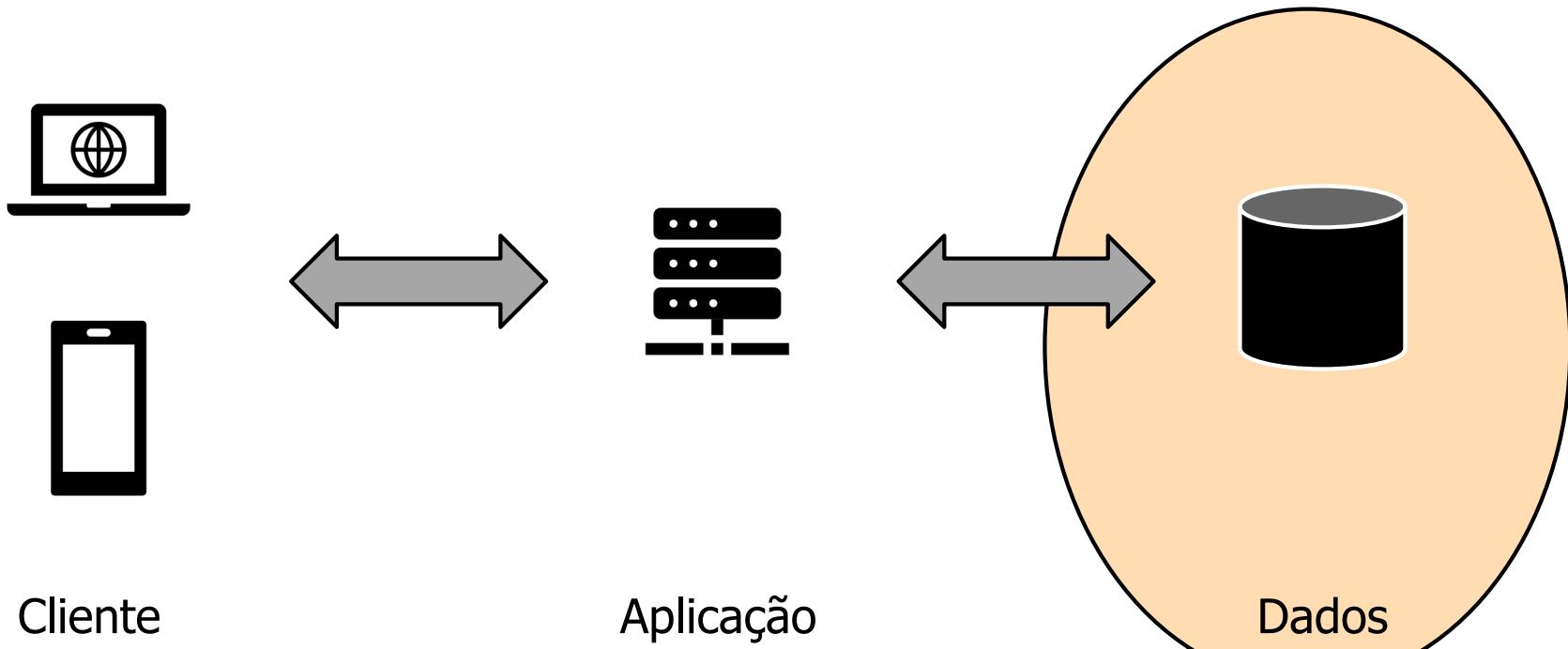
- Servidores REST / SOAP
- Páginas web dinâmicas e estáticas
- Implementado usando servidores aplicacionais
 - E.g.: Tomcat, Wildfly, ASP.NET, etc.



ARQUITETURA EM CAMADAS: MODELO THREE-TIER

O nível dos dados é composto por:

- Sistemas de ficheiros, BLOB stores, key-value stores, bases de dados SQL, etc.
- Outros serviços: caches, filas de mensagens, etc.



ARQUITETURA DISTRIBUÍDA

Arquitetura do sistema

- Organização de um sistema (complexo) em componentes **mais simples** com funcionalidades/responsabilidades próprias

Arquitetura do sistema distribuído

- Define os componentes, o que fazem, onde estão e como interagem entre si.
- Terá implicações em diversas propriedades do sistema: desempenho, fiabilidade e segurança do sistema

ARQUITETURA DISTRIBUÍDA

Arquitetura de um sistema distribuído pode (e deve) ser determinada por diversos fatores:

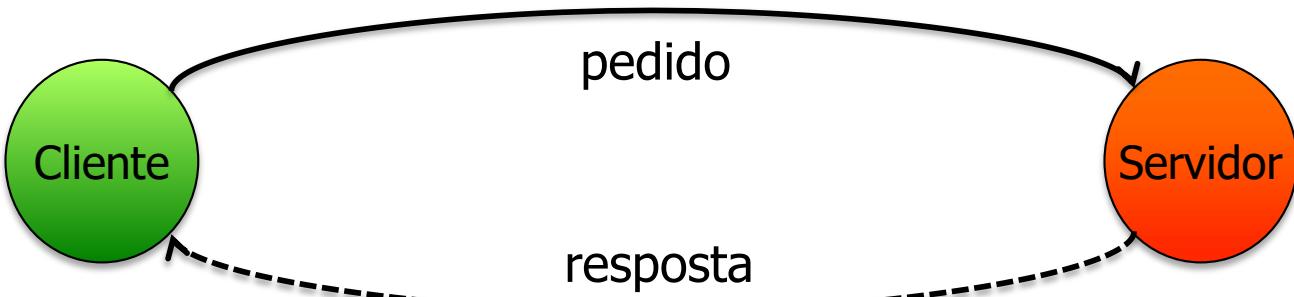
- Requisitos funcionais:
- “tudo relacionado com a função do sistema”
- e.g. Lógica de negócio

Requisitos não funcionais:

- Desempenho (escalabilidade, latência), disponibilidade
- Custo (desenvolvimento, operação, manutenção)
- Segurança, confiabilidade

Arquitetura distribuída mais simples e muito comum?

CLIENTE/SERVIDOR



Sistema em que os processos podem ser divididos em dois tipos, de acordo com o seu modo de operação:

Cliente: programa que solicita pedidos a um processo servidor

Servidor: programa que executa operações solicitadas pelos clientes, enviando-lhes o respectivo resultado

CLIENTE/SERVIDOR: PROPRIEDADES

Arquitetura mais simples, muito comum e usada na prática...

Positivo

- Interação simples facilita implementação
- Segurança apenas tem de se concentrar no servidor

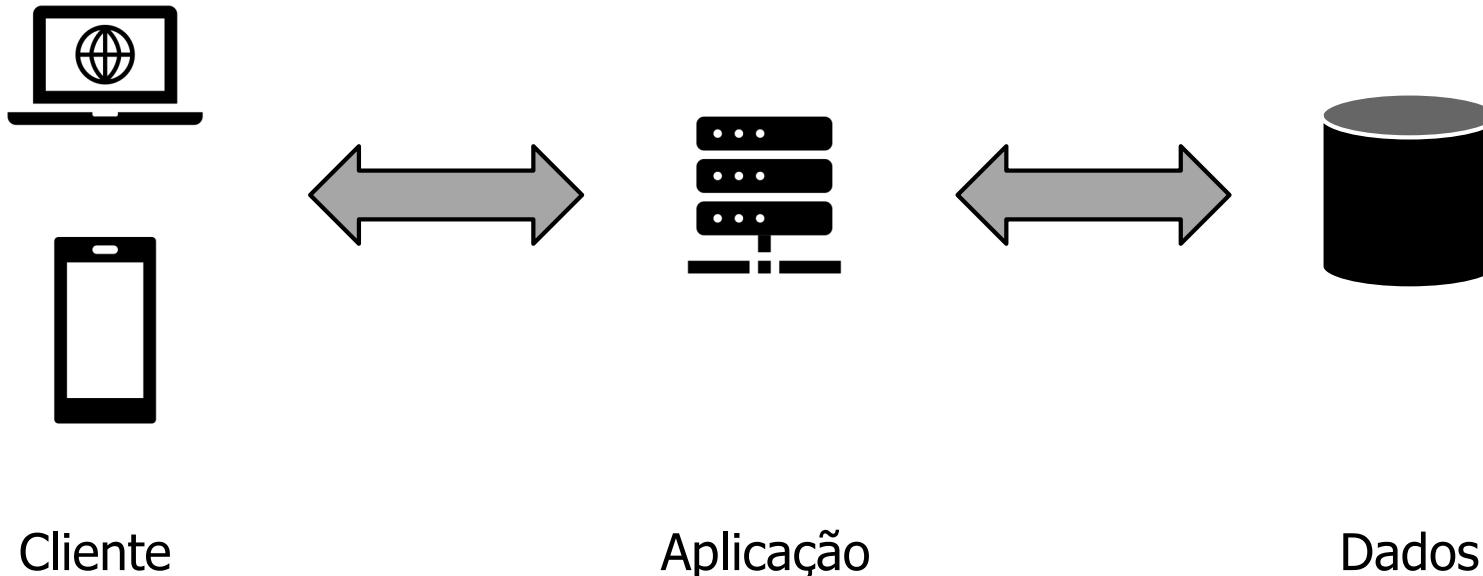
Negativo

- Servidor é um ponto de falha único
- Não escala para além dum dado limite (servidor pode tornar-se ponto de contenção - *bottleneck*)

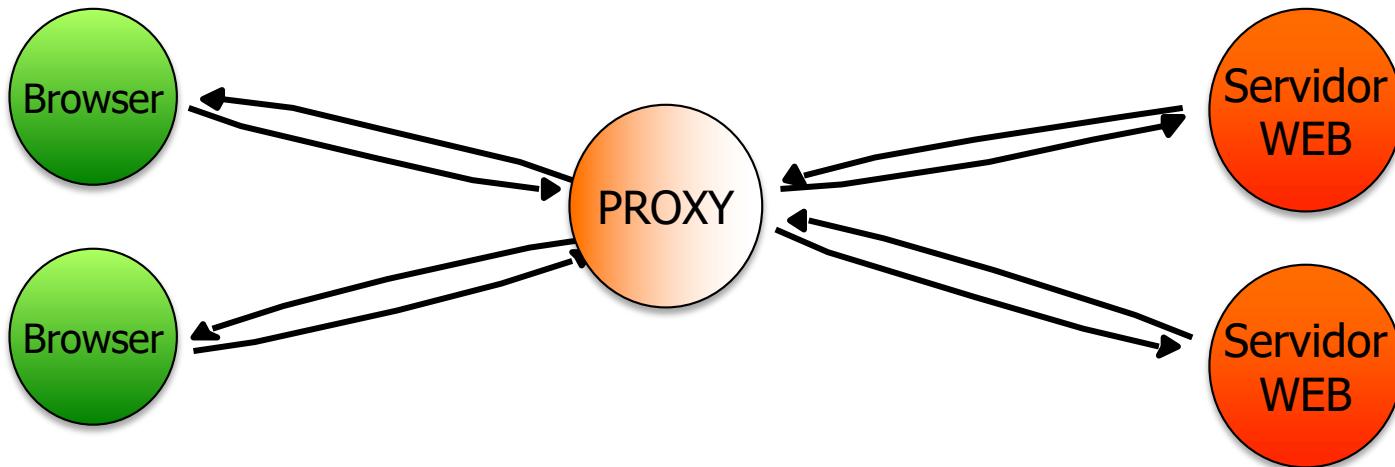
ARQUITETURA EM CAMADAS E MODELO CLIENTE/SERVIDOR

As aplicações que usam uma arquitetura em três camadas tipicamente usam interação cliente/servidor entre as várias camadas

- Cliente <-> Aplicação
- Aplicação <-> Dados



Noção de PROXY DE UM SERVIÇO



Proxy de um serviço

Componente que fornece um serviço recorrendo a um servidor (do serviço) para executar o serviço

Utilizações possíveis

Intermediário simples (apenas encaminha pedidos e respostas)

Intermediário complexo (*gateway*)

Transformação dos pedidos

Serviço adicional através do *caching* das respostas

Diminuição do tempo de resposta (latência inferior para o proxy)

Diminuição da carga do servidor

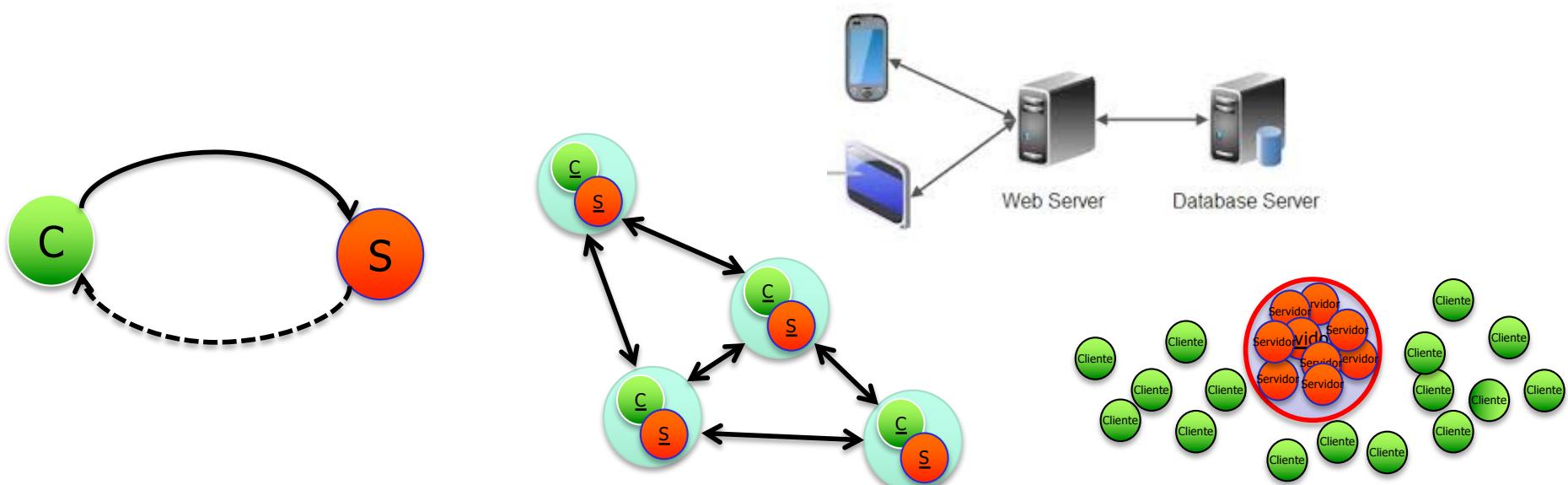
Mascarar falhas do servidor / desconexão

CLIENTE/SERVIDOR: COMPOSIÇÃO

Arquiteturas mais sofisticadas, com novas propriedades, podem ser obtidas por composição do modelo C/S base

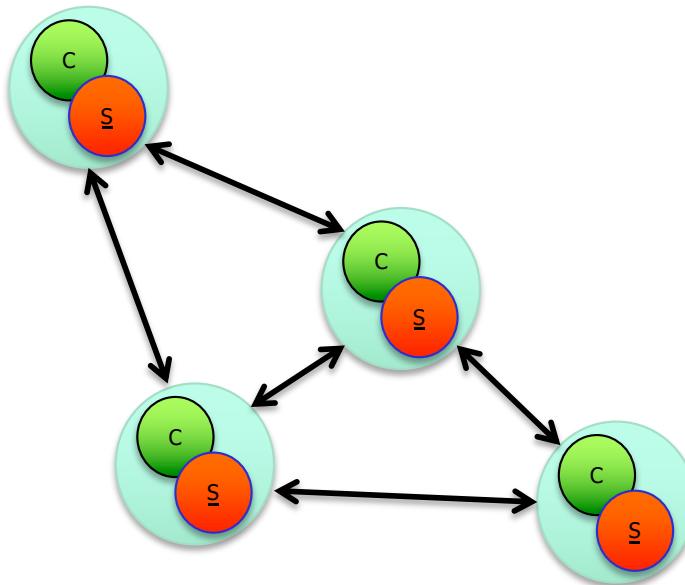
Servidor [particionado, replicado, geo-replicado]

P2P, 3-Tier, etc.



Modelo alternativo para lidar com limitações do
modelo cliente/servidor

MODELO PEER-TO-PEER (P2P)

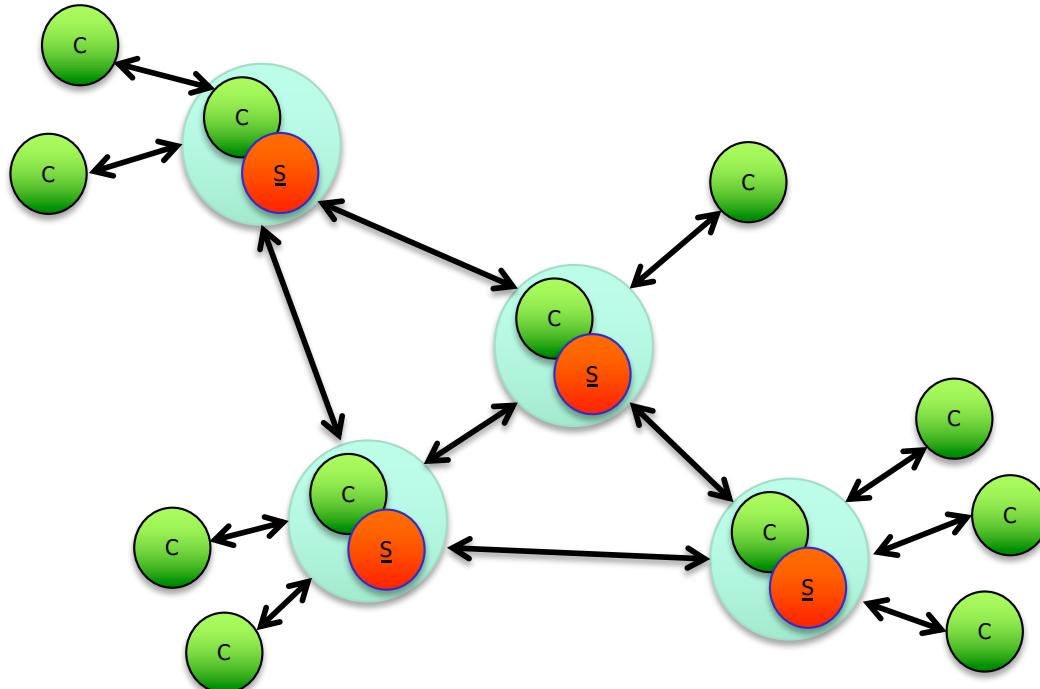


Todos os processos têm funcionalidades semelhantes

Durante a sua operação podem assumir o papel de clientes e servidores do mesmo serviço em diferentes momentos

Exemplos: partilha de ficheiros, VoIP, edição colaborativa

MODELO PEER-TO-PEER (P2P)



Frequentemente o modelo peer-to-peer é usado para implementar um serviço, existindo clientes que usam este serviço contactando um nó do serviço usando o modelo cliente/servidor.

MODELO *PEER-TO-PEER*: PROPRIEDADES

Positivo

- Não existe ponto único de falha
- Melhor potencial de escalabilidade
- Baixo custo de operação

Negativo

- Interação mais complexa (do que num sistema cliente/servidor) leva a implementações mais complexas
 - Operações de pesquisa podem ser complexas
- Maior número de computadores envolvidos pode colocar questões relativas a heterogeneidade e segurança

MODELO *PEER-TO-PEER*: PROPRIEDADES

Apropriado para ambientes em que todos os participantes querem cooperar para fornecer uma dado serviço

Razões possíveis para cooperação:

- Aumentar a capacidade do sistema
 - Capacidade agregada >> capacidade individual
 - E.g. Sistemas P2P de partilha de ficheiros.
- Manter controlo sobre os dados
 - Cada servidor controla parte dos dados
 - E.g. Sistema de email.

Variantes do modelo cliente/servidor para lidar com limitações de escalabilidade e tolerância a falhas

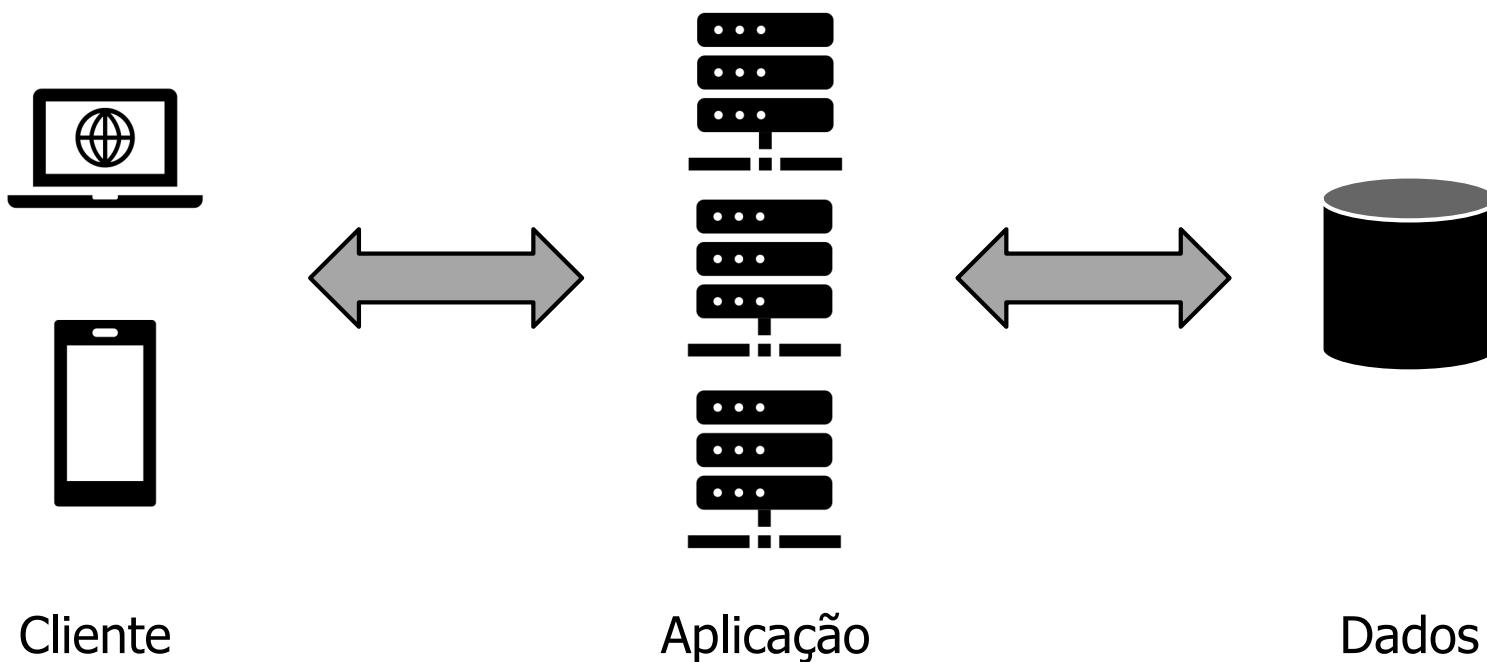
Soluções do lado do servidor

NOTA PRÉVIA

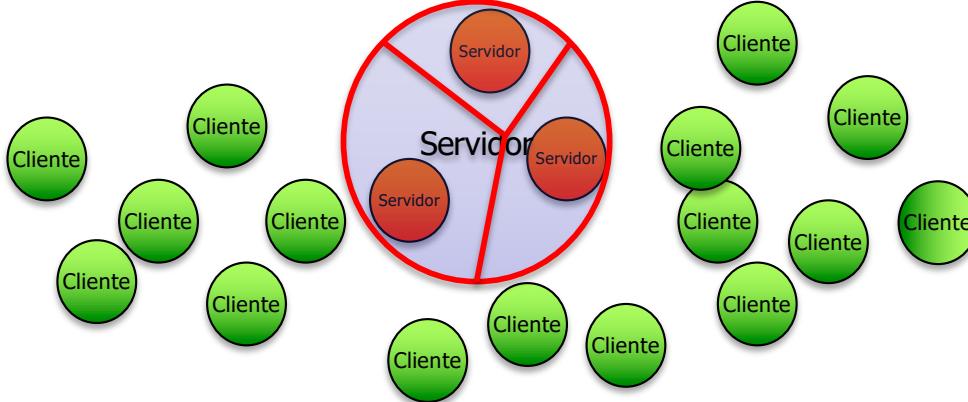
Qual a dificuldade de replicar / particionar um servidor?

Necessário lidar com o estado armazenado pelo servidor.

Se o servidor não tiver estado, basta criar novas cópias do servidor. E.g. replicar o o servidor de aplicação *stateless* numa arquitetura de três níveis.



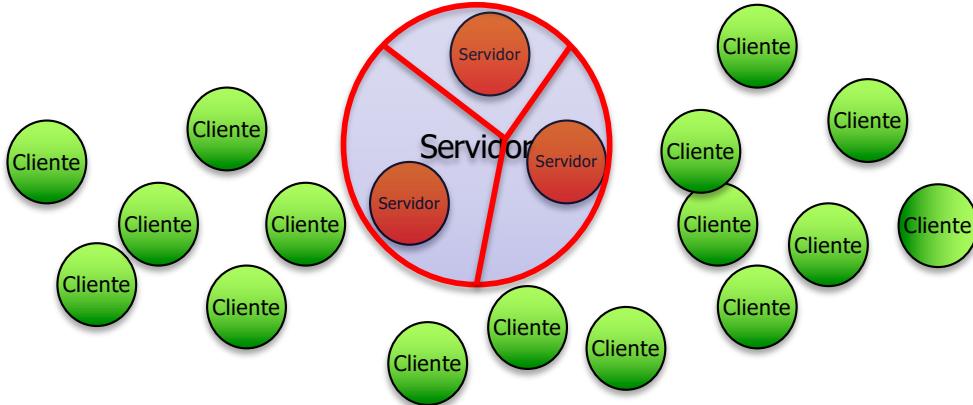
CLIENTE/SERVIDOR PARTICIONADO



Existem vários servidores com a mesma interface, cada um capaz de responder **a uma parte** dos pedidos

Exemplo: DNS.

CLIENTE/SERVIDOR PARTICIONADO



Como é que o pedido do cliente é enviado para o servidor correto?

- Servidor redirige cliente para outro servidor (iterativo)
- Servidor invoca pedido noutro servidor (recursivo)

CLIENTE/SERVIDOR PARTICIONADO

Positivo

- Permite distribuir a carga, melhorando o desempenho (potencialmente)
- Não existe um ponto de falha único

Negativo

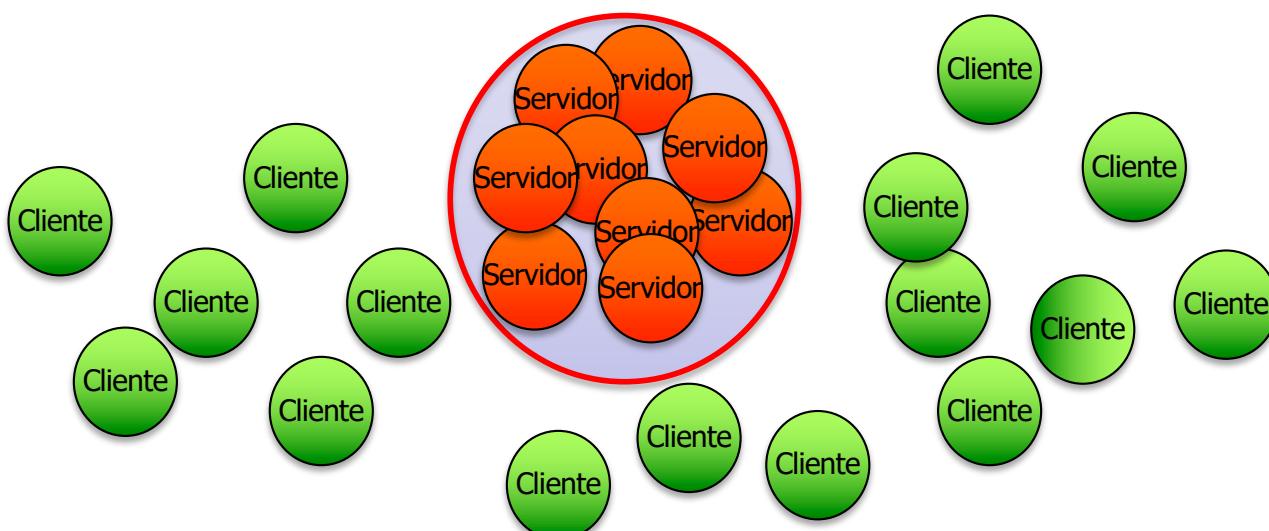
- Falha de um servidor impede acesso aos dados presentes nesse servidor
- Difícil de aplicar em alguns modelos de dados

Sendo: w : nº de escritas; r : nº de leituras; n : nº de partições

Cada partição recebe, em média: $w / n + r / n$ pedidos

CLIENTE/SERVIDOR REPLICADO

Existem vários servidores idênticos (i.e. capazes de responder aos mesmo pedidos)



CLIENTE/SERVIDOR REPLICADO

Positivo

- Redundância - não existe um ponto de falha único
- Permite distribuir a carga, melhorando o desempenho (potencialmente)

Negativo

- Coordenação - manter estado do servidor coerente em todas as réplicas
- Recuperar da falha parcial de um servidor

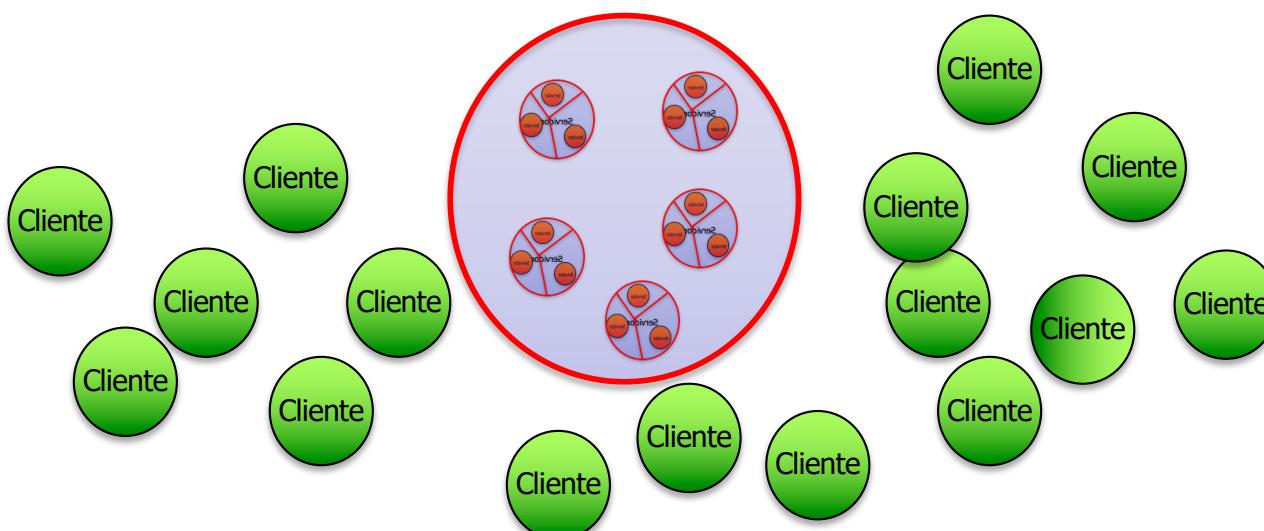
Sendo: w : nº de escritas; r : nº de leituras; n : nº de réplicas

Se todas as réplicas receberem todos os pedidos de escrita, cada réplica recebe: $w + r / n$ pedidos

CLIENTE/SERVIDOR REPLICADO + PARTICIONADO

Na prática, as implementações frequentemente combinam o particionamento com a replicação.

Cada partição é replicada em várias máquinas.



CLIENTE/SERVIDOR REPLICADO + PARTICIONADO

Positivo

- Redundância - não existe um ponto de falha único
- Permite distribuir a carga, melhorando o desempenho (potencialmente)

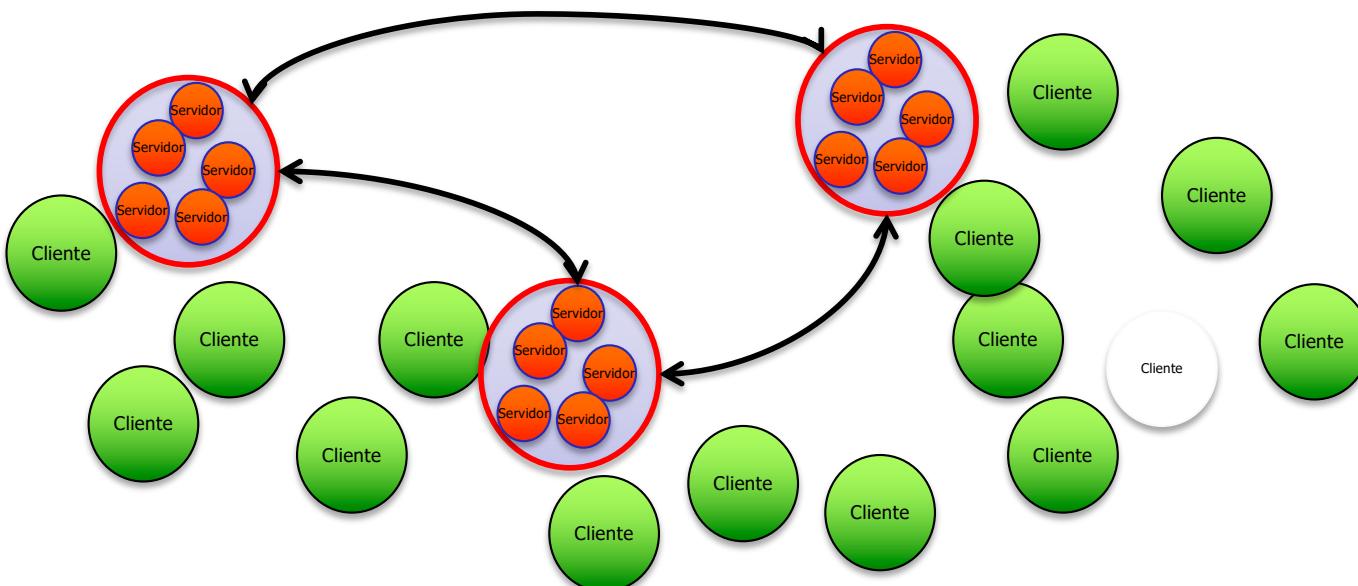
Negativo

- Coordenação - manter estado do servidor coerente em todas as réplicas

CLIENTE/SERVIDOR GEO-REPLICADO

Servidor replicado, com réplicas distribuídas geograficamente

Tipicamente, dentro de cada localização geográfica, os dados são particionados+replicados



CLIENTE/SERVIDOR GEO-REPLICADO

Positivo

- Proximidade aos clientes melhora a qualidade de serviço (latência)
- Redundância acrescida – as falhas das réplicas são (ainda) mais independentes

Negativo

- Coordenação mais dispendiosa – maior separação física das réplicas traduz-se na utilização de canais com latência significativa

SISTEMAS DISTRIBUÍDOS

Capítulo 2

Arquiteturas e Modelos de Sistemas Distribuídos

... NA AULA ANTERIOR

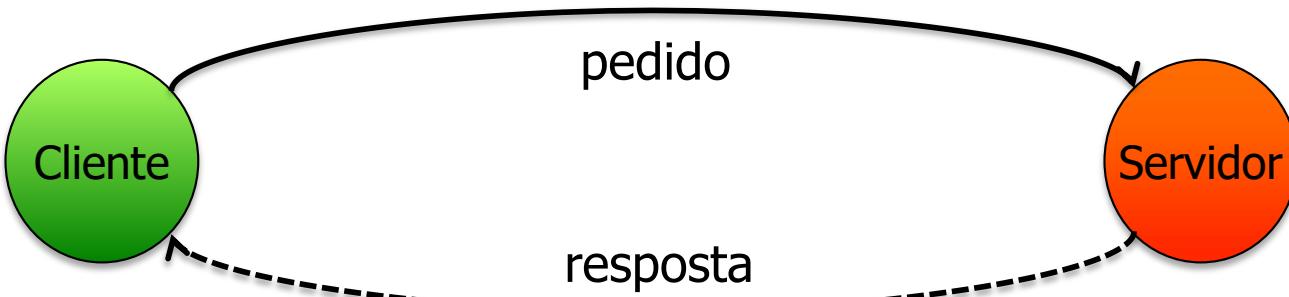
Arquitetura de um sistema distribuído

Estabelece a sua organização em componentes mais simples com funcionalidades/responsabilidades próprias

Define: quais os componentes, o que fazem, onde estão e como interagem entre si

Desenhada: com base num conjunto de requisitos (funcionais e não funcionais)

CLIENTE/SERVIDOR



Positivo

Interação simples facilita implementação

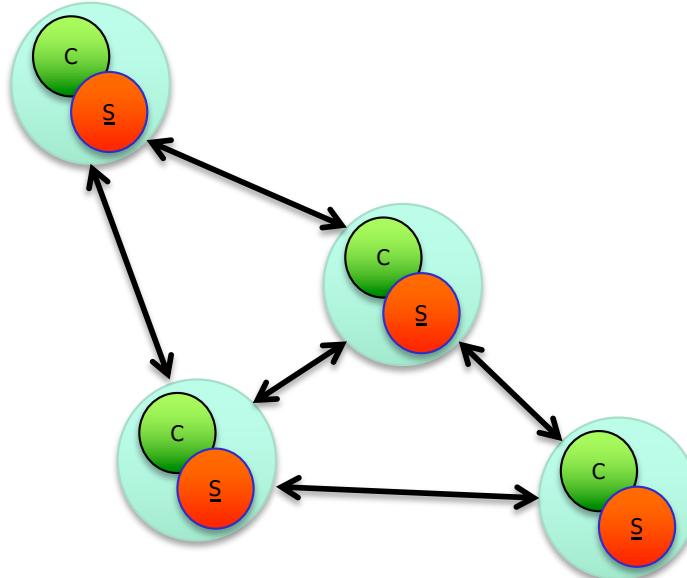
Segurança apenas tem de se concentrar no servidor

Negativo

Servidor é um ponto de falha único

Não escala para além dum dado limite (servidor pode tornar-se ponto de contenção - *bottleneck*)

MODELO PEER-TO-PEER (P2P)



Positivo

Não existe ponto único de falha

Melhor potencial de escalabilidade

Negativo

Interação mais complexa

Maior número de computadores envolvidos pode colocar questões relativas a heterogeneidade e segurança

... NA AULA ANTERIOR

Variantes do modelo cliente/servidor: servidor

Cliente/servidor particionado

Cliente/servidor replicado

Cliente/servidor particionado + replicado

Cliente/servidor geo-replicado

Variantes do modelo cliente/servidor para lidar com limitações de escalabilidade e tolerância a falhas

Soluções do lado do cliente

CLIENTE LEVE (*THIN CLIENT*)/SERVIDOR

O cliente apenas inclui uma interface (gráfica) para executar operações no servidor (ex.: browser)

Positivo:

- Cliente pode ser muito simples

Negativo

- Maior peso no servidor
- Impacto na interatividade (latência)

CLIENTE COMPLETO (ESTENDIDO)/SERVIDOR

O cliente executa localmente algumas operações que seriam executadas pelo servidor

Usado na Web em vários níveis: browsers fazem cache de páginas, imagens, etc; HTML 5.0 permite às aplicações Web guardar dados localmente – e.g. suporte offline no Google Docs, Gmail

CLIENTE COMPLETO (ESTENDIDO)/SERVIDOR

Positivo:

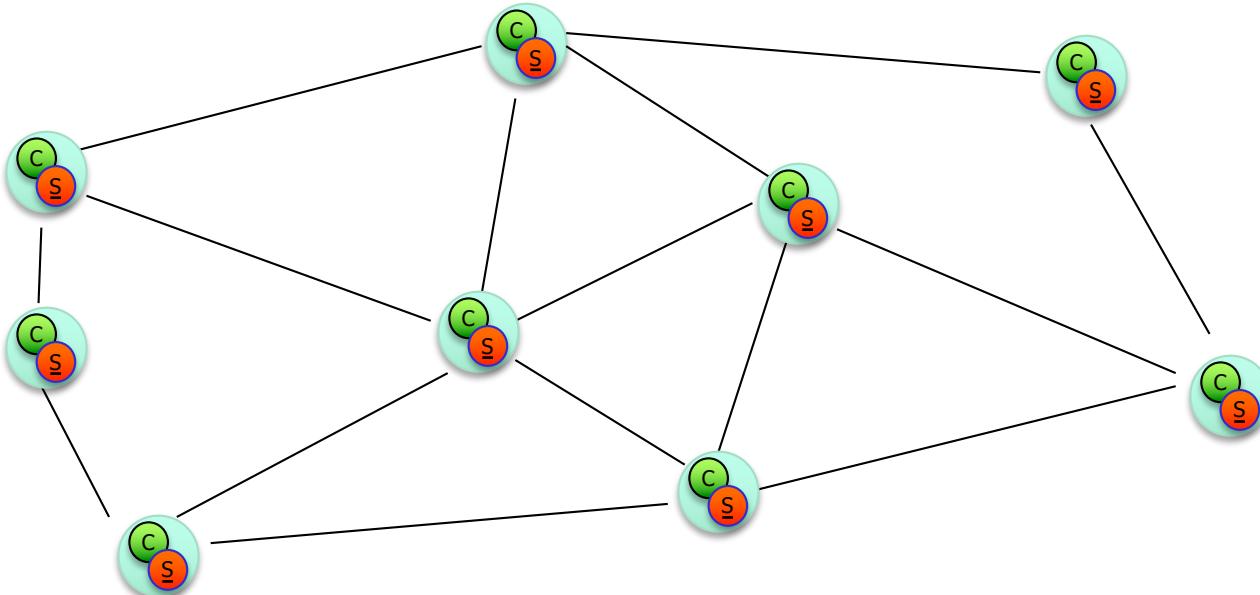
- Permite funcionar *offline*, quando não é possível contactar o servidor (recorrendo a *caching*)
- Permite diminuir a carga do servidor e melhorar o desempenho e a interatividade

Negativo:

- Implementação do cliente mais complexa
- Necessário tratar da coerência dos dados entre o cliente e o servidor

Variantes do modelo P2P com diferentes propriedades

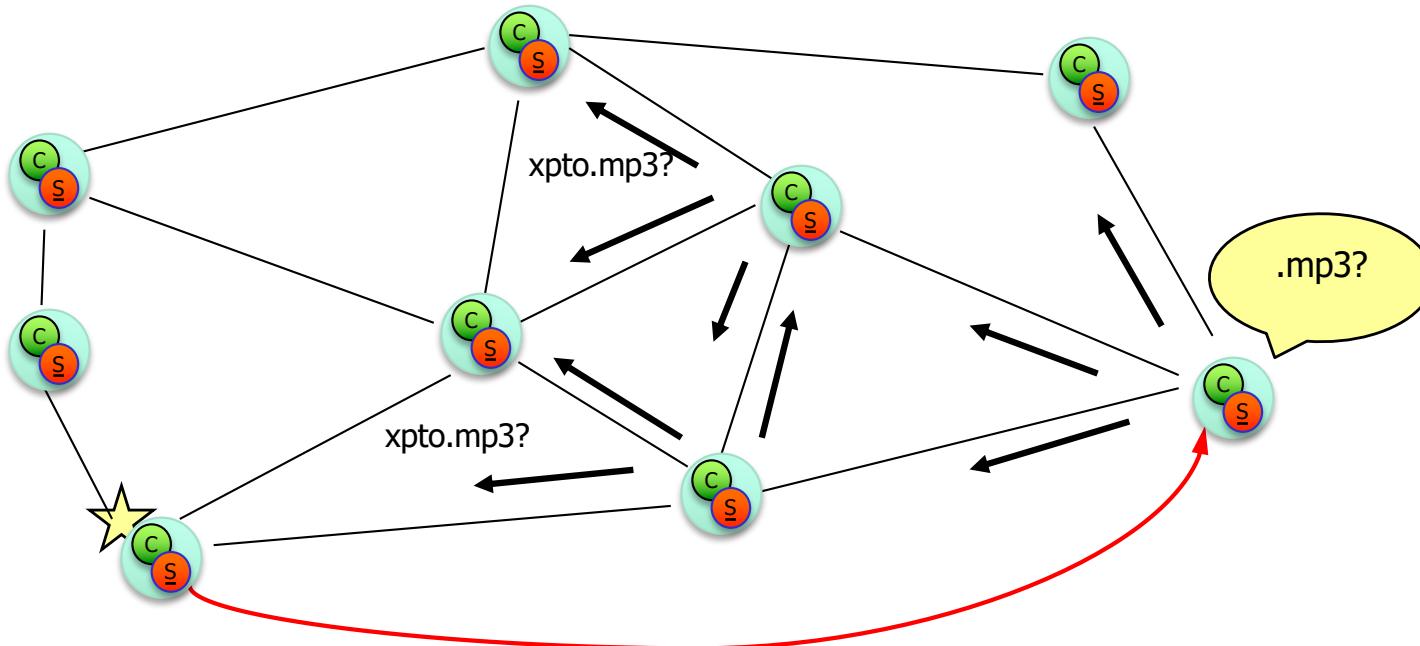
SISTEMA P2P NÃO ESTRUTURADO



As ligações entre os membros são formadas de forma **não-determinista**

E.g. quando se junta à rede, um membro escolhe para vizinhos um pequeno conjunto de contactos (os contactos podem variar durante a execução do sistema e de execução para execução)

SISTEMA P2P NÃO ESTRUTURADO



Positivo:

Simplicidade de construir, robusto ao dinamismo da rede (churn – entrada e saída de nós participantes)

Negativo:

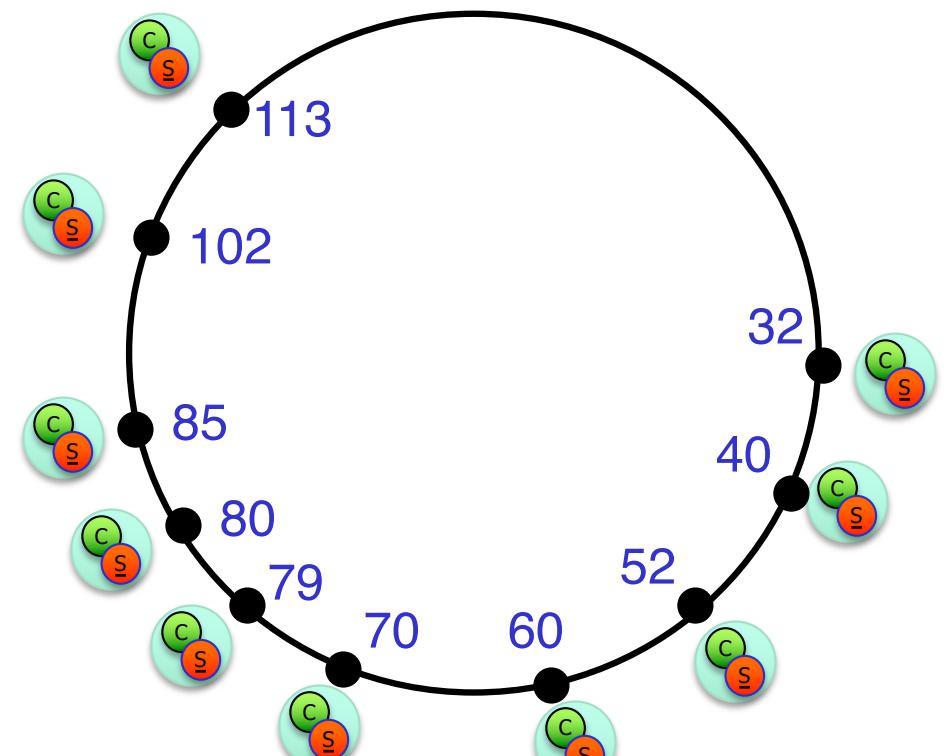
Produz topologias que podem ser difíceis de explorar de forma eficiente
eg., dificuldade em indexar informação / pesquisa pesada
(frequentemente por inundação)

SISTEMAS *P2P* ESTRUTURADOS

Os membros do sistema
comunicam de acordo com
uma organização definida de
forma determinista com base
num **endereço lógico**

A topologia é induzida por
uma relação (matemática)
entre os endereços lógicos.

Existem topologias para todos
os gostos...



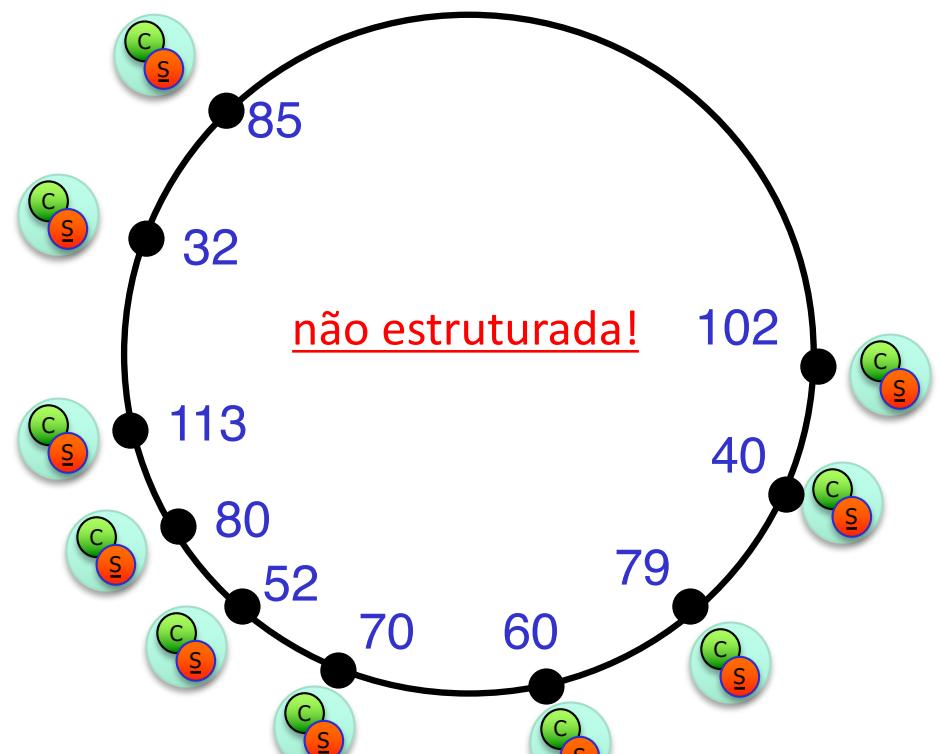
SISTEMAS *P2P* ESTRUTURADOS

Importante: A topologia (as relações de vizinhança) dos nós têm que ser induzidas pelos identificadores lógicos

Os nós podem estar organizados num anel, por exemplo, e não formar um sistema P2P estruturado...

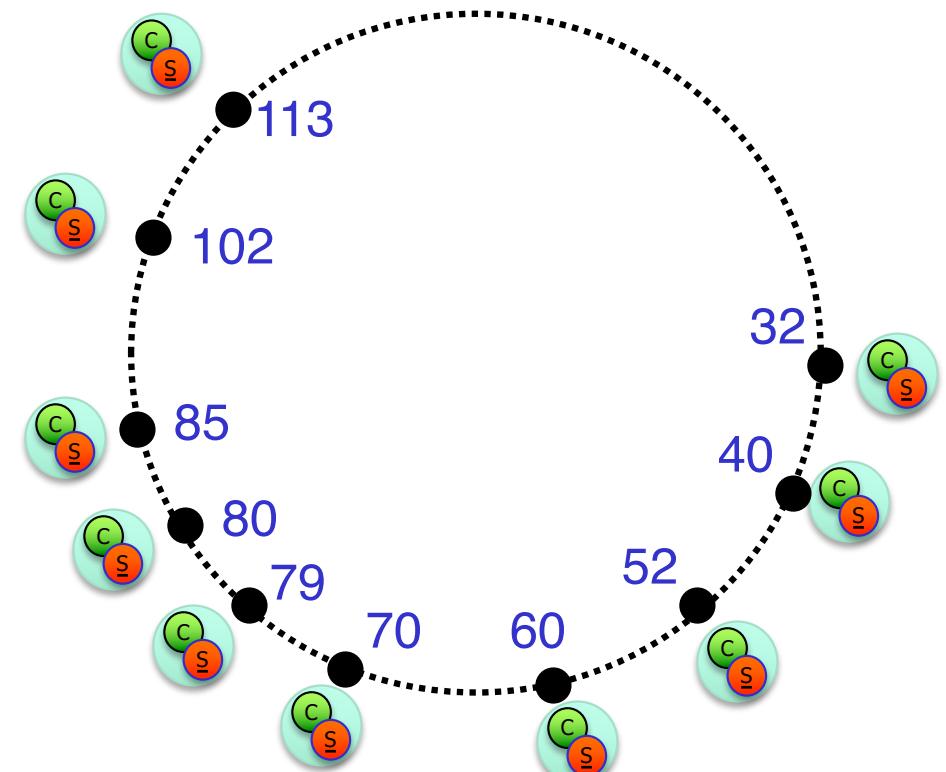
Analogia: árvore binária vs. árvore binária de pesquisa

quanto custa pesquisar um valor no primeiro caso vs. no segundo?



SISTEMAS *P2P* ESTRUTURADOS

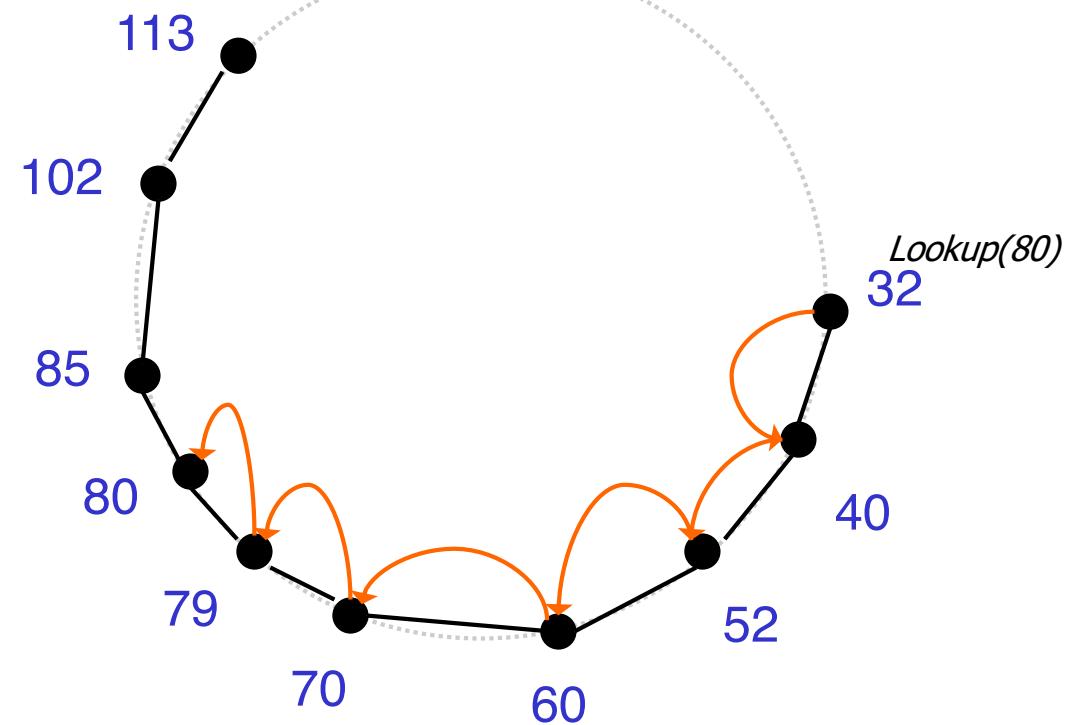
Encaminhamento e pesquisas
por identificador $O(\log N)$?



SISTEMAS *P2P* ESTRUTURADOS

Encaminhamento e pesquisas
por identificador $O(\log N)$?

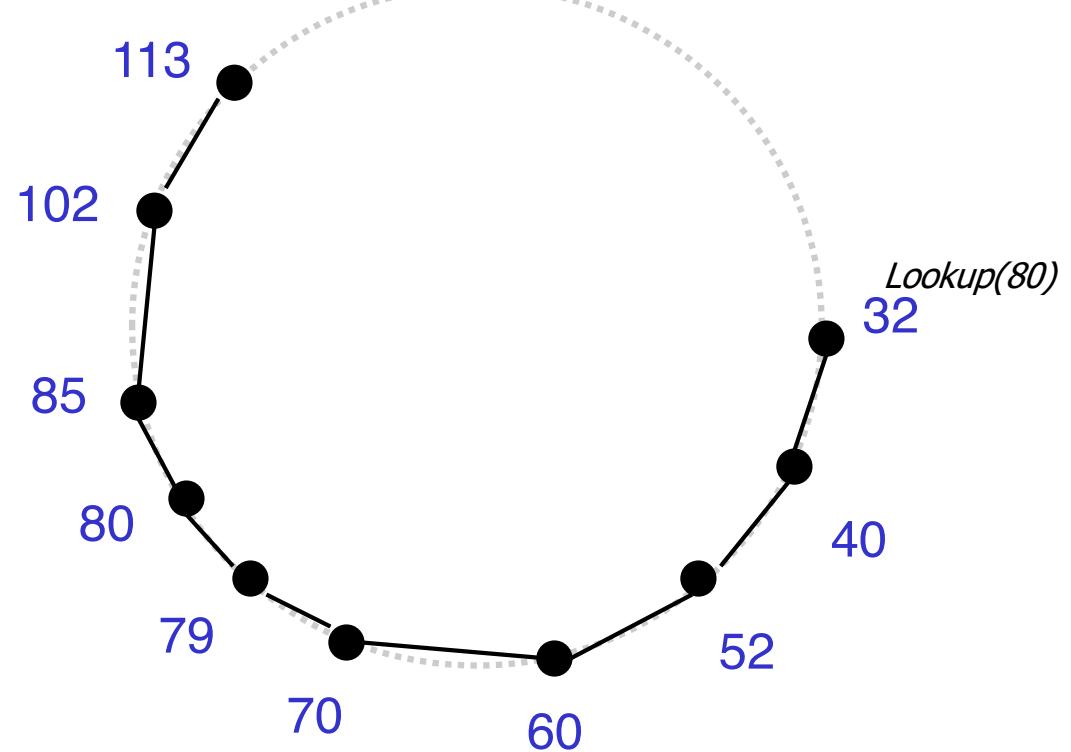
Seguir o sucessor não é
solução; tem custo $O(N)$



SISTEMAS *P2P* ESTRUTURADOS

Encaminhamento e pesquisas
por identificador $O(\log N)$?

Em cada passo é necessário
reduzir o espaço de pesquisa
para metade...

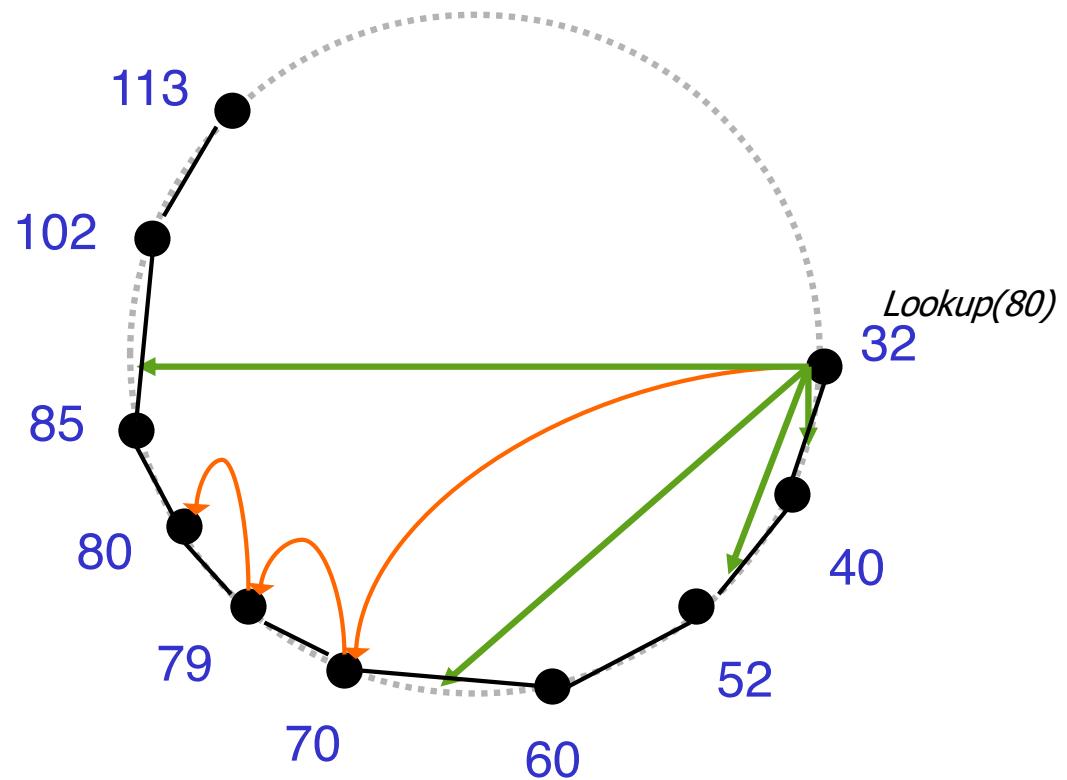


SISTEMAS *P2P* ESTRUTURADOS

Encaminhamento e pesquisas
por identificador $O(\log N)$?

Em cada passo é necessário
reduzir o espaço de pesquisa
para metade...

Ideia: cada nó liga-se a nós
noutros pontos da topologia
P2P e usa essas ligações
como atalhos: ex: $O(\log N)$
vizinhos



SISTEMAS *P2P* ESTRUTURADOS

<Simulação Sistema Chord, circa 2001>

Características:

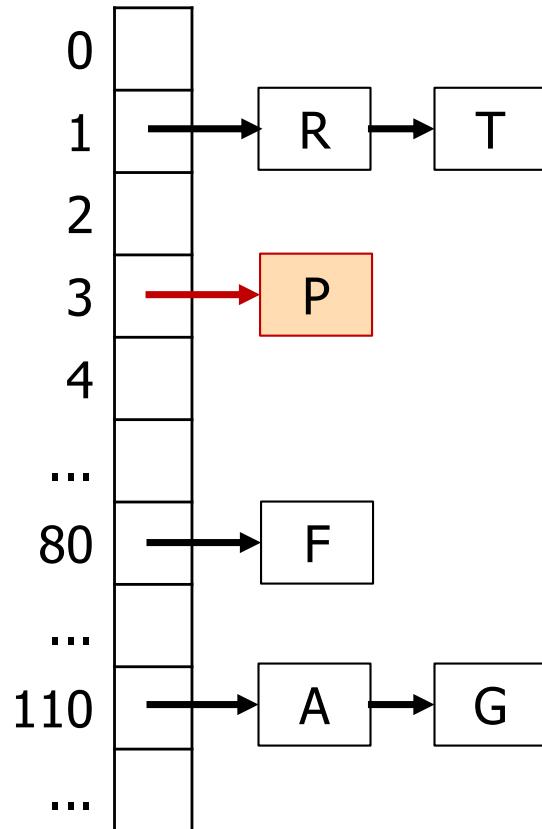
- Identificadores de 128 bits
- Tabelas de vizinhança com $O(\log N)$ peers
- Encaminhamento e Pesquisa por identificador em $O(\log N)$ passos

SISTEMAS *P2P* ESTRUTURADOS E DADOS

Um sistema P2P estruturado permite-nos descobrir um nó eficientemente (em $\log(n)$).

Como é que podemos usar esta funcionalidade para guardar informação?

DISTRIBUTED HASH TABLE (DHT)

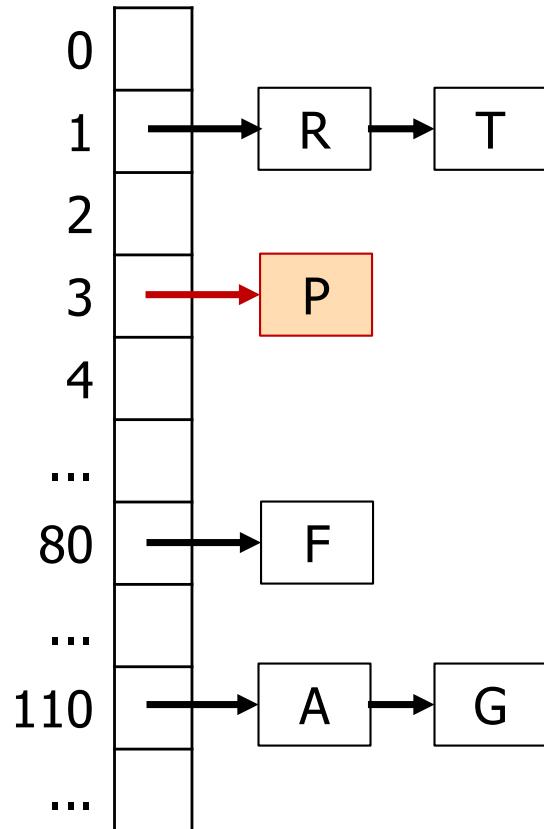


Numa **tabela de dispersão**, quando se quer guardar um valor, v , calcula-se o $\text{hash}(v)$ e guarda-se o valor nessa posição.

E.g. para inserir P , calcula-se $\text{hash}(P) = 3$, e insere-se o valor na posição respetiva.

Nota: Em geral P é um par ($\text{chave}, \text{valor}$), sendo que neste caso só se faz $\text{hash}(\text{chave})$.

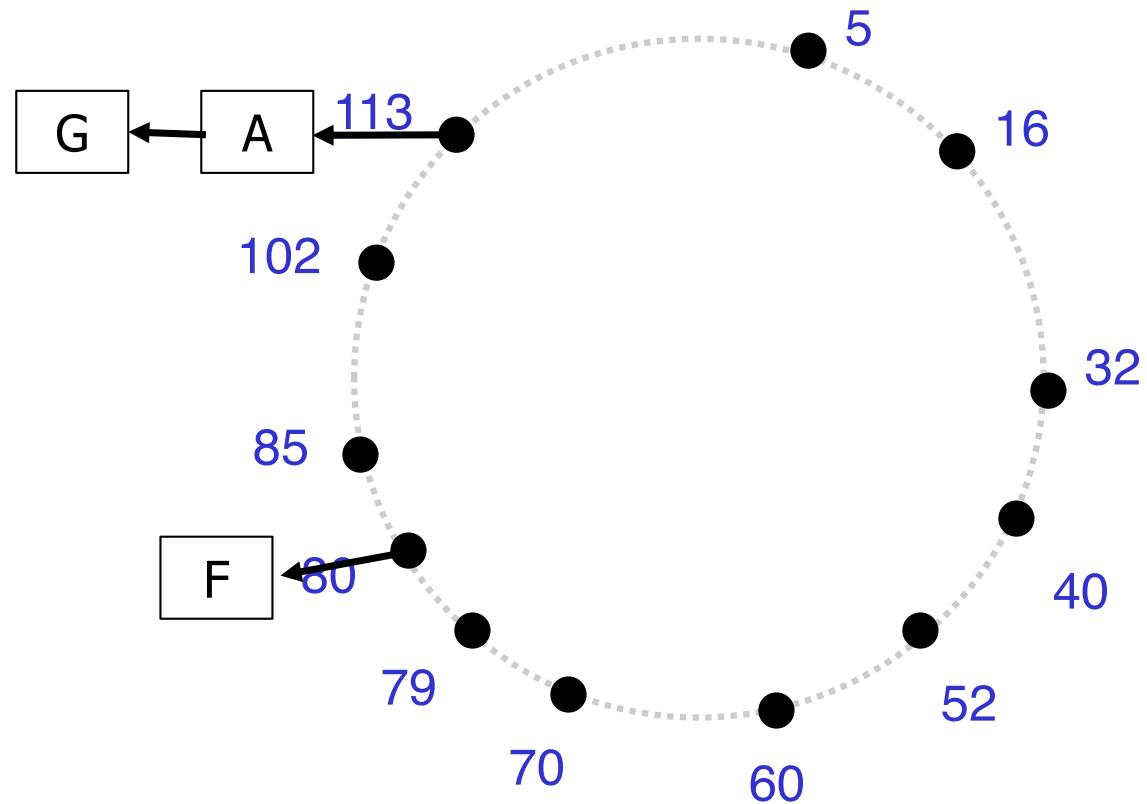
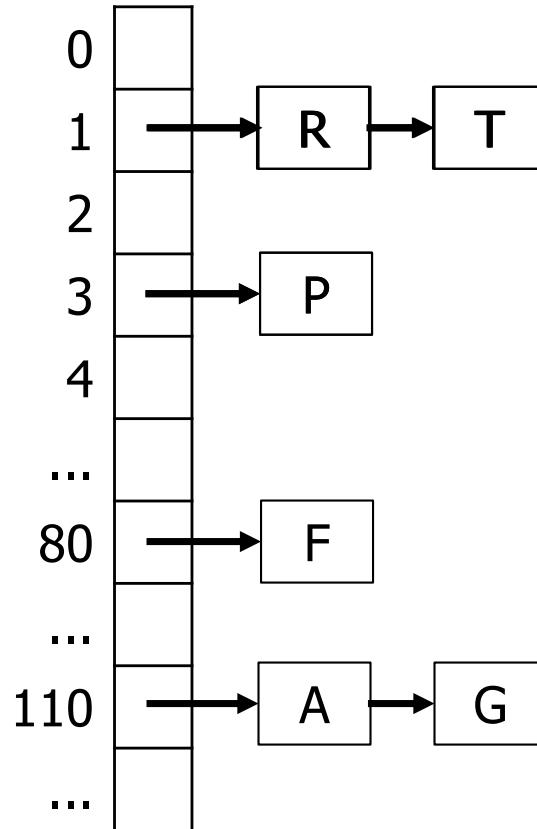
DISTRIBUTED HASH TABLE (DHT)



Numa **tabela de dispersão distribuída** (distributed hash table, DHT), vai-se usar a mesma ideia: vamos guardar ficheiros/blocos no servidor que tiver o identificador igual ao hash(chave).

DISTRIBUTED HASH TABLE (DHT)

Os dados que numa tabela de dispersão ficaria na posição n, na DHT ficam no nó com o identificador igual ou superior a n.

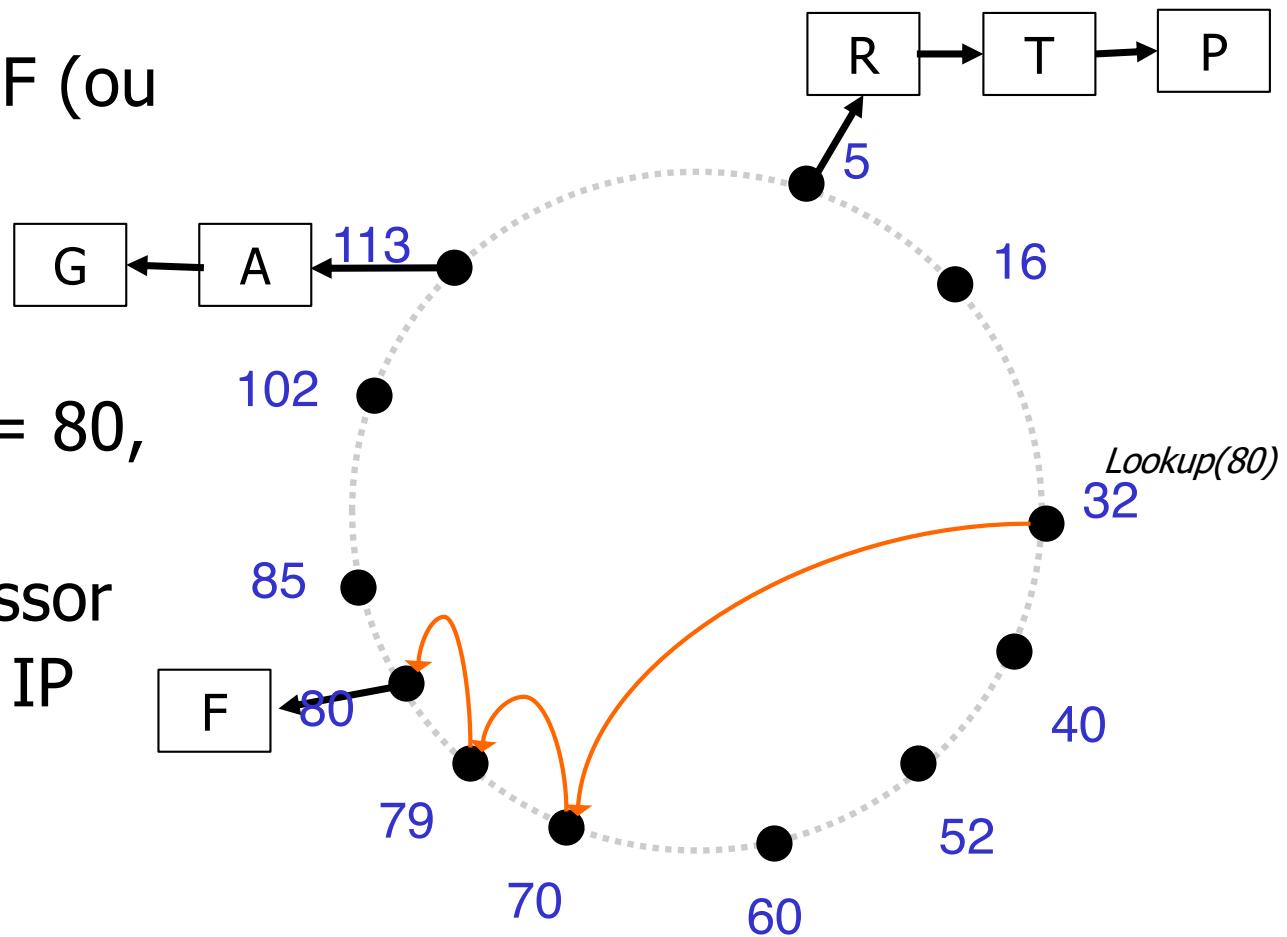


DISTRIBUTED HASH TABLE (DHT)

Para ler o ficheiro/bloco F (ou com chave F):

1. **lookup(F) -> IP₈₀**

Lookup calcula hash(F) = 80, e usa sistema P2P para procurar nó 80 (ou sucessor de 80). Sistema devolve IP do nó 80.



DISTRIBUTED HASH TABLE (DHT)

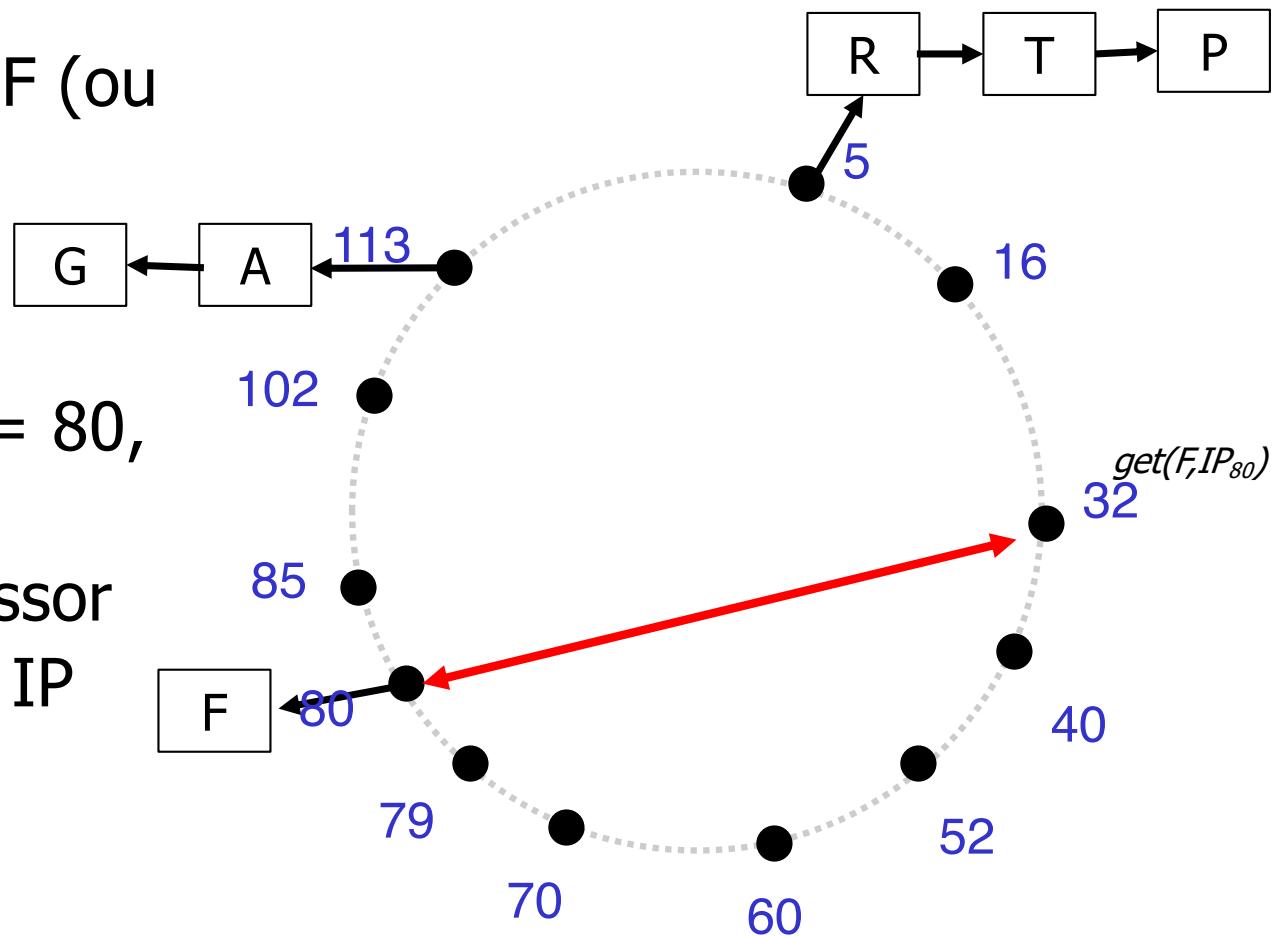
Para ler o ficheiro/bloco F (ou com chave F):

1. **lookup(F) -> IP₈₀**

Lookup calcula hash(F) = 80, e usa sistema P2P para procurar nó 80 (ou sucessor de 80). Sistema devolve IP do nó 80.

2. **get(F, IP₈₀) -> valor**

Contacta-se diretamente o nó para obter os dados.



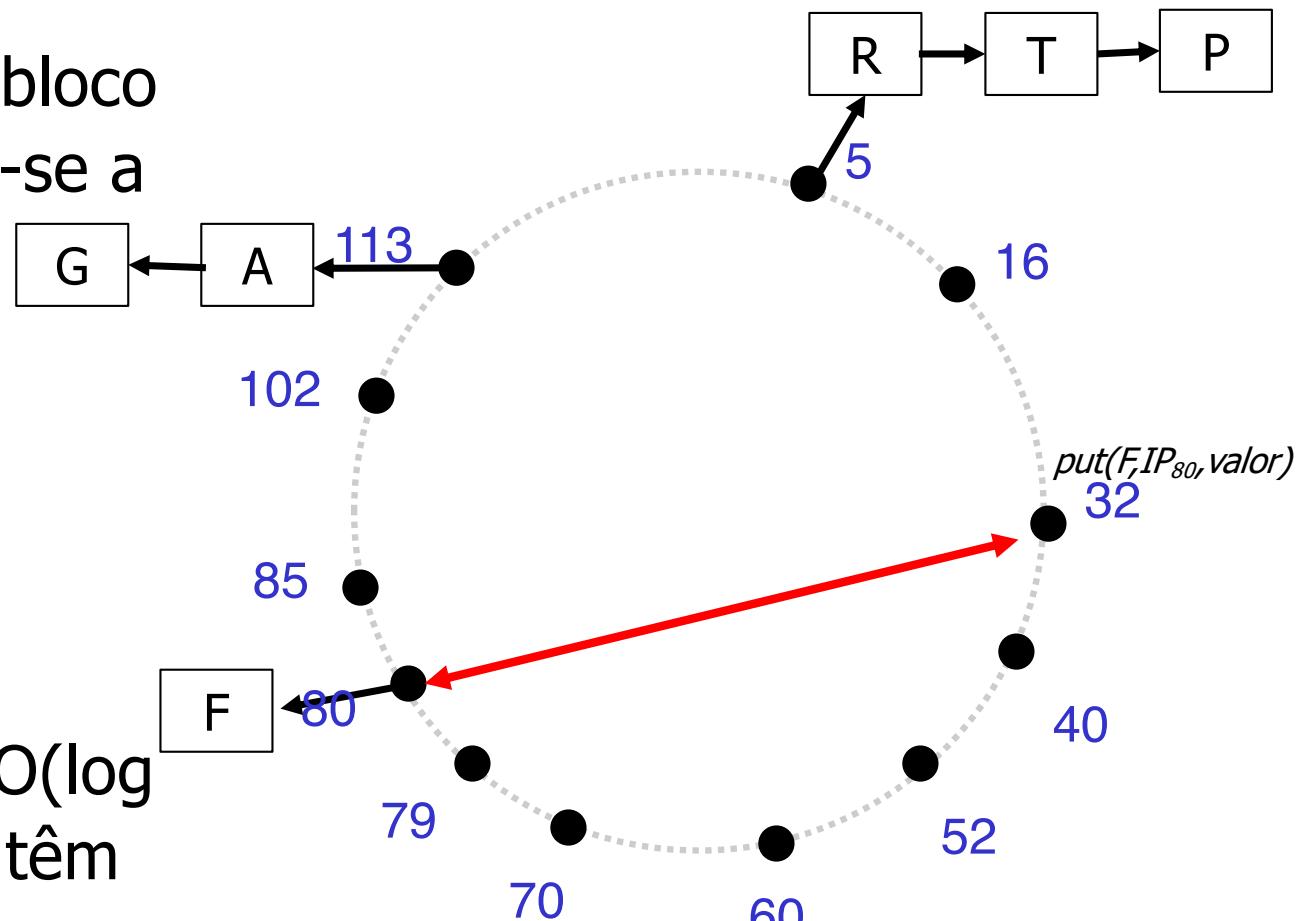
DISTRIBUTED HASH TABLE (DHT)

Para escrever o ficheiro/bloco F (ou com chave F), usa-se a mesma aproximação.

1. **lookup(F) -> IP₈₀**

2. **put(F, IP₈₀, valor)**

Lookup(key) com custo $O(\log N)$, as outras operações têm custo constante (independente de N)



DHTs: CARACTERÍSTICAS

Identifica-se a informação usando uma função de *hash*

$$\text{identificador} = \text{hash}(\text{info})$$

Cada nó fica responsável por um conjunto de identificadores (de forma determinista)

- e.g. cada nó usa um identificador único, gerado aleatoriamente, ficando responsável por manter a informação com os identificadores mais próximos do seu

Aspetos importantes...

- Pesquisa?
- Distribuição da informação?
- Replicação da informação?

DHTs: CARACTERÍSTICAS

Identifica-se a informação usando uma função de *hash*

$$\text{identificador} = \text{hash}(\text{info})$$

Cada nó fica responsável por um conjunto de identificadores (de forma determinista)

- e.g. cada nó usa um identificador único, gerado aleatoriamente, ficando responsável por manter a informação com os identificadores mais próximos do seu

Aspetos importantes...

- Pesquisa? – pesquisa por identificador deve ser eficiente e suportada por todos os nós.
- Distribuição da informação? – deve ser uniforme por todos os nós
- Replicação da informação? – a entrada e saída contínua de nós obriga a manter a informação replicada nos nós certos e em número adequado.

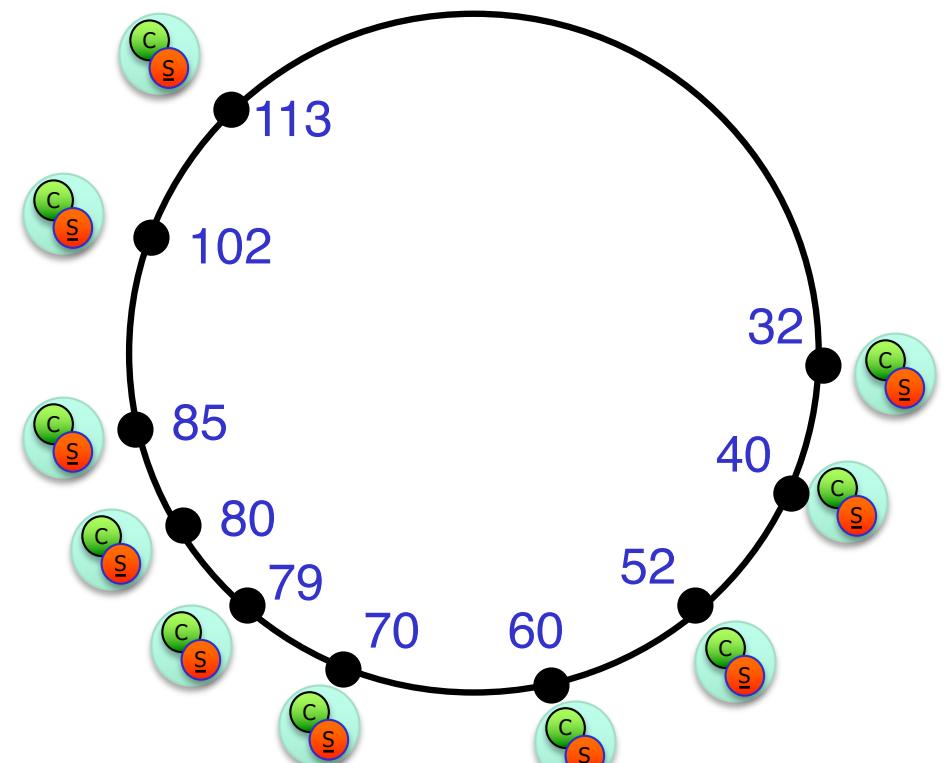
SISTEMAS *P2P* ESTRUTURADOS

Positivo

Boa latência/escalabilidade -
Conhecem-se topologias com
encaminhamento/pesquisa com
custo $O(\log n)$, com n nós no
sistema, por exemplo.

Negativo

Maior complexidade
É necessário gastar recursos
para manter a topologia correta
face às entradas e saídas dos
nós (*churn*)



Combinando os dois modelos (P2P e C/S) pode-se ter o melhor dos dois

COMBINAÇÃO DOS MODELOS C/S E P2P

Cliente + (Serviço) P2P

- Um sistema P2P pode disponibilizar um serviço a outros processos (clientes) que não pertencem ao sistema P2P

Propriedades:

- Permite limitar o número de processos que fazem parte do sistema P2P

COMBINAÇÃO DOS MODELOS C/S E P2P

Cliente/servidor + P2P

- Serviço disponibilizado por um sistema pode ser dividido em várias funcionalidades, sendo umas fornecidas por um sistema cliente/servidor e outras por um sistema P2P.
- O sistema cliente/servidor pode, por exemplo, servir como serviço de diretório, suportar pesquisas eficientes, etc.
 - e.g. BitTorrent

Propriedades:

- Permite combinar as vantagens de ambos os sistemas
- Terá contras?
- Ter também as desvantagens de ambos...

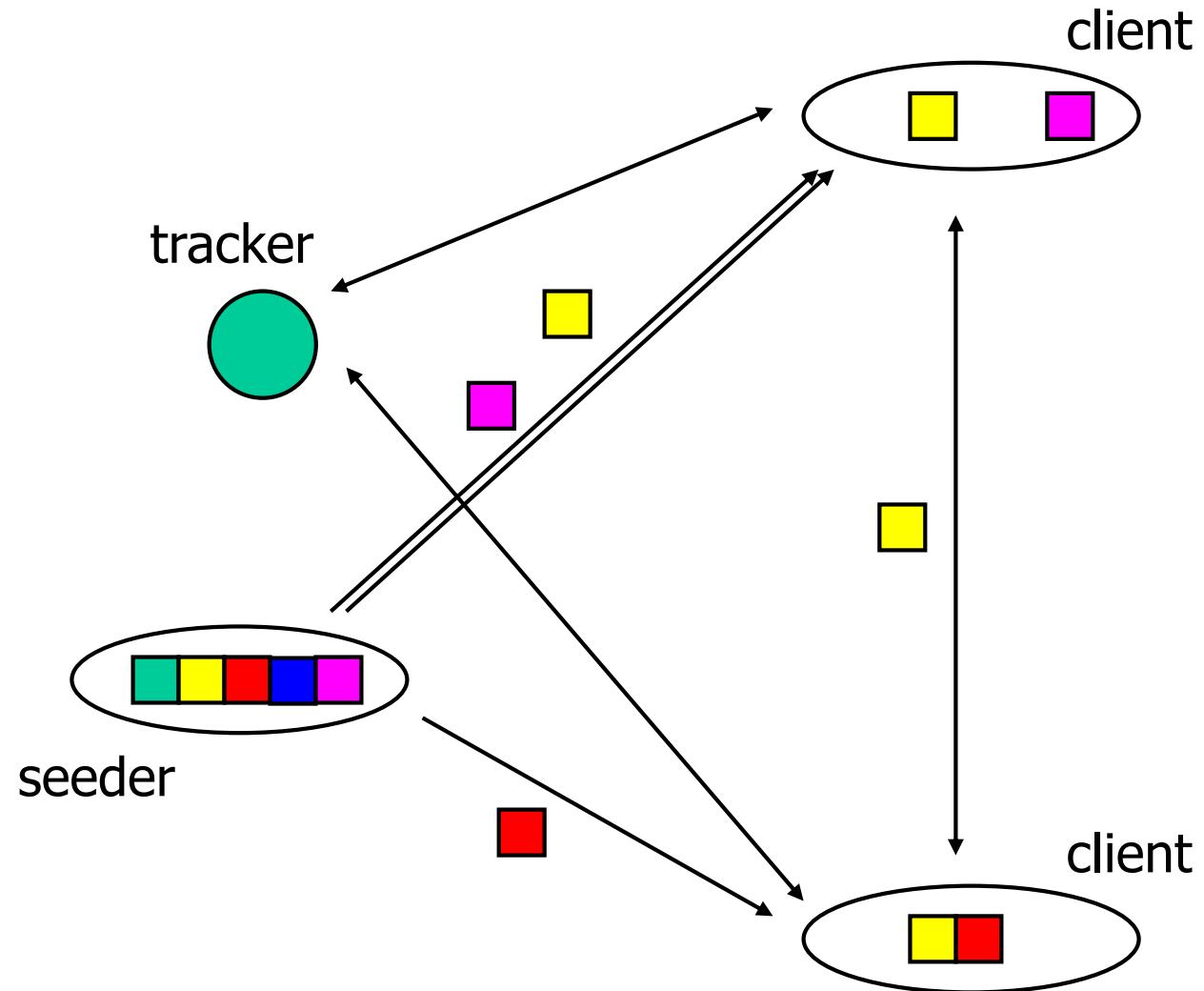
BITTORRENT-VERSÃO ORIGINAL (SIMPLIFICADO)

.torrent contém informação sobre ficheiro a ser partilhado, incluindo: url do tracker, hash seguro de cada bloco do ficheiro (SHA-1)

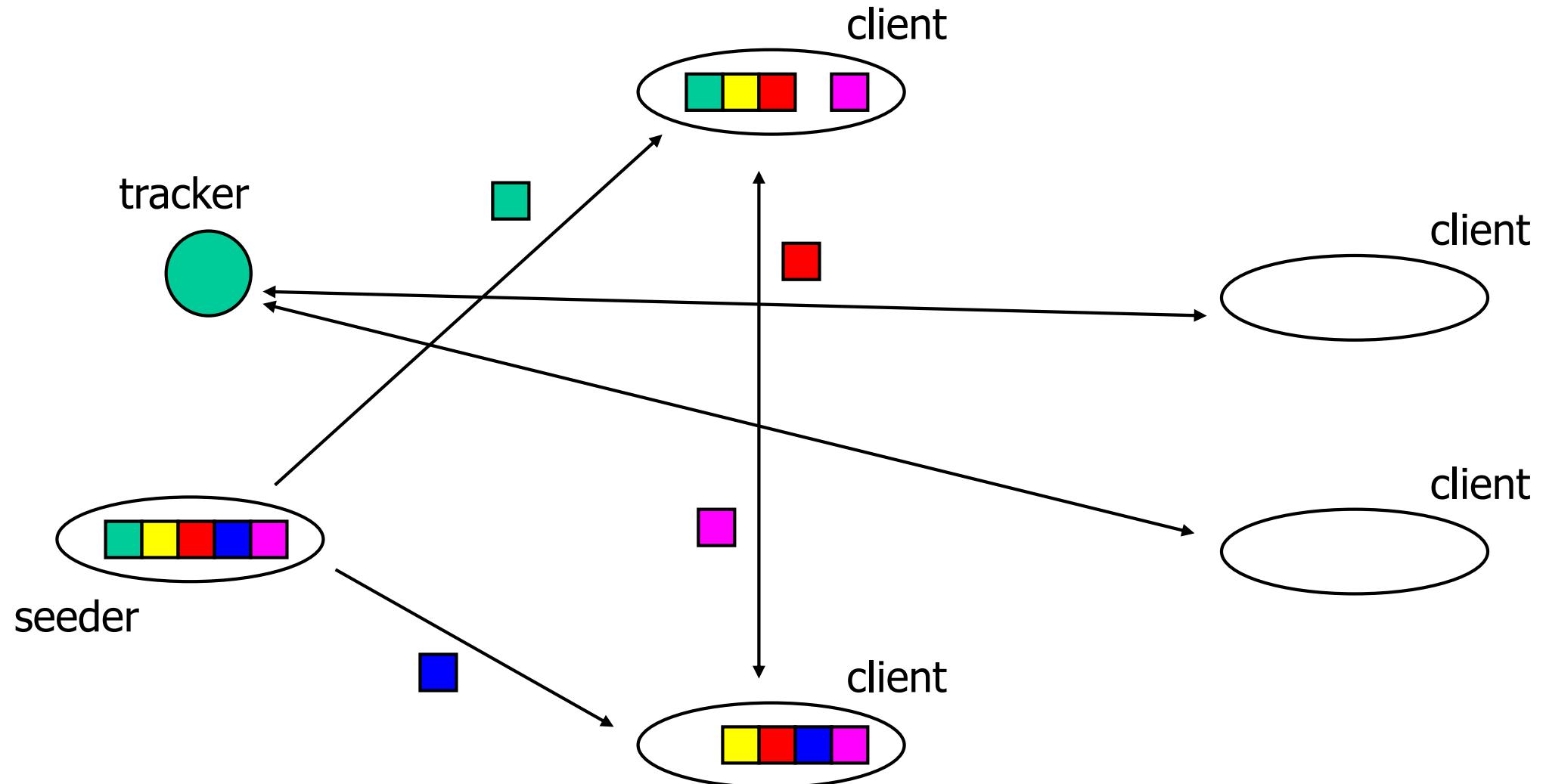
Arquitetura BitTorrent:

- C/S
 - **Tracker**: servidor centralizado HTTP que serve o ficheiro .torrent e a lista dos *peers* que estão a descarregar o ficheiro
 - **Peers**: como clientes, acedem ao *tracker* para obter a lista de outros peers a descarregar o ficheiro
- P2P
 - **Peers** comunicam entre si para trocar blocos do ficheiro
 - Toma-lá dá-cá (*tit-for-tat*): peer só envia bloco em troca de outro bloco
 - Peer envia alguns blocos a outros peers (sem contrapartida - aleatoriamente)
 - Peer descarrega blocos aleatoriamente
 - (Qual a motivação de cada uma destas propriedades?)

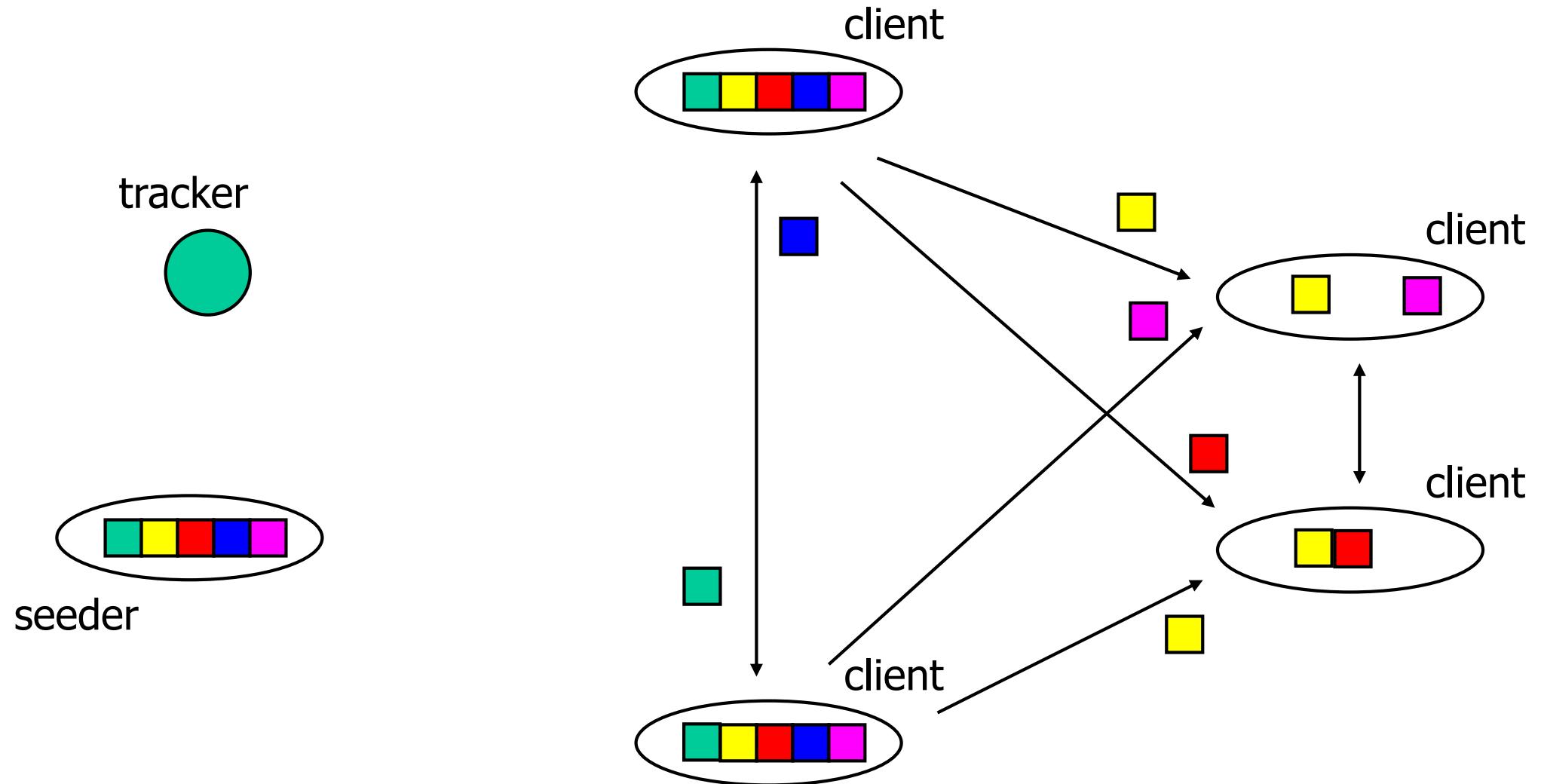
BITTORRENT



BITTORRENT



BITTORRENT



BITTORRENT DHT

BitTorrent tem a possibilidade de funcionar sem trackers

Neste caso, os peers formam uma DHT

- Cada nó gera um identificador único
- Cada *torrent* tem um identificador único
- Informação sobre quais os nós que estão a partilhar/descarregar um ficheiro/torrent é armazenada nos nós com identificadores mais próximos ao identificador do torrent
- Nota: baseado no sistema Kademlia (circa 2002)

AINDA OUTROS EXEMPLOS...

Sistemas peer-to-peer hierárquicos

- Subconjunto de super-nós que se agrupam como num sistema peer-to-peer
- Nós ligam-se a um super-nó

...

KAZAA / SKYPE (ORIGINAL)

Arquitetura:

Super-nós (SN): nós com maior capacidade tornam-se super-nós

Cada SN tem 60-150 nós ligados

Cada SN liga-se a outros 30-50 SN

Peers (ordinary node – ON)

ON liga-se a um dos SNs que conhece
(após se ter ligado, atualiza lista de SNs)

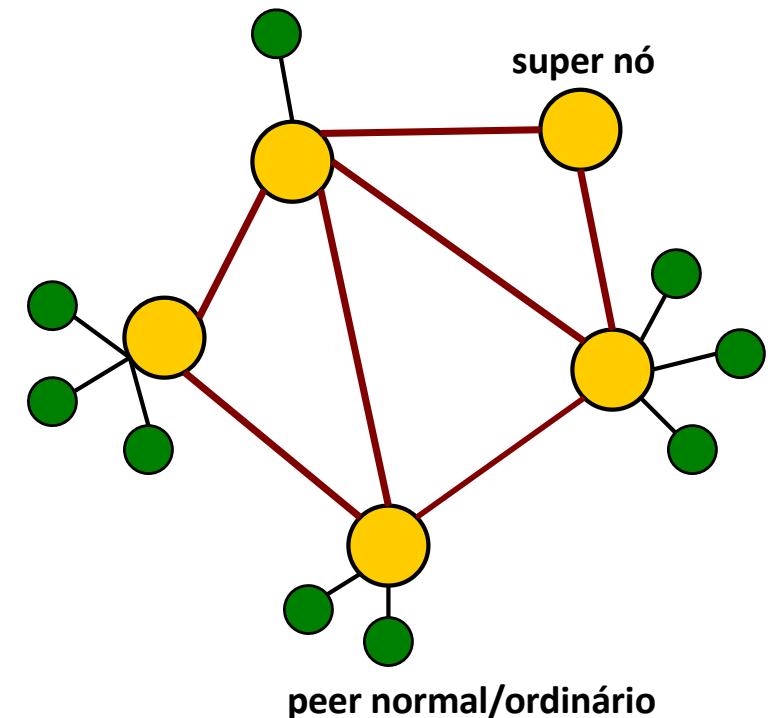
Pesquisa:

ON contacta o seu SN

SNs propagam pedidos entre si

Transferência de ficheiros:

Directamente entre **peers**



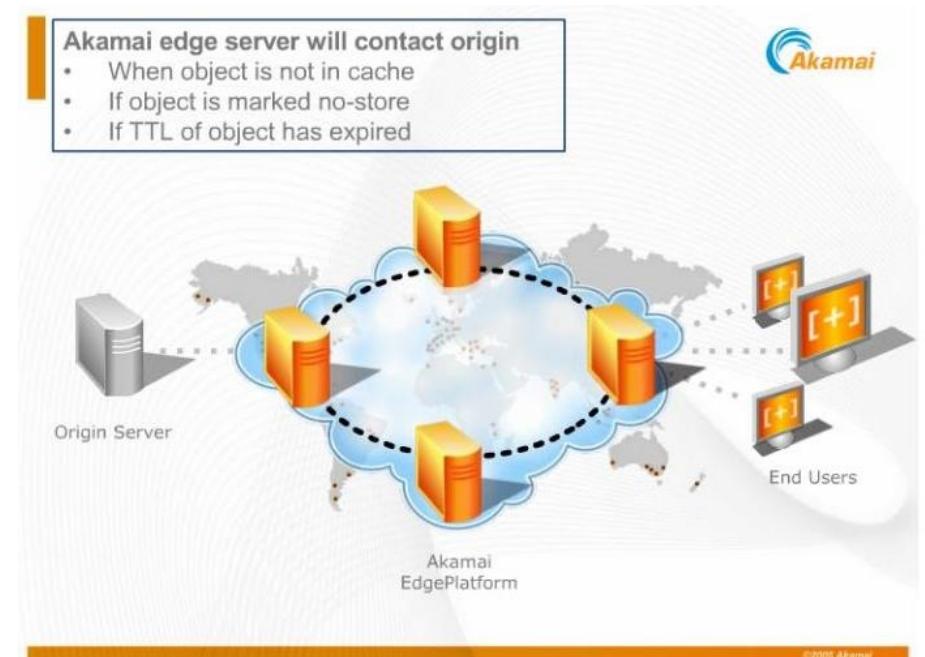
Na prática, tendem a existir mais componentes num sistema distribuído

EDGE-SERVER

Sistemas edge-server

- Existem servidores colocados nos ISP para responder a pedidos
 - e.g., content-distribution networks (CDNs)
- Propriedades
 - Menor latência, filtragem, distribuição de carga, etc.

Suportam **distribuição de conteúdos estáticos** de grande volume e/ou popularidade:
eg., streaming: Youtube, Netflix



src: <https://clickmotive.files.wordpress.com/2009/11/akamaiedgeplatform.jpg>

SISTEMAS DISTRIBUÍDOS

Capítulo 3

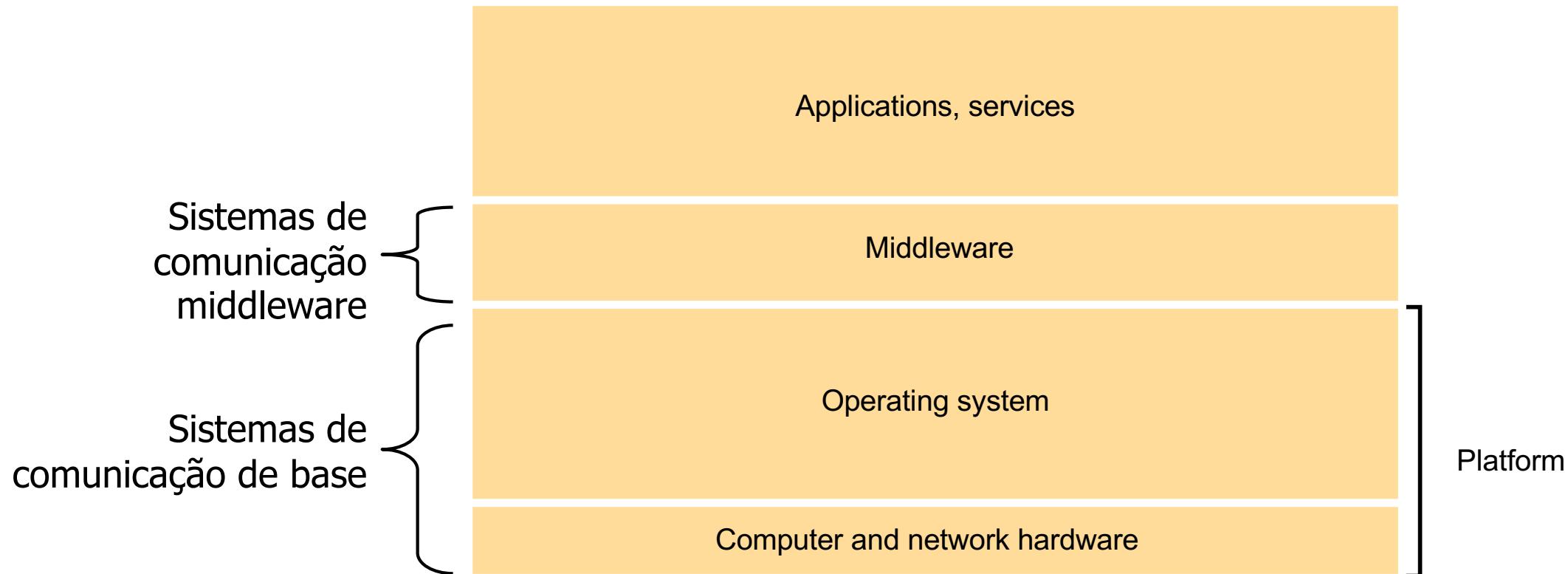
Comunicação direta

NOTA PRÉVIA

A estrutura da apresentação é semelhante e utiliza algumas das figuras do livro de base do curso

G. Coulouris, J. Dollimore and T. Kindberg,
Distributed Systems - Concepts and Design,
Addison-Wesley, 4th Edition, 2005

COMUNICAÇÃO NUM SISTEMA DISTRIBUÍDOS



SISTEMAS DE COMUNICAÇÃO DE BASE

Os sistemas de operação podem suportar a comunicação de dados entre os diferentes computadores envolvidos num sistema distribuído.

Protocolos mais populares:

- TPC/IP
- HTTP

TCP/IP: UDP

Comunicação por mensagens.

Mensagens podem-se perder, duplicar e chegar fora de ordem.

```
DatagramSocket socket = new DatagramSocket( 9000 ) ;  
  
byte[] buffer = new byte[1500] ;  
DatagramPacket packet = new DatagramPacket( buffer, buffer.length ) ;  
socket.receive( packet ) ;
```

```
byte[] msg = ...  
DatagramSocket socket = new DatagramSocket() ;  
  
DatagramPacket packet = new DatagramPacket( msg, msg.length ) ;  
packet.setAddress( InetAddress.getByName( "servername" ) ) ;  
packet.setPort( 9000 ) ;  
socket.send( packet ) ;
```

TCP/IP: IP MULTICAST

Comunicação por mensagens com múltiplos receptores.

Cliente envia mensagem para endereço do grupo. Qualquer processo se pode juntar ao grupo para receber mensagens.

Mensagens podem-se perder, duplicar e chegar fora de ordem.

```
MulticastSocket socket = new MulticastSocket( 9000 ) ;  
socket.joinGroup( InetAddress.getByName( "225.10.10.10" ) );  
  
byte[] buffer = new byte[1500] ;  
DatagramPacket packet = new DatagramPacket( buffer, buffer.length ) ;  
socket.receive( packet ) ;
```

```
byte[] msg = ...  
MulticastSocket socket = new MulticastSocket() ;  
  
DatagramPacket packet = new DatagramPacket( msg, msg.length ) ;  
packet.setAddress( InetAddress.getByName( "225.10.10.10" ) ) ;  
packet.setPort( 9000 ) ;  
socket.send( packet ) ;
```

TCP/IP: TCP

Dados transmitidos como fluxo contínuo.

Dados chegam de forma fiável a menos que o stream seja quebrado.

```
ServerSocket ss = new ServerSocket( 9000 ) ;
while( true ) {
    Socket cs = ss.accept() ;
    ....
}
```

```
byte[] msg = ...
Socket cs = new Socket("servername", 9000) ;
OutputStream os = cs.getOutputStream() ;
InputStream is = cs.getInputStream() ;

os.write( msg )
int b = is.read();
```

HTTP

Comunicação pedido/resposta sobre TCP, invocando URL.

Dados chegam de forma fiável a menos que o stream seja quebrado.

```
HttpURLConnection con = (HttpURLConnection)
    new URL( "http://asc.di.fct.unl.pt/sd/xpto").openConnection();
con.setRequestMethod("POST");
con.setDoOutput(true);
con.setDoInput(true);
OutputStream os = con.getOutputStream();
....
os.flush();
InputStream is = con.getInputStream();
....
```

HTTP ASSÍNCRONO

Comunicação pedido/resposta, com resposta a ser recebida de forma assíncrona.

- Qual o interesse?

Solução adotada nos browser: JavaScript nativo ou bibliotecas JavaScript (e.g. Jquery)

```
[javascript]
var url = ...
var xmlhttp = new XMLHttpRequest();
xmlhttp.onreadystatechange = function() {
    if (xmlhttp.readyState == 4 && xmlhttp.status == 200) {
        result = xmlhttp.responseText;
        // process result
    }
};
xmlhttp.open("GET", url, true);
xmlhttp.responseType = "json";
xmlhttp.send();
```

readyState
0: request not initialized
1: server connection established
2: request received
3: processing request
4: request finished and response is ready

WEB SOCKETS

Comunicação full-duplex sobre TCP entre clientes e servidores Web.

- Permite notificações dos servidores, streaming.

Suporte generalizado nos browsers.

```
[javascript]
var ws = new WebSocket("ws://asc.di.fct.unl.pt/websocket");

ws.onopen = function() {
    ws.send("Connecting... ");
};

ws.onmessage = function (evt) {
    var received_msg = evt.data;
    ...
};

ws.onclose = function() {
    ...
};
```

COMUNICAÇÃO NO NÍVEL MIDDLEWARE

Implementa sistema de comunicação recorrendo às primitivas de comunicação base

Fornece propriedades adicionais, atrasando a entrega das mensagens

- **Definição:** Entrega de uma mensagem num sistema de comunicação representa a ação do sistema disponibilizar a mensagem para ser lida pelas aplicações
- Atrasar a entrega de uma mensagem pode, por exemplo, permitir que a **ordem de entrega** das mensagens seja diferente da **ordem de chegada**.

FACETAS DA COMUNICAÇÃO

Forma da interação

- Streams
- Mensagens
 - Ordenação das mensagens

Número de destinatários

- Ponto-a-ponto
- Multi-ponto (estudado mais tarde)
- Um-de-muitos (anycast)

Direção de interação

- Uni-direcional
- Bi-direcional

Tipo de sincronização

- Comunicação síncrona
- Comunicação assíncrona

Persistência

- Comunicação persistente
- Comunicação volátil

Fiabilidade (modelo de falhas)

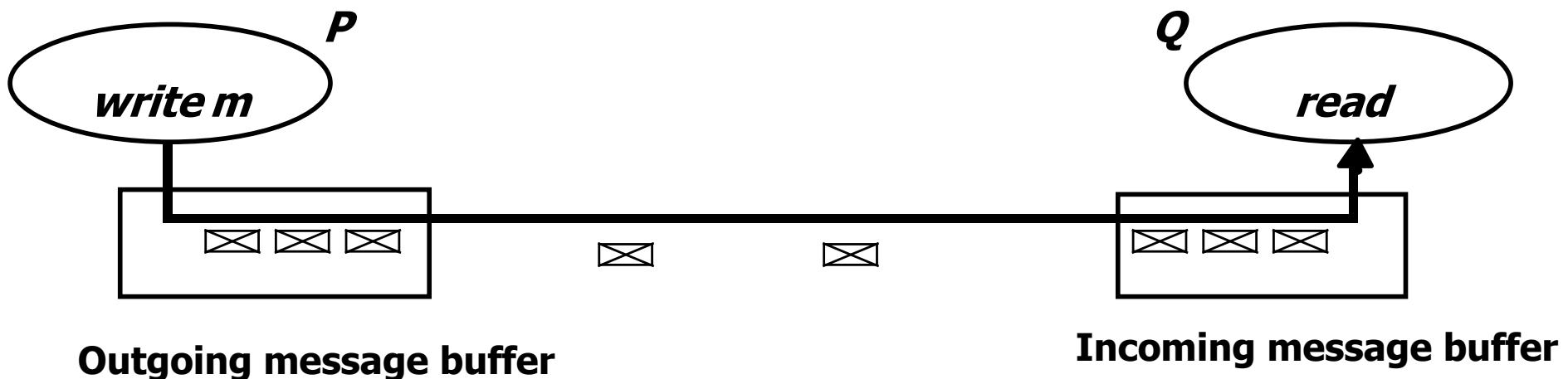
FORMA DE INTERAÇÃO: STREAMS

Emissor e receptor estabelecem um fluxo contínuo de dados

Ordem dos dados enviados é mantida;

Fronteira das escritas dos dados não é preservada.

Exemplos de situações em que é apropriado?



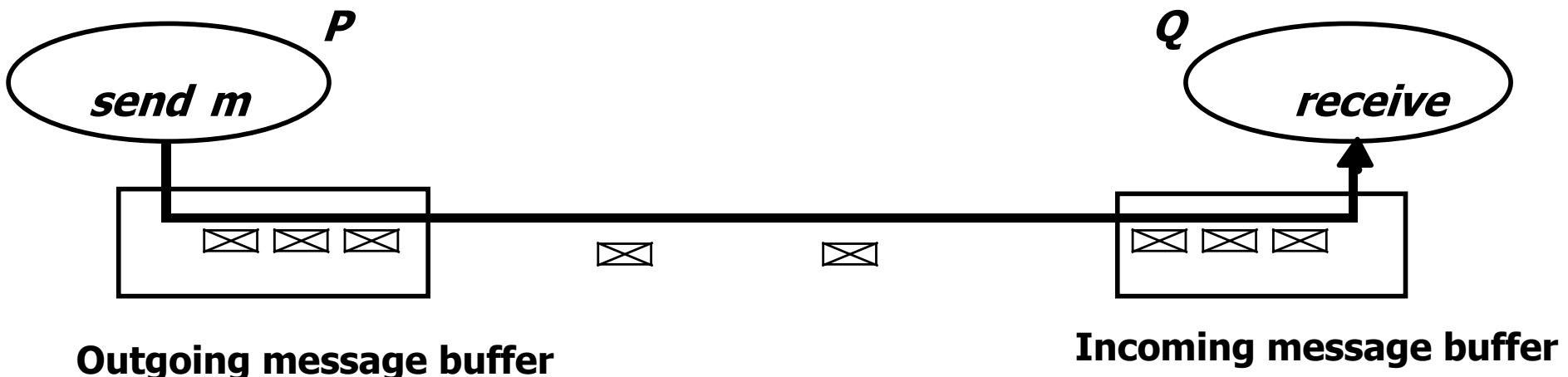
FORMA DE INTERAÇÃO: MENSAGENS

Emissor e receptor comunicam trocando mensagens

Cada mensagem tem um limite (e dimensão) bem-definida.

Exemplos de situações em que é apropriado?

Como implementar sobre TCP?



FORMA DE INTERAÇÃO: MENSAGENS

Emissor e receptor comunicam trocando mensagens

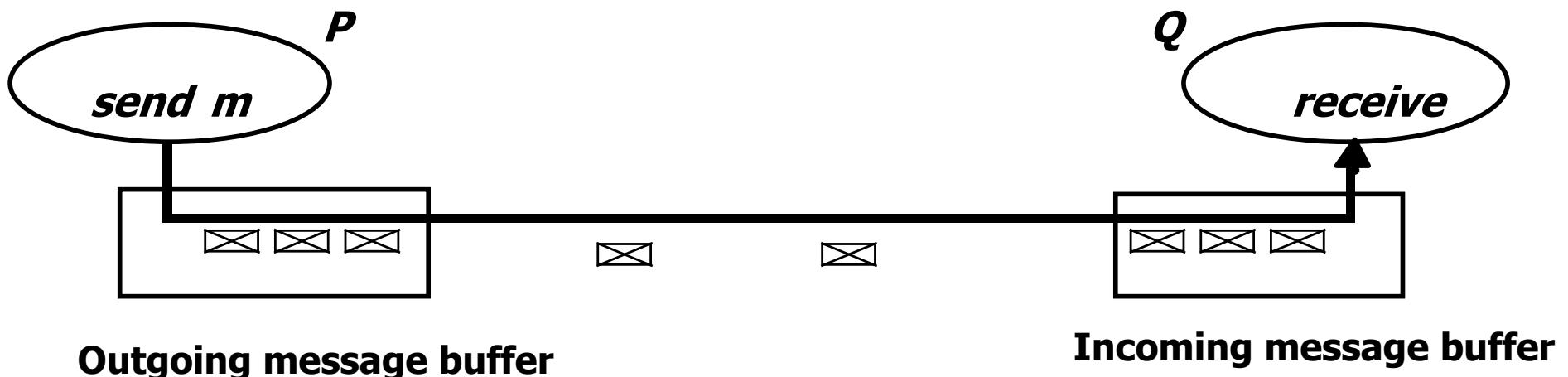
Cada mensagem tem um limite (e dimensão) bem-definida.

Exemplos de situações em que é apropriado?

Como implementar sobre TCP?

<dimensão, dados> ou <dados,delimitador> ou ...

vantagens? desvantagens de cada opção?



FORMA DE INTERAÇÃO: MENSAGENS

Emissor e receptor comunicam trocando mensagens

Cada mensagem tem um limite (e dimensão) bem-definida.

Exemplos de situações em que é apropriado?

Como implementar sobre TCP?

<dimensão, dados> ou <dados,delimitador> ou ...

No primeiro caso é necessário conhecer a priori a dimensão da mensagem. Para mensagens grandes, geradas dinamicamente poderá obrigar a ter a mensagem em memória antes de a poder enviar. Para mensagens de pequena dimensão é uma boa solução.

No segundo caso, é preciso garantir que o delimitador não ocorre dentro da mensagem, sob pena de o emissor e o receptores ficarem dessincronizados.

Uma terceira abordagem, adequada para mensagens de grande dimensão e geradas dinamicamente, pode-ase partir a mensagem numa sequência de blocos de dimensão fixa (pequena), mantendo apenas o último bloco em memória.

FORMA DE INTERAÇÃO: ORDENAÇÃO DAS MENSAGENS

Sem garantias de ordem

- Sistema não garante que as mensagens são entregues pela ordem que foram enviadas

Entrega pela mesma ordem da emissão – FIFO (first in first out)

- Sistema garante que as mensagens dum emissor são entregues pela mesma ordem que foram enviadas. Como implementar em TCP/UDP?
- Haverá outras garantias de ordem?

NÚMERO DE DESTINATÁRIOS

Comunicação ponto-a-ponto

- Comunicação entre um emissor e um receptor

Comunicação multi-ponto

- Comunicação entre um emissor e um conjunto de receptores
- **Broadcast**: envio de 1 emissor para todos os receptores
- **Multicast**: envio de 1 emissor para todos os receptores de um grupo
- **Anycast**: envio de 1 emissor para um receptor de um grupo

DIRECÇÃO DE INTERAÇÃO

Comunicação uni-direccional:

- Comunicação apenas num sentido: emissor->recetor

Comunicação bi-direccional:

- Comunicação nos dois sentidos

SÍNCRONIZAÇÃO

Comunicação assíncrona:

- o emissor só fica bloqueado até o seu pedido de envio ser tomado em consideração
- o recetor fica bloqueado até ser possível receber dados
 - Em geral, o sistema de comunicação do receptor armazena (algumas) mensagens caso não exista nenhum recetor bloqueado no momento da sua recepção. Assim, funciona como um *buffer* entre o emissor e o recetor
 - É possível variante em que o recetor não fica bloqueado e devolve erro ou a receção é efectuada em background

Comunicação síncrona:

- o emissor fica bloqueado até:
 - o recetor “receber” os dados – comunicação síncrona unidireccional
 - receber a resposta do receptor – comunicação pedido / resposta ou cliente / servidor
- o receptor fica bloqueado até ser possível consumir dados

PERSISTÊNCIA

Comunicação volátil: mensagens apenas são encaminhadas se o receptor existir e estiver a executar, caso contrário são destruídas.

- Exemplo: ???

Comunicação persistente: mensagens são guardadas pelo sistema de comunicação até serem consumidas pelos destinatários, que podem não estar a executar. Mensagens são guardadas num receptáculo independente do receptor – mailbox, canal, porta persistente, etc.

- Exemplo: ???

FIABILIDADE

Comunicação fiável: o sistema garante a entrega das mensagens em caso de falha temporária. Como implementar?

Comunicação não-fiável: em caso de falha, as mensagens podem-se perder

PARA SABER MAIS

George Coulouris, Jean Dollimore, Tim Kindberg and Gordon Blair,

Distributed Systems – Concepts and Design,
Addison-Wesley, 5th Edition, 2011

- Capítulo 4.1-4.3.

SISTEMAS DISTRIBUÍDOS

Capítulo 4

Invocação remota

NOTA PRÉVIA

A estrutura da apresentação é semelhante e utiliza algumas das figuras do livro de base do curso

G. Coulouris, J. Dollimore and T. Kindberg,
Distributed Systems - Concepts and Design,
Addison-Wesley, 5th Edition, 2009

Para saber mais:

- RMI/RPCs - capítulo 5.
- Representação de dados e protocolos - capítulo 4.3.
- Web services – capítulo 9

NA ÚLTIMA AULA

Mecanismos de comunicação base

- TCP, UDP, IP multicast
- HTTP assíncrono, Web sockets

Propriedade dos sistemas de comunicação

- **Forma da interação:** streams, mensagens
- **Número de destinatários:** ponto-a-ponto, multi-ponto, um-de-muitos
- **Direção de interação:** uni-directional, bi-direcional
- **Tipo de sincronização:** comunicação síncrona, comunicação assíncrona
- **Persistência:** comunicação persistente, comunicação volátil
- **Fiabilidade:** fiável, não fiável

MOTIVAÇÃO

Estruturar uma aplicação distribuída com base nas mensagens trocadas pelos seus componentes é a abordagem mais óvia, mas:

- exige atenção a muitos detalhes de baixo nível; a estrutura dos programas espelha os padrões de comunicação, em vez da lógica da aplicação no seu todo.
- em particular, os servidores ficam estruturados em função das mensagens que sabem tratar.

PROBLEMAS ?

De aplicação para aplicação, verifica-se que muitas linhas de código são *repetitivas*, não contêm nenhum significado aplicacional específico e referem-se ao processamento das comunicações.

Em particular, boa parte do código está dedicado a:

- criação de *communication end points* e sua associação aos processos criação, preenchimento e interpretação das mensagens;
- selecção do código a executar consoante o tipo da mensagem recebida gestão de temporizadores/tratamento das falhas

OBJETIVO

- Não será possível automatizar aquilo que é repetitivo?
- Não será possível que o programador apenas especifique o código aplicacional?

INVOCAÇÃO REMOTA

Nas linguagem imperativas definem-se funções / procedimentos / métodos para executar uma dada operação. Num ambiente distribuído, uma extensão natural consiste em permitir que a **execução ocorra noutra máquina**.

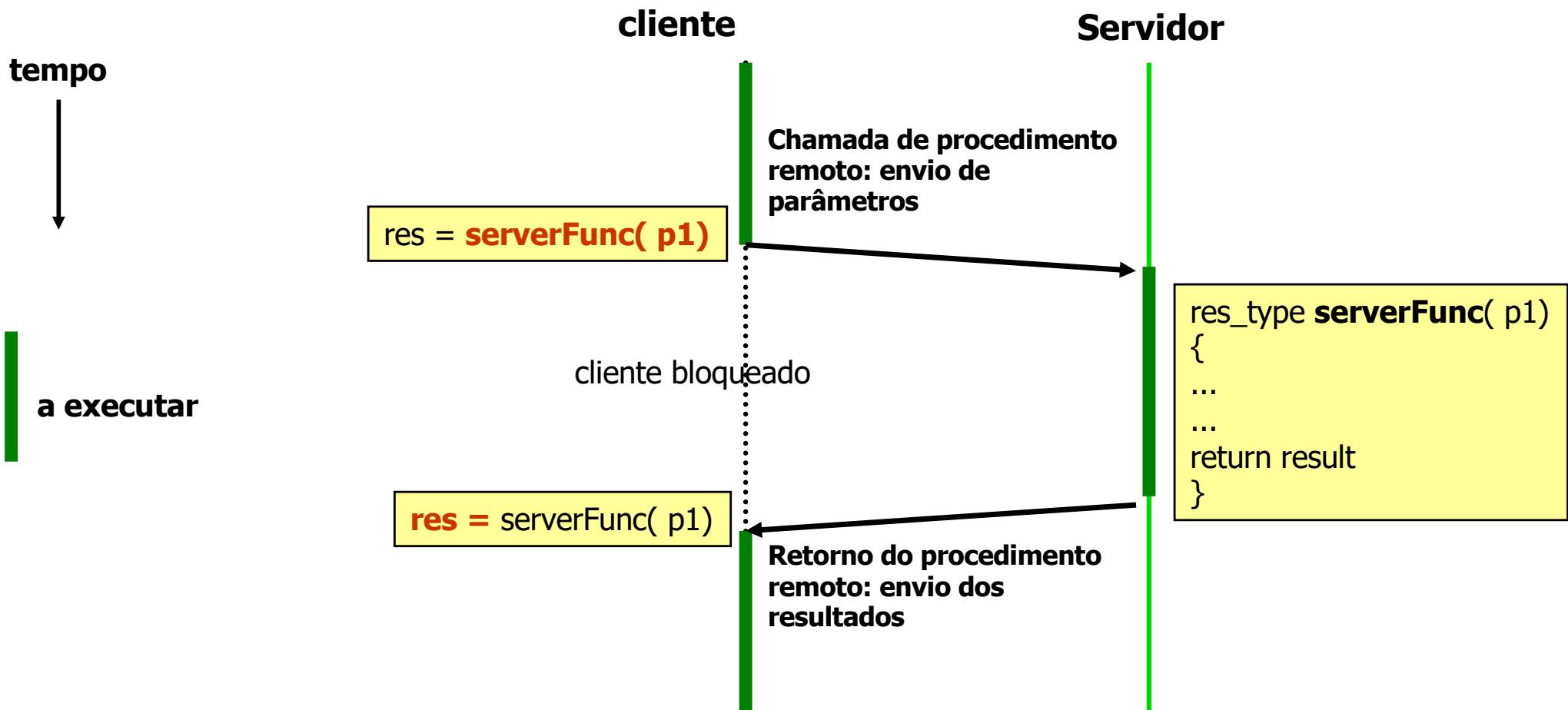
Invocação Remota de Procedimentos (RPCs), quando são executados funções/procedimentos remotamente.

- gRPC (<https://grpc.io/>), ONC/RPC, DCE

Invocação Remota de Métodos (RMI), quando são executados métodos de objetos remotos.

- JAVA RMI, .NET Remoting, Corba.
- Web Services (REST e SOAP)

INVOCAÇÃO DE PROCEDIMENTOS REMOTOS (RPCs)



Modelo

Servidor exporta interface com operações que sabe executar

Cliente invoca operações que são executadas remotamente e (normalmente) aguarda pelo resultado

INVOCAÇÃO REMOTA - PROPRIEDADES

Extensão natural do paradigma imperativo/procedimental a um ambiente distribuído

- Modelo síncrono de comunicação suporta chamadas bloqueantes no cliente

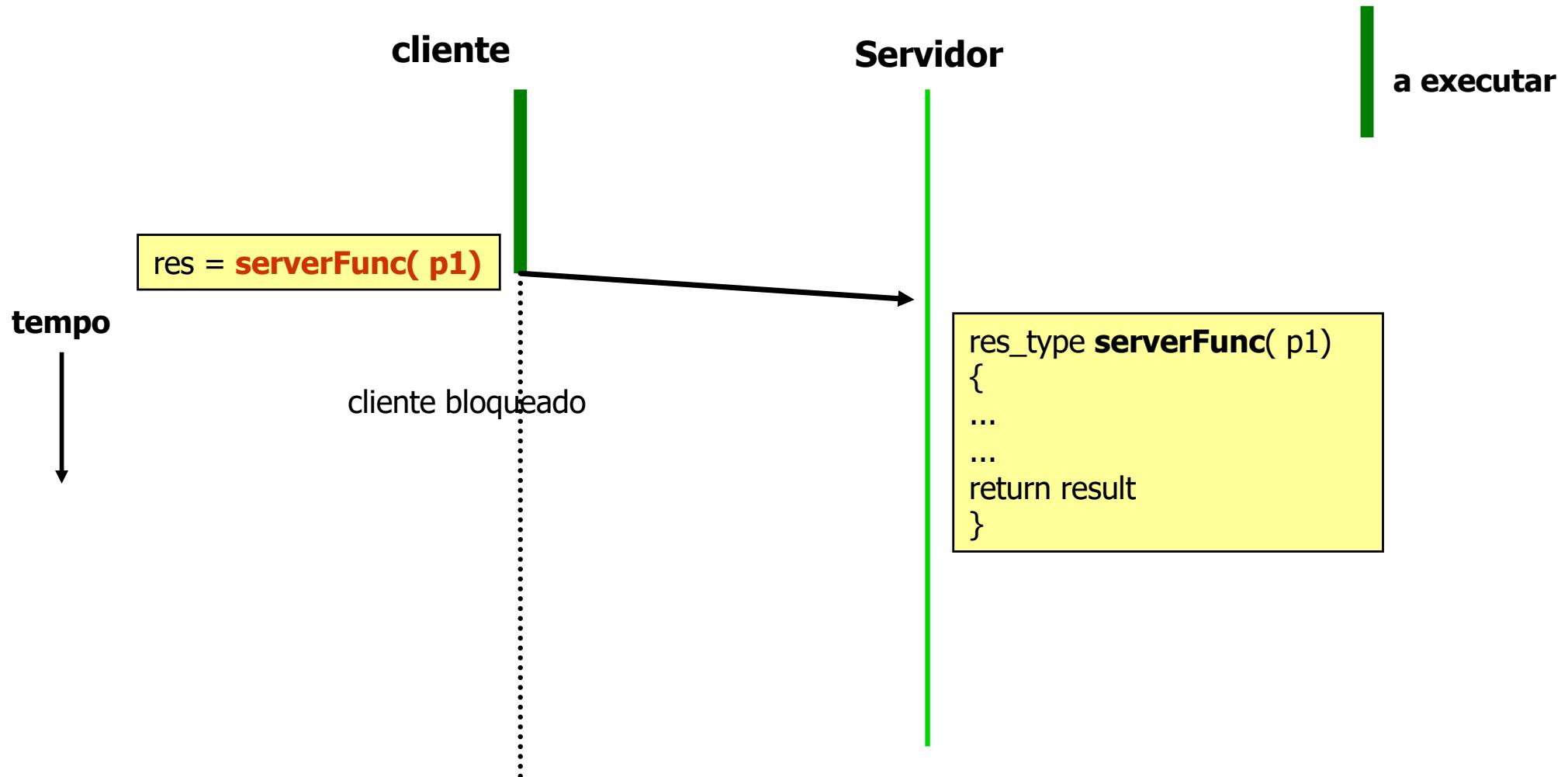
Esconde detalhes de comunicação (e tarefas repetitivas)

- Construção, envio, recepção e tratamento das mensagens
- Tratamento básico de erros (devem ser tratados ao nível da aplicação)
- Heterogeneidade da representação dos dados

Simplifica disponibilização de serviços

- Interface bem definida, facilmente documentável e independente dos protocolos de transporte
- Sistema de registo e procura de serviços

RPCs: COMO IMPLEMENTAR



COMO ESCONDER OS DETALHES?

(SEM SUPORTE ESPECÍFICO DO RUNTIME DA LINGUAGEM)

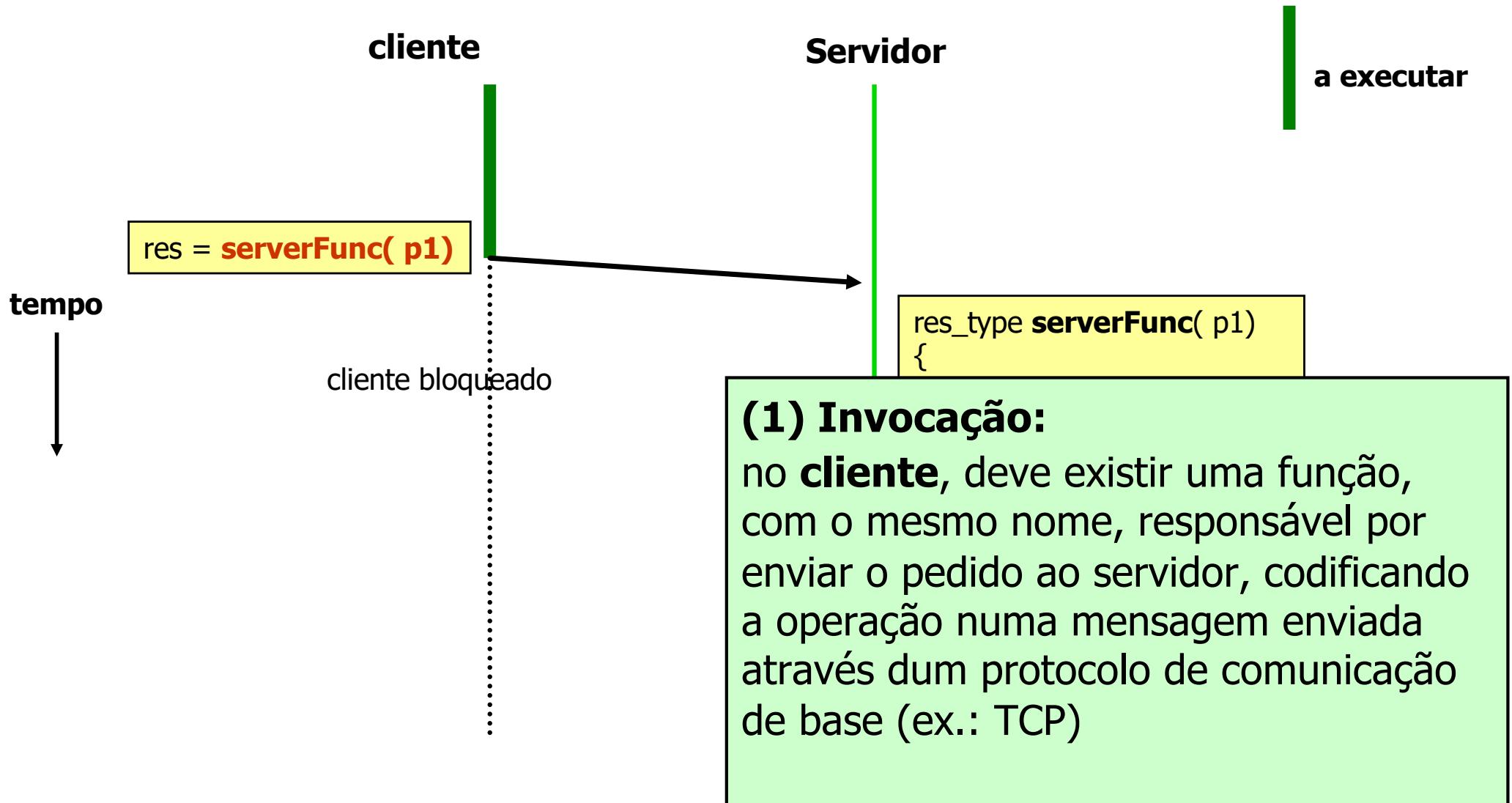
Cliente

```
res = serverFunc( p1 )
```

Servidor

```
res_type serverFunc( T1 p1 ) {  
    ...  
    return result  
}
```

RPCs: COMO IMPLEMENTAR



COMO ESCONDER OS DETALHES

(SEM SUPORTE ESPECÍFICO DO RUNTIME DA LINGUAGEM)

Cliente

```
res = serverFunc( p1 )
```

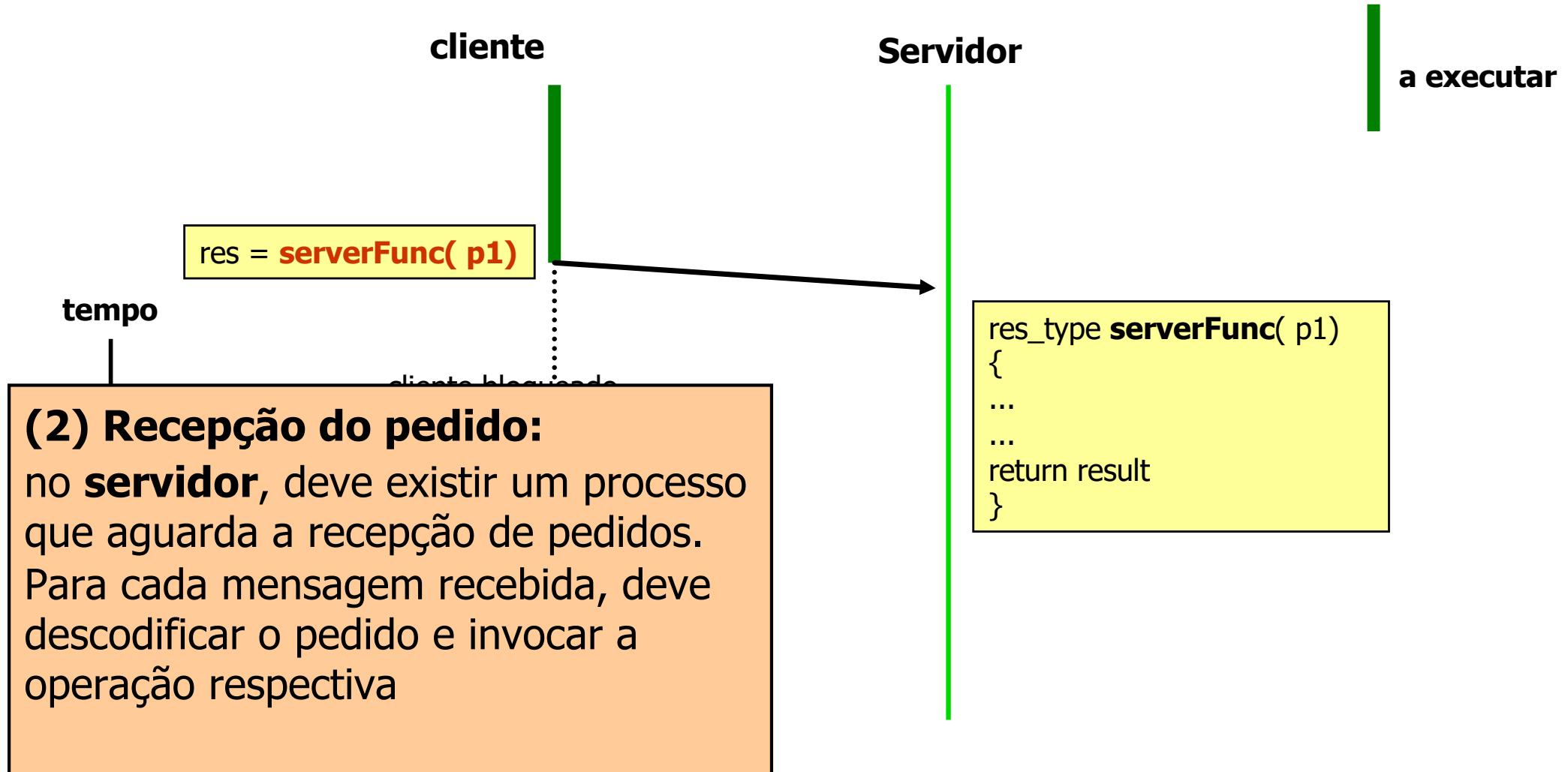
```
res_type serverFunc( T1 p1 )
s = new Socket( host, port )
s.send( msg( "serverFunc",[p1]))
```

(1) Invocação:

no **cliente**, deve existir uma função, com o mesmo nome, responsável por enviar o pedido ao servidor, codificando a operação numa mensagem enviada através dum protocolo de comunicação de base (ex.: TCP)

```
}
```

RPCs: COMO IMPLEMENTAR



COMO ESCONDER OS DETALHES?

(2) Recepção do pedido:

no **servidor**, deve existir um processo que aguarda a recepção de pedidos. Para cada mensagem recebida, deve descodificar o pedido e invocar a operação respectiva

```
res_type serverFunc( T1 p1)
    s = new Socket( host, port)
    s.send( msg( "serverFunc",[p1]))
```

LINGUAGEM)

Servidor

```
res_type serverFunc( T1 p1) {
    ...
    return result
}
```

```
s = new ServerSocket
forever
    Socket c = s.accept();
    c.receive( msg( op, params))
    if( op = "serverFunc")
        res = serverFunc( params[0]);
    else if( op = ...)
        ...
```

RPCs: COMO IMPLEMENTAR

(3) Envio da resposta:

no **servidor**, quando a execução do procedimento termina, os resultados (ou apenas a informação de fim) devem ser codificado e enviado para o cliente

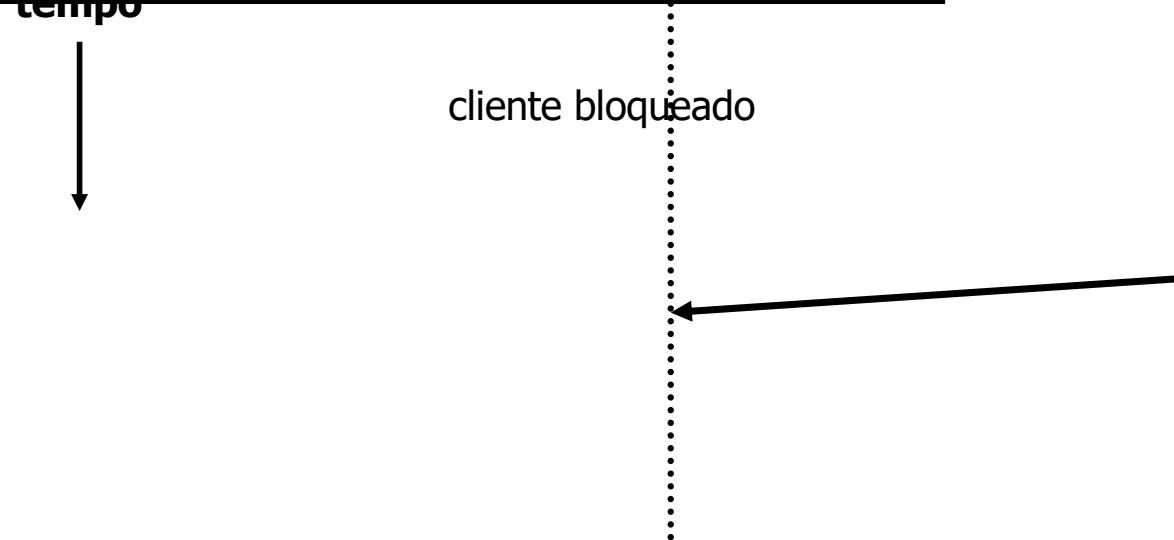
tempo

cliente bloqueado

Servidor

a executar

```
res_type serverFunc( p1 )
{
...
...
return result
}
```



COMO ESCONDER OS DETALHES?

(SEM SUPORTE ESPECÍFICO DO RUNTIME DA LINGUAGEM)

(3) Envio da resposta:

no **servidor**, quando a execução do procedimento termina, os resultados (ou apenas a informação de fim) devem ser codificado e enviado para o cliente

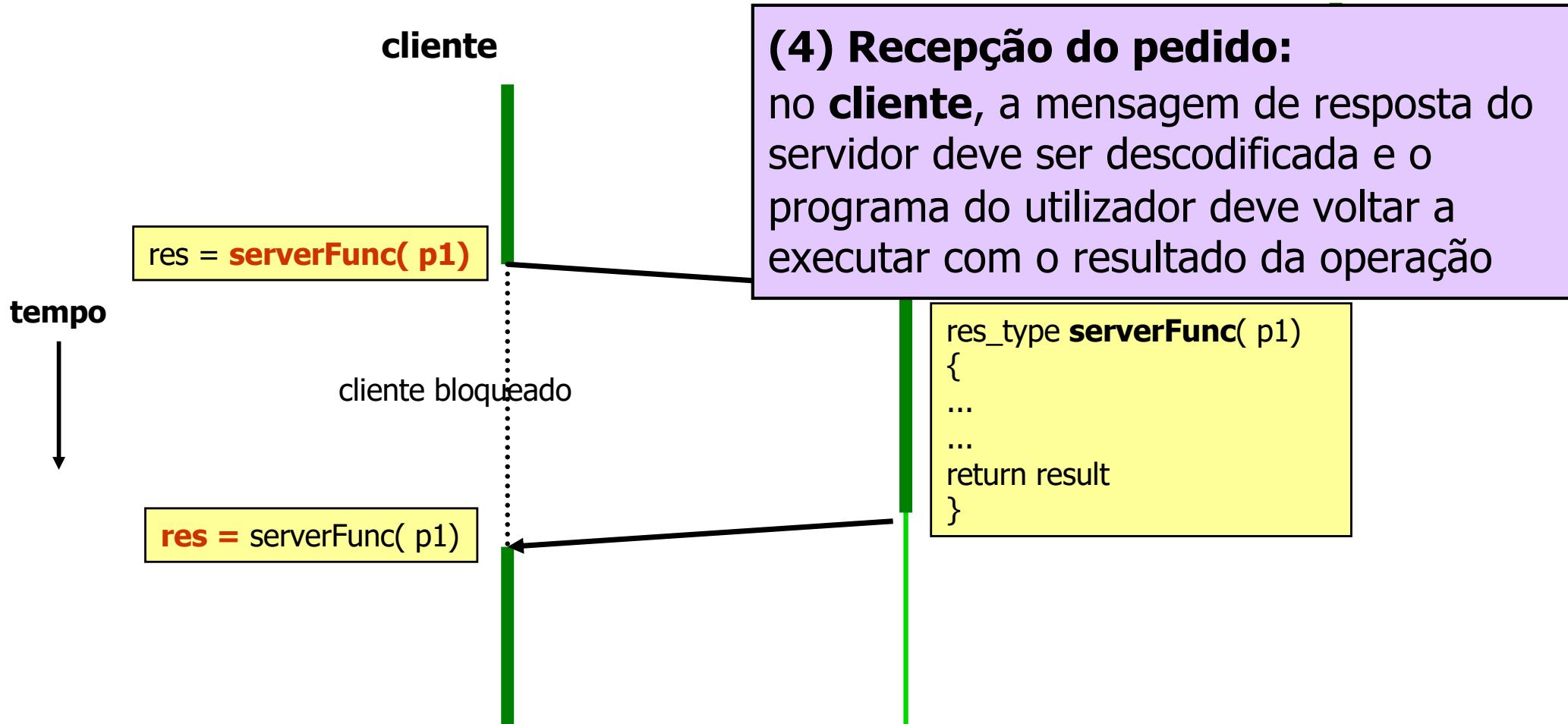
```
res_type serverFunc( T1 p1)
    s = new Socket( host, port)
    s.send( msg( "serverFunc",[p1]))
```

Servidor

```
res_type serverFunc( T1 p1) {
    ...
    return result
}
```

```
s = new ServerSocket
forever
    Socket c = s.accept();
    c.receive( msg( op, params))
    if( op = "serverFunc")
        res = serverFunc( params[0]);
    else if( op = ...)
        ...
    c.send( msg(res))
    c.close
```

RPCs: COMO IMPLEMENTAR



COMO ESCONDER OS DETALHES?

(SEM SUPORTE ESPECÍFICO DO RUNTIME DA LINGUAGEM)

Cliente

```
res = serverFunc( p1)
```

```
res_type serverFunc( T1 p1)
  s = new Socket( host, port)
  s.send( msg( "serverFunc",[p1]))
  s.receive( msg( result))
  s.close
  return result
```

(4) Recepção do pedido:
no **cliente**, a mensagem de resposta do servidor deve ser descodificada e o programa do utilizador deve voltar a executar com o resultado da operação



```
s = new ServerSocket
forever
  Socket c = s.accept();
  c.receive( msg( op, params))
  if( op = "serverFunc")
    res = serverFunc( params[0]);
  else if( op = ...)
    ...
  c.send( msg(res))
  c.close
```

COMO ESCONDER OS DETALHES?

(SEM SUPORTE ESPECÍFICO DO RUNTIME DA LISP)

Cliente

```
res = serverFunc( p1)
```

```
res_type serverFunc( T1 p1)
  s = new Socket( host, port)
  s.send( msg( "serverFunc",[p1]))
  s.receive( msg( result))
  s.close
  return result
```

Na prática, sucessivas invocações podem partilhar o mesmo socket...

Stub do cliente ou proxy do servidor

(4) Recepção do pedido:

no **cliente**, a mensagem de resposta do servidor deve ser descodificada e o programa do utilizador deve voltar a executar com o resultado da operação

(1) Invocação:

no **cliente**, deve existir uma função, com o mesmo nome, responsável por enviar o pedido ao servidor, codificando a operação numa mensagem enviada através dum protocolo de comunicação de base (ex.: TCP)

COMO ESCONDER OS DETALHES?

(SEM SUPORTE ESPECÍFICO DO RUNTIME DA LINGUAGEM)

Stub ou skeleton do servidor

(3) Envio da resposta:

no **servidor**, quando a execução do procedimento termina, os resultados (ou apenas a informação de fim) devem ser codificado e enviado para o cliente

(2) Recepção do pedido:

no **servidor**, deve existir um processo que aguarda a recepção de pedidos. Para cada mensagem recebida, deve descodificar o pedido e invocar a operação respectiva

Servidor

```
res_type serverFunc( T1 p1 ) {  
    ...  
    return result  
}
```

s = **new** ServerSocket
forever

```
    Socket c = s.accept();  
    c.receive( msg( op, params ))  
    if( op = "serverFunc")  
        res = serverFunc( params[0]);  
    else if( op = ...)  
        ...  
    c.send( msg(res))  
    c.close
```

RPCs – AUTOMATIZAÇÃO (PROXY/STUB COMPILERS)

Nos sistemas de RPC/RMI, o código de comunicação é transparente para a aplicação.

É costume designar-se de **stub do cliente** às funções do cliente que efetuam a comunicação com o servidor para executar o método no servidor;

Do lado do servidor, o **stub ou *skeleton* do servidor** corresponde ao código de comunicação para esperar as invocações e executá-las, devolvendo o resultado;

RPCs – AUTOMATIZAÇÃO (PROXY/STUB COMPILERS)

Em alguns sistemas e ambientes usam-se ferramentas (compiladores) para gerar os stubs; e.g., wsimport para WebServices SOAP.

Noutros sistemas a geração é automática: no servidor, quando este é instanciado; no cliente quando este se liga ao servidor da primeira vez:

- e.g., Java RMI, Servidor JAX-RS(Jersey);

Há ainda casos onde a invocação remota faz parte da própria especificação do ambiente/linguagem e é parte integrante do runtime:

- e.g., .NET Remoting.

AGENDA

Invocação remota de procedimentos/objects

- Motivação
- Modelo
- Organização do servidor
- **Definição de interfaces e método de passagem de parâmetros**
- Codificação dos dados
- Mecanismos de ligação (binding)
- Protocolos de comunicação
- Sistemas de objetos distribuídos

INTERFACE DEFINITION LANGUAGES (IDL)

Problema: Necessário especificar quais as operações que estão disponíveis:

- Interface do serviço – assinatura das funções
- Tipos e constantes usados

Em alguns sistemas, os clientes e os servidores podem ser implementados em linguagens diferentes.

Os IDL são usados para definir as interfaces (não o código das operações):

- Por vezes, esta distinção é difícil de fazer porque os IDLs estão integrados com linguagem
- Em certos sistemas (e.g. .NET remoting), a interface pode não ser definida autonomamente

IDLs – APROXIMAÇÕES POSSÍVEIS

Usar subconjunto de uma linguagem já existente

- Ex.: Java RMI

Definir linguagem específica para especificar interfaces dos servidores/objectos remotos

- Ex.: WSDL
- Geralmente baseado numa linguagem existente
- Necessidade de mapear o IDL e as linguagens de desenvolvimento dos clientes/servidores

INTERFACE REMOTA EM JAVA RMI

```
public interface ContaBancaria {
```

extends Remote

Interfaces remotos
estendem **Remote**

```
    public void depositar ( float quantia )  
        throws RemoteException;
```

```
    public void levantar ( float quantia )  
        throws SaldoDescoberto, RemoteException;
```

```
    public float saldoActual ( )  
        throws RemoteException;
```

```
}
```

Interfaces definidos em Java
standard

Métodos devem lançar
RemoteException para tratar
erros de comunicação

INTERFACE DEFINIDA EM C# PARA .NET REMOTING

```
using System;
namespace IRemoting
{
    public interface ContaBancaria
    {
        double SaldoActual
        {
            get;
        }
        void depositar ( float quantia );
        void levantar (float quantia);
    }
}
```

Permite definir atributos acessíveis por operações associadas (get/set)

Interface definida em C# comum

INTERFACE DEFINIDA EM C# PARA .NET REMOTING

```
using System;
namespace IRemoting
{
    public interface ContaBancaria
    {
        double SaldoActual { get; }
        void depositar ( float quantia );
        void levantar (float quantia );
    }
}
```

```
public class ServiceClass :
    System.MarshalByRefObject
{
    public void depositar(float quantia) {
        Console.WriteLine (quantia);
    }
    ...
}
```

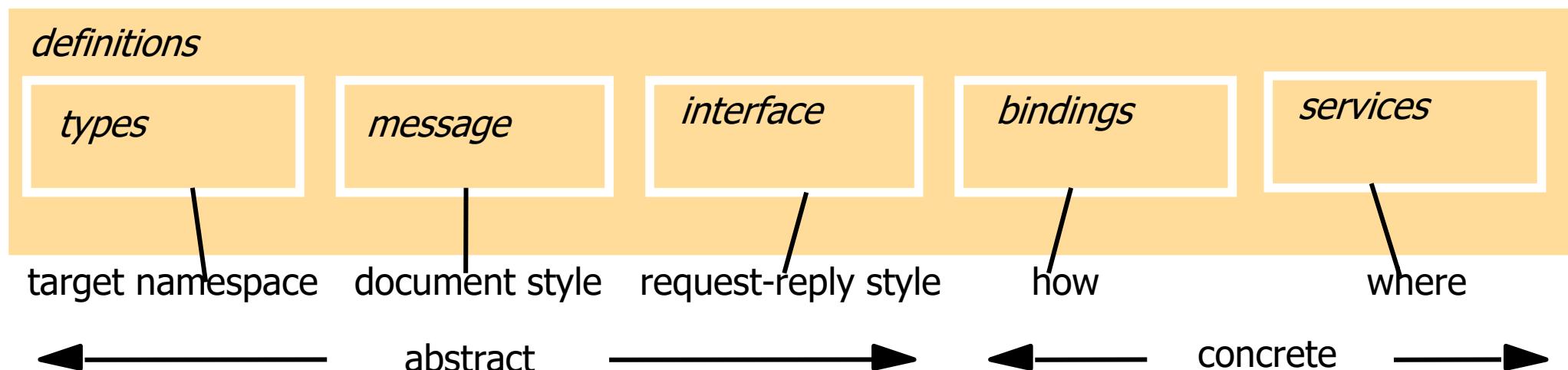
No .NET Remoting não é necessário definir qual a interface remota – esta pode ser inferida a partir da definição do servidor

Um objecto remoto deve estender MarshalByRefObject

WSDL – IDL PARA WEB SERVICES

Definição da interface em XML

- WSDL permite definir a interface do serviço, indicando quais as mensagens trocadas na interacção
- WSDL permite também definir a forma de representação dos dados e a forma de aceder ao serviço
- Especificação WSDL bastante verbosa – normalmente criada a partir de interface ou código do servidor
 - Ex. JAX-WS tem ferramentas para criar especificação a partir de



WSDL - EXEMPLO

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="HelloService"
    targetNamespace="http://www.ecerami.com/wsdl/HelloService.wsdl"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://www.ecerami.com/wsdl/HelloService.wsdl"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">

    <message name="SayHelloRequest">
        <part name="firstName" type="xsd:string"/>
    </message>
    <message name="SayHelloResponse">
        <part name="greeting" type="xsd:string"/>
    </message>

    <portType name="Hello_PortType">
        <operation name="sayHello">
            <input message="tns:SayHelloRequest"/>
            <output message="tns:SayHelloResponse"/>
        </operation>
    </portType>
```

(exemplo do livro Web Services Essentials, O'Reilly, 2002.)

WSDL - EXEMPLO

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="HelloService"
    targetNamespace="http://www.ecerami.com/wsdl/HelloService.wsdl"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://www.ecerami.com/wsdl/HelloService.wsdl"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">

    <message name="SayHelloRequest">
        <part name="firstName" type="xsd:string"/>
    </message>
    <message name="SayHelloResponse">
        <part name="greeting" type="xsd:string"/>
    </message>

    <portType name="Hello_PortType">
        <operation name="sayHello">
            <input message="tns:SayHelloRequest"/>
            <output message="tns:SayHelloResponse"/>
        </operation>
    </portType>
```

<definitions>: The HelloService

<message>:

- 1) sayHelloRequest: firstName parameter
- 2) sayHelloResponse: greeting return value

<portType>: sayHello operation that consists of a request/response service

<binding>: Direction to use the SOAP HTTP transport protocol.

<service>: Service available at: http://localhost:8080/soap/servlet/rpcrouter

WSDL - EXEMPLO

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="HelloService"
    targetNamespace="http://www.ecerami.com/wsdl/HelloService.wsdl"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://www.ecerami.com/wsdl/HelloService.wsdl"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">

    <message name="SayHelloRequest">
        <part name="firstName" type="xsd:string"/>
    </message>
    <message name="SayHelloResponse">
        <part name="greeting" type="xsd:string"/>
    </message>

    <portType name="Hello_PortType">
        <operation name="sayHello">
            <input message="tns:SayHelloRequest"/>
            <output message="tns:SayHelloResponse"/>
        </operation>
    </portType>
```

<definitions>: The HelloService

<message>:

- 1) sayHelloRequest: firstName parameter
- 2) sayHelloResponse: greeting return value

<portType>: sayHello operation that consists of a request/response service

<binding>: Direction to use the SOAP HTTP transport protocol.

<service>: Service available at: http://localhost:8080/soap/servlet/rpcrouter

WSDL - EXEMPLO

```
<binding name="Hello_Binding" type="tns:Hello_PortType">
<soap:binding style="rpc"
  transport="http://schemas.xmlsoap.org/soap/http"/>
<operation name="sayHello">
  <soap:operation soapAction="sayHello"/>
  <input>
    <soap:body
      encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
      namespace="urn:examples:helloservice"
      use="encoded"/>
  </input>
  <output>
    <soap:body
      encodingStyle="http://sc
      namespace="urn:examples:
      use="encoded"/>
  </output>
</operation>
</binding>

<service name="Hello_Service">
  <documentation>WSDL File for Hell
  <port binding="tns:Hello_Binding"
    <soap:address
      location="http://localhost:
    </port>
  </service>
</definitions>
```

<definitions>: The HelloService

<message>:

- 1) sayHelloRequest: firstName parameter
- 2) sayHelloResponse: greeting return value

<portType>: sayHello operation that consists of a request/response service

<binding>: Direction to use the SOAP HTTP transport protocol.

<service>: Service available at: http://localhost:8080/soap
/servlet/rpcrouter

WSDL - EXEMPLO

```
<binding name="Hello_Binding" type="rpc">
<soap:binding style="rpc"
    transport="http://schemas.xmlsoap.org/soap/http"/>
<operation name="sayHello">
    <soap:operation soapAction="http://urn:examples:sayHello"/>
    <input>
        <soap:body
            encodingStyle="http://schemas.xmlsoap.org/soap/encoding"
            namespace="urn:examples"
            use="encoded"/>
    </input>
    <output>
        <soap:body
            encodingStyle="http://schemas.xmlsoap.org/soap/encoding"
            namespace="urn:examples"
            use="encoded"/>
    </output>
    </operation>
</binding>
```

```
<service name="Hello_Service">
    <documentation>WSDL File for HelloService</documentation>
    <port binding="tns:Hello_Binding" name="Hello_Port">
        <soap:address
            location="http://localhost:8080/soap/servlet/rpcrouter"/>
    </port>
</service>
</definitions>
```

<definitions>: The HelloService

<message>:

- 1) sayHelloRequest: firstName parameter
- 2) sayHelloResponse: greeting return value

<portType>: sayHello operation that consists of a request/response service

<binding>: Direction to use the SOAP HTTP transport protocol.

<service>: Service available at: http://localhost:8080/soap/servlet/rpcrouter

WSDL A PARTIR DO JAVA (JAX-WS)

```
@WebService()  
public class SimpleWSServer {  
    ...  
    public SimpleWSServer() {  
        ...  
    }  
    @WebMethod()  
    public String[] list( String path) {  
        ...  
    }  
}
```

INTERFACE SERVIDOR REST EM JAVA (JAX-RS)

```
@Path("/files")
public interface FileServerREST {
    @GET
    @Path("/{path}")
    @Produces(MediaType.APPLICATION_JSON)
    public String[] list( @PathParam("path") String path);

    @POST
    @Path("/{path}")
    @Consumes(MediaType.OCTET_STREAM)
    @Produces(MediaType.APPLICATION_JSON)
    public Response upload (@PathParam("path") String path, byte[] contents);
}
```

SISTEMAS DISTRIBUÍDOS

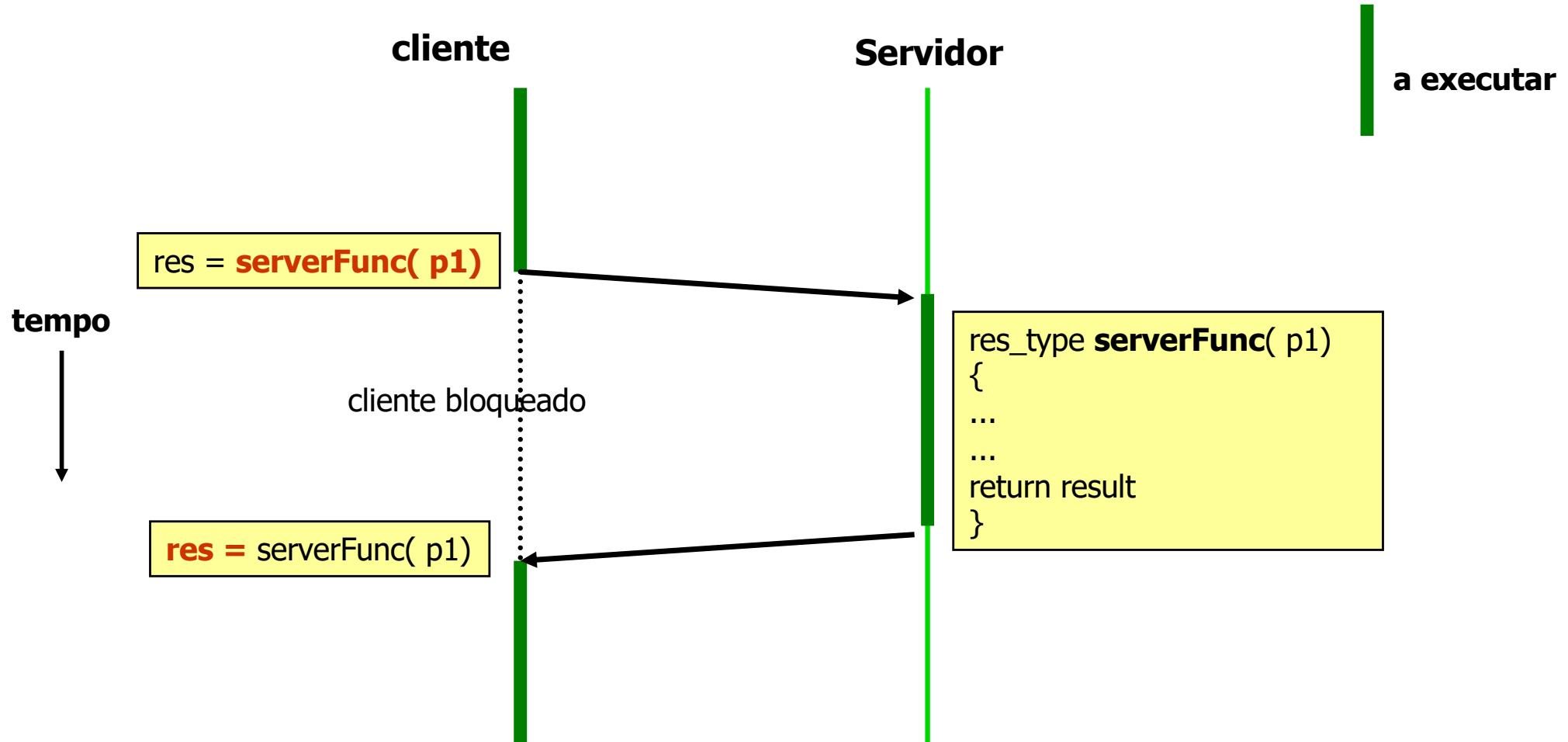
Capítulo 4 Invocação remota

NA ÚLTIMA AULA

Invocação remota de procedimentos/objects

- Motivação
- Modelo
- Concorrência no servidor
- Definição de interfaces e método de passagem de parâmetros
- Codificação dos dados
- Mecanismos de ligação (binding)
- Protocolos de comunicação
- Sistemas de objetos distribuídos

INVOCAÇÃO REMOTA



NA ÚLTIMA AULA

Cliente

```
res = serverFunc( p1)
```

```
res_type serverFunc( T1 p1)
  s = new Socket( host, port)
  s.send( msg( "serverFunc",[p1]))
  s.receive( msg( result))
  s.close
  return result
```

Stub do cliente ou proxy do servidor

(4) Recepção do pedido:

no **cliente**, a mensagem de resposta do servidor deve ser descodificada e o programa do utilizador deve voltar a executar com o resultado da operação

(1) Invocação:

no **cliente**, deve existir uma função, com o mesmo nome, responsável por enviar o pedido ao servidor, codificando a operação numa mensagem enviada através dum protocolo de comunicação de base (ex.: TCP)

NA ÚLTIMA AULA

Stub ou skeleton do servidor

(3) Envio da resposta:

no **servidor**, quando a execução do procedimento termina, os resultados (ou apenas a informação de fim) devem ser codificado e enviado para o cliente

(2) Recepção do pedido:

no **servidor**, deve existir um processo que aguarda a recepção de pedidos. Para cada mensagem recebida, deve descodificar o pedido e invocar a operação respectiva

Servidor

```
res_type serverFunc( T1 p1 ) {  
    ...  
    return result  
}
```

```
s = new ServerSocket  
forever
```

```
    Socket c = s.accept();  
    c.receive( msg( op, params ))  
    if( op = "serverFunc")  
        res = serverFunc( params[0]);  
    else if( op = ...)  
        ...  
    c.send( msg(res))  
    c.close
```

IDLs – APROXIMAÇÕES POSSÍVEIS

Os IDL são usados para definir as interfaces (não o código das operações). Aproximações possíveis:

Usar subconjunto de uma linguagem já existente

- Ex.: Java RMI

Definir linguagem específica para especificar interfaces dos servidores/objectos remotos

- Ex.: WSDL
- Geralmente baseado numa linguagem existente
- Necessidade de mapear o IDL e as linguagens de desenvolvimento dos clientes/servidores

AGENDA

Invocação remota de procedimentos/objects

- Motivação
- Modelo
- Organização do servidor
- Definição de interfaces e método de passagem de parâmetros
- Codificação dos dados
- Mecanismos de ligação (binding)
- Protocolos de comunicação
- Sistemas de objetos distribuídos

ORGANIZAÇÃO DOS SERVIDORES

Ativação dos servidores

- Servidor a executar continuamente
- Servidor ativado quando necessário

Organização interna

- Iterativo vs. concorrente

ORGANIZAÇÃO DOS SERVIDORES: ATIVAÇÃO

Existem duas formas para lidar com os pedidos dos clientes:

- Existe apenas uma instância do código do servidor para atender todos os clientes
 - Aproximação mais comum
- Cria-se uma instância do código do servidor para atender cada cliente
 - E.g. .NET remoting: servidor *SingleCall*
 - REST em Java: cada pedido é tratado por um objeto criado no momento

ORGANIZAÇÃO DOS SERVIDORES: JAVA REST

No suporte REST do Java, quando se regista uma classe, são criadas múltiplas instâncias para tratar os vários pedidos.

Pode-se indicar que se pretende apenas uma instância com a anotação `@Singleton`.

```
20
21 @Singleton
22 public class MessageResource implements MessageService {
23
24     private Random randomNumberGenerator;
25 }
```

Alternativamente, pode-se registar um objeto do recurso em vez dum a class.

```
30     URI serverURI = URI.create(String.format("http://%s:%s/rest", ip, PORT));
31
32     ResourceConfig config = new ResourceConfig();
33     config.register(new UserResource(domain, serverURI));
34
35     JdkHttpServerFactory.createHttpServer(serverURI, config);
36 }
```

VANTAGENS / DESVANTAGENS

Uma instância por pedido

- Não existem problemas de concorrência devido a múltiplos pedidos
- Não é possível manter estado na instância do servidor
 - Em geral, o estado duma aplicação é guardado numa base de dados

Uma instância apenas

- Necessário lidar com concorrência devido a múltiplos pedidos
- É possível manter estado na instância do servidor

ATIVAÇÃO DE OBJETOS REMOTOS (E.G. JAVA RMI)

Motivação: num sistema pode haver um número muito elevado de objetos remotos cujo estado se quer que persista durante tempo ilimitado, mas que não estão em uso durante grande parte do tempo

Solução: ativam-se os objetos remotos apenas quando necessário

- Quando um método é invocado ou quando uma referência remota é obtida

Activator: servidor responsável por:

- Manter informação sobre os objetos ativáveis
- Ativar os objetos remotos quando solicitado por um cliente
- Manter informação sobre localização dos objetos ativados

Objeto remoto *passivo* (quando não ativado)

- Código
- Estado do objeto *marshalled*

Referência remota mantém informação necessária para solicitar a ativação do objecto

ORGANIZAÇÃO DOS SERVIDORES

Ativação dos servidores

- Servidor a executar continuamente
- Servidor ativado quando necessário

Organização interna

- **Iterativo vs. concorrente**

ORGANIZAÇÃO DOS SERVIDORES: *THREADS*

Servidor iterativo: o servidor executa os pedidos de forma sequencial, executando um de cada vez

- Modelo simples

Para alguns tipos de serviços, esta aproximação pode ter um desempenho inadequado

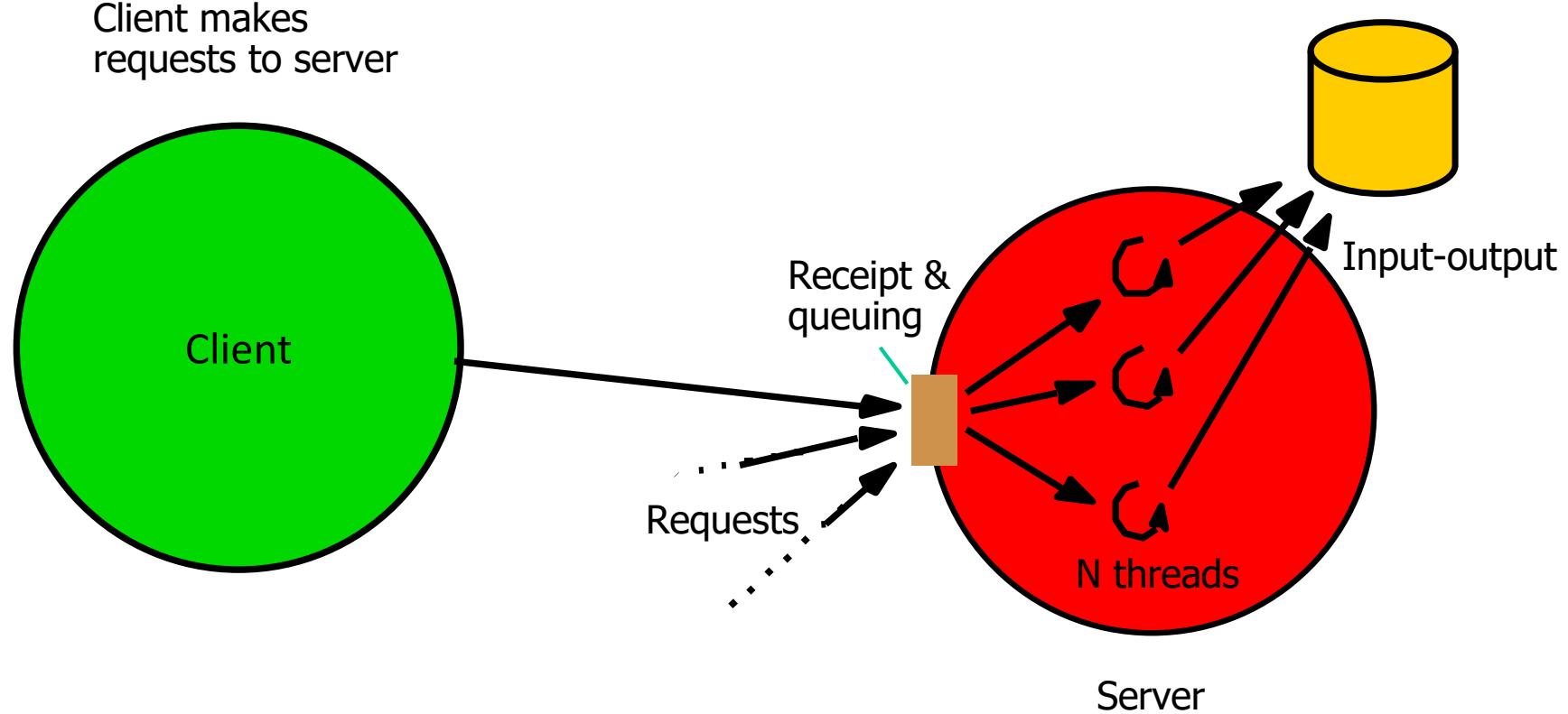
- Exemplos: servidores de bases de dados, de ficheiros, etc. Porquê?
- Exemplo: serviços que chamam outros serviços em ambos os sentidos (A->B e B->A). Porquê?

Em geral, quando a execução de uma operação remota pode ser longa é interessante introduzir concorrência no servidor.

Porquê?

Permite aproveitar os recursos computacionais da máquina.

UTILIZAÇÃO DE THREADS NUM SERVIDOR



A ter em atenção:

Possíveis problemas de concorrência: necessidade de sincronizar execução dos vários *threads*.

Como é que os threads se organizam e se relacionam com os pedidos?

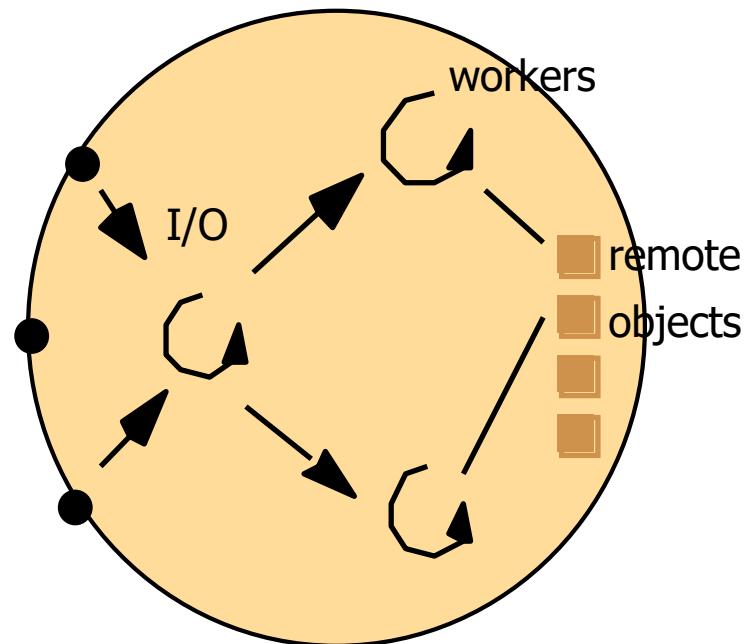
THREAD POR PEDIDO

Cada pedido é atendido por um *thread*.

Pode-se criar um *thread* quando chega um pedido ou existir um conjunto de *threads* que podem ser usadas para atender os pedidos.

Podem existir múltiplos *threads* a executar no mesmo servidor/objeto.

- Necessário executar controlo de concorrência no acesso aos dados.



a. Thread-per-request

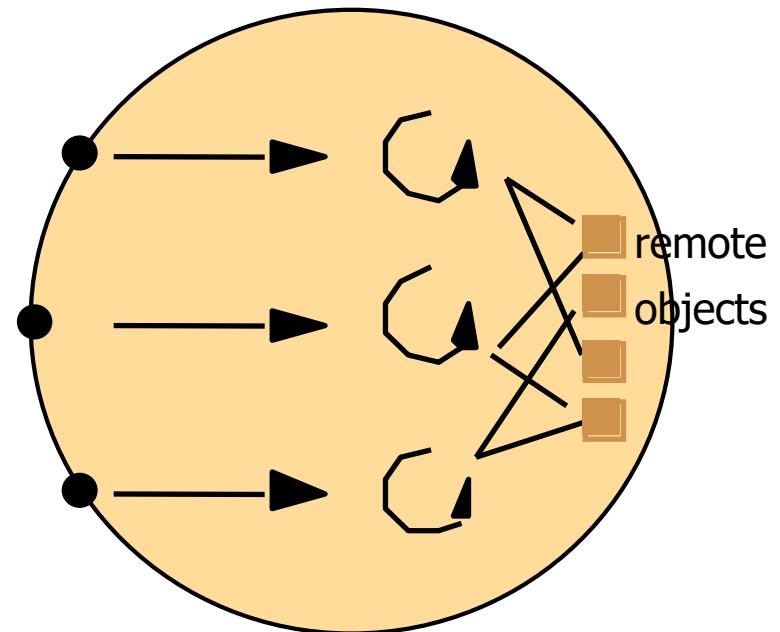
THREAD POR CONEXÃO

Cada conexão é atendida por um *thread*.

Pode-se criar um *thread* quando se cria uma conexão ou existir um conjunto de *threads* que podem ser usadas para atender os pedidos.

Podem existir múltiplos *threads* a executar no mesmo servidor/objeto.

- Necessário executar controlo de concorrência no acesso aos dados.



THREAD POR OBJETO

Os pedidos de um objeto são atendidos todos pelo mesmo *thread*, de forma sequencial.

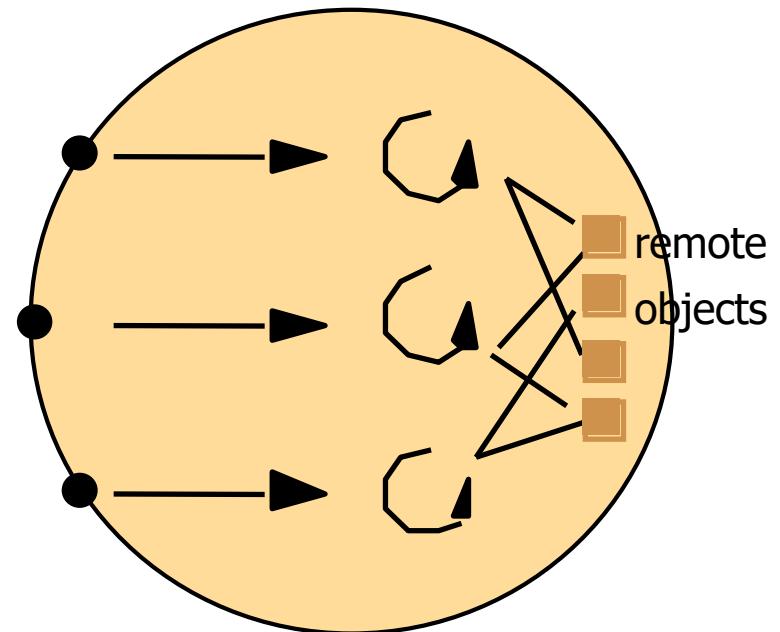
Cada objeto tem um *thread* associada.

Não existem problemas de concorrência no acesso ao estado dum servidor/objeto.

- Podem existir problemas se um servidor puder aceder a outros objetos.

Pode levar a problemas de deadlocks se comunicação com outros servidores for síncrona.

- Modelo de atores e CSP disponível em linguagens como o Erlang e o Go.



b. Thread-per-connection

ORGANIZAÇÃO DOS *THREADS*

Nos sistemas que usam múltiplos *threads* é comum:

- Existir um *thread* responsável por distribuir as invocações e existir um conjunto de *threads* responsáveis por executar as invocações, sendo reutilizados em sucessivas invocações
- *Pools de threads*
 - Em cada momento, o sistema mantém informação sobre os *threads* que não estão a processar nenhuma operação, os quais se encontram a *dormir*
 - Quando uma nova invocação é recebida, a informação sobre a mesma é passada para um *thread* da *pool*, o qual fica responsável por processar o pedido
 - No fim de processar o pedido, o *thread* volta à *pool*
- Esta aproximação permite dimensionar o número de *threads* à capacidade da máquina em que o servidor corre.

CONTROLO DE CONCORRÊNCIA NOS SERVIDORES: ACESSO A ESTADO INTERNO

Quando existem múltiplos threads a executar concorrentemente e a aceder aos mesmos recursos é necessário controlar estes acessos

- Porquê?

Porque durante a execução duma operação no servidor, o estado das variáveis pode ser alterado por outro *thread*

ACESSO CONCORRENTE A ESTRUTURAS DE DADOS: EXEMPLO

```
@Singleton
public class UsersResource implements RestUsers {
    private final Map<String,User> users = new HashMap<String, User>();
    @Override
    public String createUser(User user) {
        if( users.containsKey(user.getUserId()))
            throw new WebApplicationException( Status.CONFLICT );
        users.put(user.getUserId(), user);
        return user.getUserId();
    }
    @Override
    public List<User> searchUsers(String pattern) {
        List<User> result = new ArrayList<User>();
        users.values().stream().forEach( u -> { if( u.getFullName().
            indexOf(pattern) != -1) result.add( new User(u)); });
        return result;
    }
}
```

ACESSO CONCORRENTE A ESTRUTURA DE DADOS

EXEMPLO

```
@Singleton
```

```
public class UsersResource implements
```

```
    private final Map<String,User> users = new HashMap<String, User>();
```

```
@Override
```

```
public String createUser(User user) {
```

```
    if( users.containsKey(user.getUserId()))
```

```
        throw new WebApplicationException( Status.CONFLICT );
```

```
    users.put(user.getUserId(), user);
```

```
    return user.getUserId();
```

```
}
```

```
@Override
```

```
public List<U
```

```
    List<User> users =
```

```
    return
```

```
}
```

```
java.util.ConcurrentModificationException
    at java.base/java.util.HashMap$ValueSpliterator.forEachRemaining(HashMap.java:1429)
    at java.base/java.util.stream.ReferencePipeline$Head.forEach(ReferencePipeline.java:547)
    at sd2021.aula2.server.resources.UsersResource.searchUsers(UsersResource.java:27)
    at jdk.internal.reflect.GeneratedMethodAccessor5.invoke(Unknown Source)
    at java.base/jdk.internal.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
    at java.base/java.lang.reflect.Method.invoke(Method.java:564)
    at org.glassfish.jersey.server.model.internal.ResourceMethodInvocationHandler$1.run(ResourceMethodInvocationHandler.java:154)
    at org.glassfish.jersey.server.model.internal.AbstractJavaResourceMethodDispatcher.dispatch(AbstractJavaResourceMethodDispatcher.java:87)
    at org.glassfish.jersey.server.model.internal.AbstractJavaResourceMethodDispatcher.dispatch(AbstractJavaResourceMethodDispatcher.java:87)
    at org.glassfish.jersey.server.model.internal.JavaResourceMethodDispatcherProvider$TypeMethodInvoker.invoke(JavaResourceMethodDispatcherProvider.java:108)
    at org.glassfish.jersey.server.model.internal.AbstractJavaResourceMethodDispatcher.dispatch(AbstractJavaResourceMethodDispatcher.java:87)
    at org.glassfish.jersey.server.model.ResourceMethodInvoker.invoke(ResourceMethodInvoker.java:102)
    at org.glassfish.jersey.server.model.ResourceMethodInvoker.apply(ResourceMethodInvoker.java:92)
    at org.glassfish.jersey.server.model.ResourceMethodInvoker.apply(ResourceMethodInvoker.java:85)
    at org.glassfish.jersey.server.ServerRuntime$1.run(ServerRuntime.java:255)
```

Execução concorrente do método **createUser** e **searchUsers** pode levar a exceções, por acesso concorrente ao mapa values.

ACESSO CONCORRENTE A ESTRUTURAS DE DADOS: EXEMPLO

```
@Singleton
public class UsersResource implements RestUsers {
    private final Map<String,User> users = new HashMap<String, User>();
    @Override
    public String createUser(User user) {
        if( users.containsKey(user.getUserId()))
            throw new WebApplicationException( Status.CONFLICT );
        users.put(user.getUserId(), user);
        return user.getUserId();
    }
    @Override
    public List<User> getAllUsers() {
        List<User> result = new ArrayList<User>(users.values());
        return result;
    }
}
```

Execução concorrente do método **createUser** por dois threads, com users diferentes mas o mesmo userId pode levar a que ambos os threads verifiquem que o userId não existe e pensem que inseriram o utilizador, quando apenas o último ficará guardado.

TÉCNICAS DE CONTROLO DE CONCORRÊNCIA

Monitores (e.g. métodos/blocos synchronized no Java)

Locks

Estruturas de dados concorrentes (e.g. java.util.concurrent)

Transações

MÉTODOS SYNCHRONIZED

```
@Singleton
public class UsersResource implements RestUsers {
    private final Map<String,User> users = new HashMap<String, User>();
    @Override
    public synchronized String createUser(User user) {
        if( users.containsKey(user.getUserId()))
            throw new WebApplicationException( Status.CONFLICT );
        users.put(user.getUserId(), user);
        return user.getUserId();
    }
    @Override
    public synchronized List<User> searchUsers(String pattern) {
        List<User> result = new ArrayList<User>();
        users.values().stream().forEach( u -> { if( u.getFullName().
            indexOf(pattern) != -1) result.add( new User(u)); });
        return result;
    }
}
```

MÉTODOS SYNCHRONIZED

```
@Singleton  
public class UsersResource implements  
    private final Map<String,User> users = new HashMap<String,User>();  
  
@Override  
public synchronized String createUser(User user) {  
    if( users.containsKey(user.getUserId()) )  
        throw new WebApplicationException( Status.CONFLICT );  
    users.put(user.getUserId(), user);  
    return user.getUserId();  
}  
  
@Override  
public synchronized List<User> searchUsers(String pattern) {  
    List<User> result = new ArrayList<User>();  
    users.values().stream()  
        .filter( user -> user.getName().toLowerCase().indexOf(pattern) >= 0 )  
        .forEach( user -> result.add(user) );  
    return result;  
}
```

Num dado momento, apenas um *thread* pode executar nos métodos marcados como synchronized. Se todos os métodos forem synchronized, apenas um thread modifica o servidor em cada momento, resolvendo os problemas anteriores.

Potenciais problemas:

- Se um método for lento, limita o desempenho do servidor;
- Se um método invocar outro servidor pode, no limite, levar a problemas de deadlock no caso de pedidos cruzados entre servidores (ou ciclos de invocação).

BLOCOS SYNCHRONIZED

```
@Singleton
public class UsersResource implements RestUsers {
    private final Map<String,User> users = new HashMap<String, User>();
    @Override
    public String createUser(User user) {
        if(user.getUserId() == null || user.getPassword() == null ||
           user.getFullName() == null || user.getEmail() == null)
            throw new WebApplicationException( Status.BAD_REQUEST );
        synchronized( users ) {
            if( users.containsKey(user.getUserId()) )
                throw new WebApplicationException( Status.CONFLICT );
            users.put(user.getUserId(), user);
        }
        return user.getUserId();
    }
}
```

É possível reduzir a zona que executa em exclusão mútua usando um bloco *synchronized* apenas para proteger as operações que podem causar problema. Neste exemplo, as verificações se o objeto User está correto não precisam de ser protegidas – apenas o acesso a variáveis que podem ser acedidas concorrentemente precisa.

```
private final Map<String,User> users = new HashMap<String, User>();  
@Override  
public String createUser(User user) {  
    if(user.getUserId() == null || user.getPassword() == null ||  
        user.getFullName() == null || user.getEmail() == null)  
        throw new WebApplicationException( Status.BAD_REQUEST );  
    synchronized( users ) {  
        if( users.containsKey(user.getUserId()) )  
            throw new WebApplicationException( Status.CONFLICT );  
        users.put(user.getUserId(), user);  
    }  
    return user.getUserId();  
}
```

Em cada momento, apenas um thread pode aceder a um bloco *synchronized* relativo à variável X.

BLOCOS SYNCHRONIZED

```
public class Example {  
    private final Queue<String> strs = new LinkedList<String>();  
  
    public void addOne(String s) {  
        synchronized( strs ) {  
            strs.put( s );  
            try {  
                strs.notifyAll();  
            } catch( InterruptedException e) { }  
        }  
    }  
    public String getOne() {  
        synchronized( strs ) {  
            while (strs.peek() == null)  
                try {  
                    res = strs.wait();  
                } catch(Exception e) { }  
            return strs.poll();  
        }  
    }  
}
```

BLOCOS SYNCHRONIZED

```
public class Example {  
    private final Queue<String> strs = new LinkedList<String>();  
  
    public void addOne(String s) {  
        synchronized( strs ) {  
            strs.put( s );  
            try {  
                strs.notifyAll();  
            } catch( InterruptedException e) { }  
        }  
    }  
  
    public String getOne() {  
        synchronized( strs ) {  
            while (strs.peek() == null)  
                try {  
                    res = strs.wait();  
                } catch(Exception e) { }  
            return strs.poll();  
        }  
    }  
}
```

Os monitores permitem ainda que um *thread* se bloqueeie no meio dum bloco synchronized, permitindo a outros *threads* entrarem na região crítica e mais tarde desbloquear o *thread* bloqueado – esta funcionalidade é usada, por exemplo, quando se quer esperar que alguma condição ocorra.

TÉCNICAS DE CONTROLO DE CONCORRÊNCIA

Monitores (e.g. métodos/blocos synchronized no Java)

Locks

- Estudado em FSO

Estruturas de dados concorrentes (e.g. java.util.concurrent)

Transações

ESTRUTURAS DE DADOS CONCORRENTES

```
public class Example {  
    private final Queue<String> strs = new LinkedBlockingQueue<String>();  
  
    public void addOne(String s) {  
        strs.put( s );  
    }  
    public String getOne() {  
        return strs.take();  
    }  
}
```

O pacote **java.util.concurrent** tem implementações que permitem acesso concorrente, incluindo a iteradores. Não resolve todos os problemas de concorrência.

ESTRUTURAS DE DADOS CONCORRENTES (CONT.)

```
@Singleton
public class UsersResource implements RestUsers {
    private final Map<String,User> users = new ConcurrentHashMap<String, User>();
    @Override
    public String createUser(User user) {
        if( users.containsKey(user.getUserId()) )
            throw new WebApplicationException( Status.CONFLICT );
        users.put(user.getUserId(), user);
        return user.getUserId();
    }
    @Override
    public List<User> searchUsers(String pattern) {
        List<User> result = new ArrayList<User>();
        users.values().stream().forEach( u -> { if( u.getFullName().indexOf(pattern) != -1) result.add( new User(u)); });
        return result;
    }
}
```

JAVA.UTIL.CONCURRENT

CopyOnWriteArrayList, CopyOnWriteArraySet

Criam uma cópia da estrutura de dados sempre que a mesma é alterada

ConcurrentHashMap, ConcurrentSkipListSet

Lidam com acesso concorrentes.

Nota: acessos sucessivos não são atómicos

```
ConcurrentHashMap<String,User> map = ...
```

```
if(! map.containsKey( "SD"))
```

```
    map.put("SD",user);
```

Nada garante que dois threads concorrentemente não entram no ramo *then* do *if*, fazendo put de dois valores diferentes.

TÉCNICAS DE CONTROLO DE CONCORRÊNCIA

Monitores (e.g. métodos/blocos synchronized no Java)

Locks

- Estudado em FSO

Estruturas de dados concorrentes (e.g. java.util.concurrent)

Transações

- É comum os servidores aplicacionais serem *stateless* e todo o estado persistente estar numa base de dados.
- Controlo de concorrência pode ser delegado para a base de dados – operação da aplicação origina transação na base de dados - controlo de concorrência efetuado na base de dados.

CONTROLO DE CONCORRÊNCIA NOS SERVIDORES: DEADLOCKS

Quando se utiliza um mecanismo de controlo de concorrência baseado em locks – e.g. monitores , locks – é necessário lidar com potenciais problemas de deadlocks.

Os deadlock podem surgir dentro dum servidor ou entre servidores.

CONTROLO DE CONCORRÊNCIA NOS SERVIDORES: DEADLOCKS NUM SERVIDOR (EXEMPLO)

```
private final User user1 = ...  
private final User user2 = ...  
  
public String copy1To2 () {  
    synchronized(user1) {  
        synchronized(user2) {  
            User aux = user1;  
            user1 = user2;  
            user2 = aux;  
        }  
    }  
}  
  
public String copy2To1 () {  
    synchronized(user2) {  
        synchronized(user1) {  
            User aux = user2;  
            user2 = user1;  
            user1 = aux;  
        }  
    }  
}
```

CONTROLO DE CONCORRÊNCIA NOS SERVIDORES: DEADLOCKS

Para evitar deadlocks num servidor, podem-se usar diferentes aproximações:

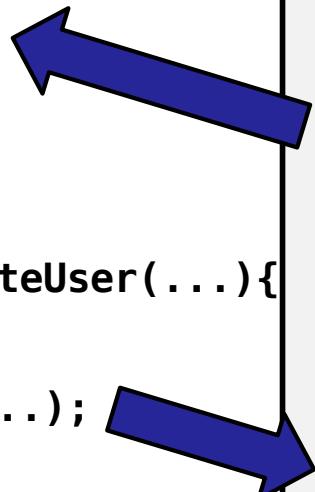
1. Ter apenas um lock;
2. Obter locks sempre por ordem – e.g. se tivermos os locks l_1, l_2, l_3, \dots , uma operação obtém os locks que precisa por ordem, i.e., após obter o lock l_i , nunca se obtém o lock l_j , tal que $j < i$.

CONTROLO DE CONCORRÊNCIA NOS SERVIDORES: DEADLOCKS DISTRIBUÍDOS

Os deadlocks podem surgir entre servidores.

```
public class UsersResource
    implements RestUsers {
    @Override
    public synchronized User getUser(...) {
        ...
    }
    @Override
    public synchronized User deleteUser(...){
        ...
        directory.deleteUserFiles(...);
    }
}
```

```
public class DirectoryResource
    implements RestDirectory {
    @Override
    public synchronized FileInfo writeFile (...){
        ...
        User u = users.getUser( ...);
    }
    @Override
    public synchronized User deleteUserFiles(...){
        ...
    }
}
```



CONTROLO DE CONCORRÊNCIA NOS SERVIDORES: DEADLOCKS DISTRIBUÍDOS

Para evitar deadlocks distribuídos, deve-se evitar fazer uma invocação remota enquanto de tem um *lock*.

SISTEMAS DISTRIBUÍDOS

Capítulo 4

Invocação de procedimentos e de métodos remotos

NA AULA DE HOJE

Invocação remota de procedimentos/objects

- Motivação
- Modelo
- Concorrência no servidor
- Definição de interfaces e método de passagem de parâmetros
- Codificação dos dados
- Mecanismos de ligação (binding)
- Protocolos de comunicação
- Sistemas de objetos distribuídos

CODIFICAÇÃO DOS DADOS - PROBLEMA

Como representar dados trocados entre os clientes e os servidores?

CODIFICAÇÃO DOS DADOS - PROBLEMA

Várias dimensões do problema

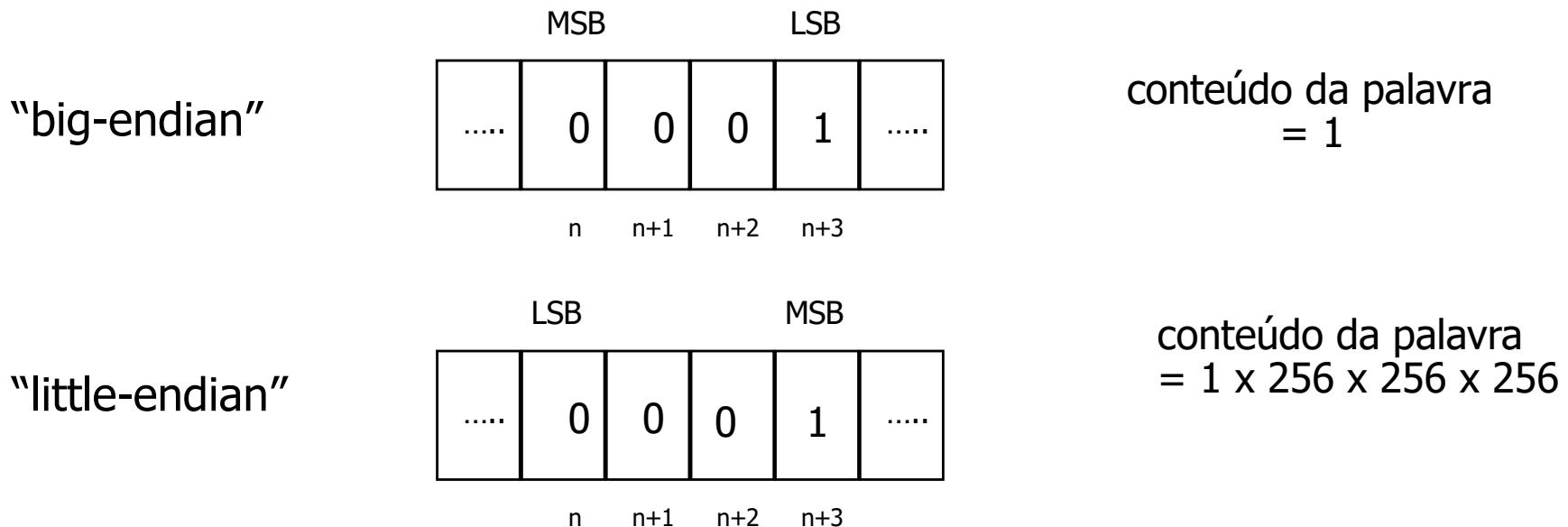
- Diferentes representações de tipos primitivos dependendo do sistema/processador
- Diferentes representações dos tipos complexos em diferentes linguagens

Dados têm se ser enviados como uma sequência/array de bytes

REPRESENTAÇÃO DOS TIPOS PRIMITIVOS

Diferentes sistemas representam os tipos primitivos de formas diferentes
Inteiros armazenados por ordem diferente em memória – big-endian vs. little endian
Diferentes representações para números reais – IEEE 754, decimal32, etc.
Caracteres com diferentes codificações – ASCII, UTF-8 ,UTF-16, etc.

Simples transmissão dos valores armazenados pode levar a resultados errados



REPRESENTAÇÕES DOS DADOS – TIPOS COMPLEXOS

Aplicações manipulam estruturas de dados complexas

- Ex.: representadas por grafos de objectos

Mensagens são sequências de bytes

O que é necessário fazer para propagar estrutura de dados complexa?

- É necessário convertê-la numa sequência de bytes
- Por exemplo, para um objecto é necessário:
 - Converter as variáveis *internas*, incluindo outros objectos
 - Necessário lidar com ciclos nas referências

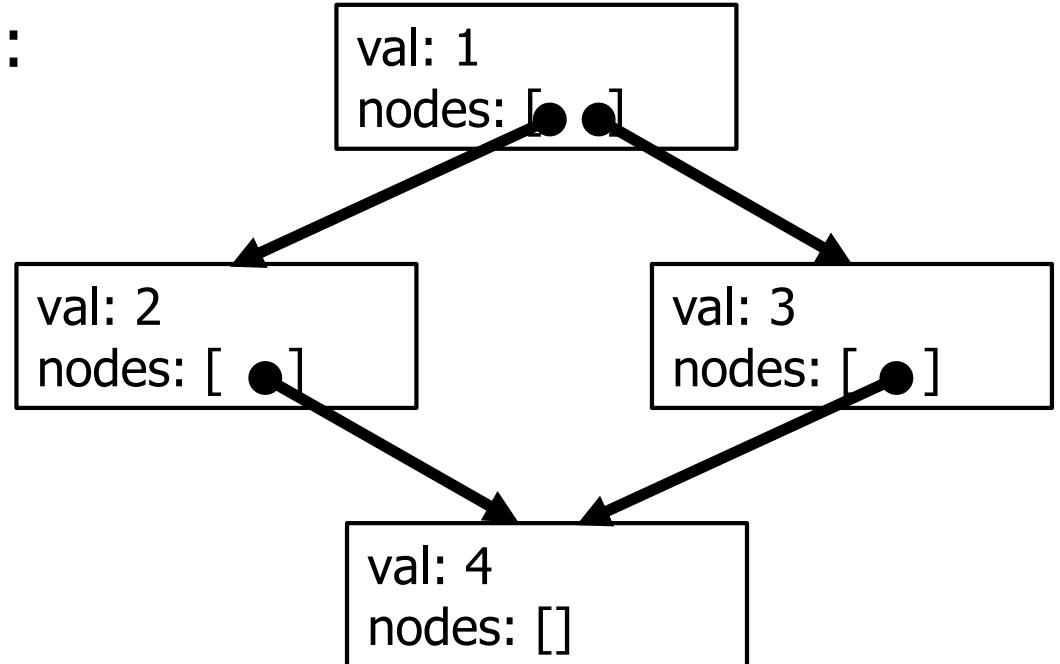
Marshalling – processo de codificar do formato interno para o formato rede

Unmarshalling – processo de descodificar do formato rede para o formato interno

PORQUE É QUE O MARSHALLING É COMPLEXO

Considere o seguinte exemplo:

```
class Node {  
    int val;  
    Node[] nodes;  
}
```

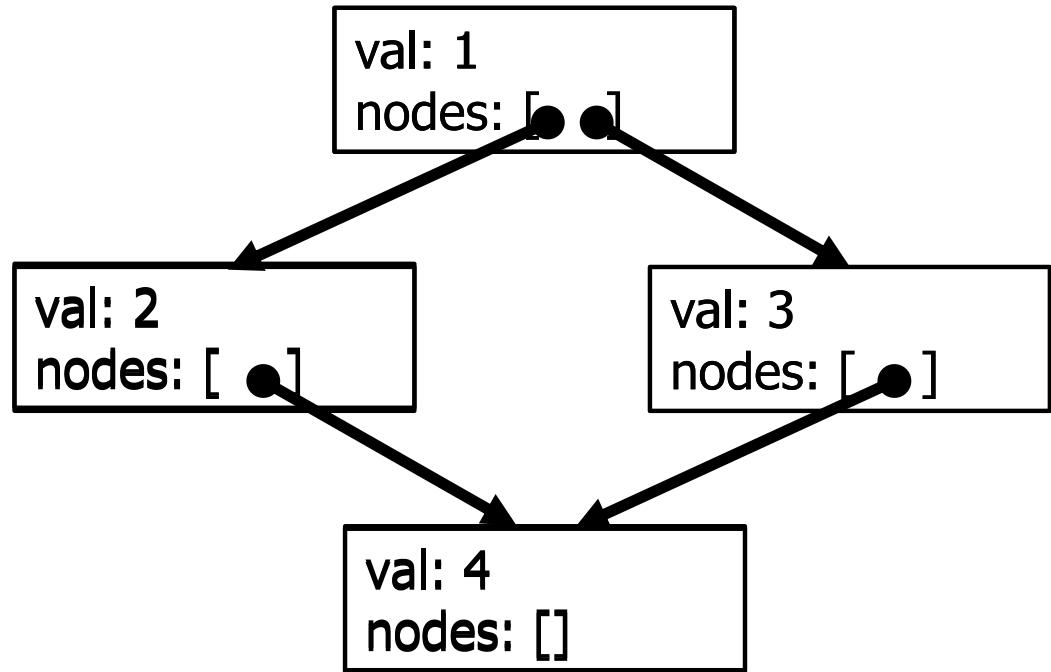


PORQUE É QUE O MARSHALLING É COMPLEXO?

Serializamos outra vez o node com val = 4 ?

Necessário ter forma de referenciar que um objeto já foi serializado.

Necessário representar as referências.
O que serializamos a seguir?



APROXIMAÇÕES À CODIFICAÇÃO DOS DADOS

Utilização de formato intermédio independente (network standard representation)

- Emissor converte da representação nativa para a representação da rede
- O receptor converte da representação da rede para a representação standard

Utilização do formato do emissor (receiver makes it right)

- Emissor envia usando a sua representação interna e indicando qual ela é
- Receptor, ao receber, faz a conversão para a sua representação

Utilização do formato do receptor (sender makes it right)

Propriedades:

- Desempenho ?
 - rep. intermédia tem pior desempenho - exige duas transformações
- Complexidade (número de transformações a definir) ?
 - rep. intermédia exige apenas que em cada plataforma se saiba converter de/para formato intermédio

JAVA SERIALIZATION

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

The true serialized form contains additional type markers; h0 and h1 are handles

```
public class Person
    implements Serializable
{
    private String name;
    private String place;
    private int year;
    ...
}
```

Assume-se que o processo de *deserialization* não tem informação sobre os objectos serializados
Forma serializada inclui informação dos tipos
Serialização grava estado de um grafo de objectos
A cada objecto é atribuído um *handle*. Permite escrever apenas uma vez cada objecto, mesmo quando existem várias referência para o mesmo no grafo de objectos.

SERIALIZAÇÃO DE OBJECTOS

Permite codificar/descodificar grafos de objectos

- Detecta e preserva ciclos pois incorpora a identidade dos objectos no grafo

Adaptável em cada classe (os métodos responsáveis podem ser redefinidos)

Os objectos devem ser serializáveis

- por omissão não são – porquê?
 - poderia abrir problemas de segurança. Exemplo?
 - Permitia acesso a campos private, por exemplo.

Os campos static e transient não são serializados

Usa *reflection* – permite obter informação sobre os tipos em runtime

- Assim, não necessita de funções especiais de marshalling e unmarshalling

EXTENSIBLE MARKUP LANGUAGE (XML)

XML permite descrever estruturas de dados complexas

Tags usadas para descrever a estrutura dos dados

Permite associar pares atributo/valor com a estrutura lógica

XML é extensível

Novas tags definidas quando necessário

Num documento XML toda a informação é textual

Podem-se codificar valores binários, por exemplo, em base64

No contexto dos sistemas de RPC/RMI, o XML pode ser usado para:

Codificar parâmetros em sistemas de RPC

Codificar invocações (SOAP)

Etc.

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

```
<?xml version="1.0"?>  
<methodCall>  
    <methodName>inc</methodName>  
>  
    <params>  
        <param>  
            <value><i4>41</i4></value>  
        </param>  
    </params>  
</methodCall>
```

EXTENSIBLE MARKUP LANGUAGE (XML)

XML permite descrever estruturas de dados complexas

Tags usadas para descrever a estrutura dos dados

```
<person id="123456789">  
  <name>Smith</name>  
  <place>London</place>  
  <age>1021</age>
```

```
<?xml version='1.0' encoding='UTF-8'?>  
<soap:Envelope xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'  
  xmlns:xsd='http://www.w3.org/2001/XMLSchema'  
  xmlns:soap='http://schemas.xmlsoap.org/soap/envelope/'  
  xmlns:soapenc='http://schemas.xmlsoap.org/soap/encoding/'  
  soap:encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'>  
<soap:Body>  
  <n:sayHello xmlns:n='urn:examples:helloservice'>  
    <firstName xsi:type='xsd:string'>World</firstName>  
  </n:sayHello>  
</soap:Body>  
</soap:Envelope>
```

Etc.

```
  <value><i4>41</i4></value>  
  </param>  
</params>  
</methodCall>
```

XML SCHEMA / XML NAMESPACES

Um XML namespace permite criar espaço de nomes para os nomes dos elementos e atributos usados nos documentos XML

Um XML schema define os elementos e atributos que podem aparecer num documento XML

```
<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>
```

XML SCHEMA / XML NAMESPACES

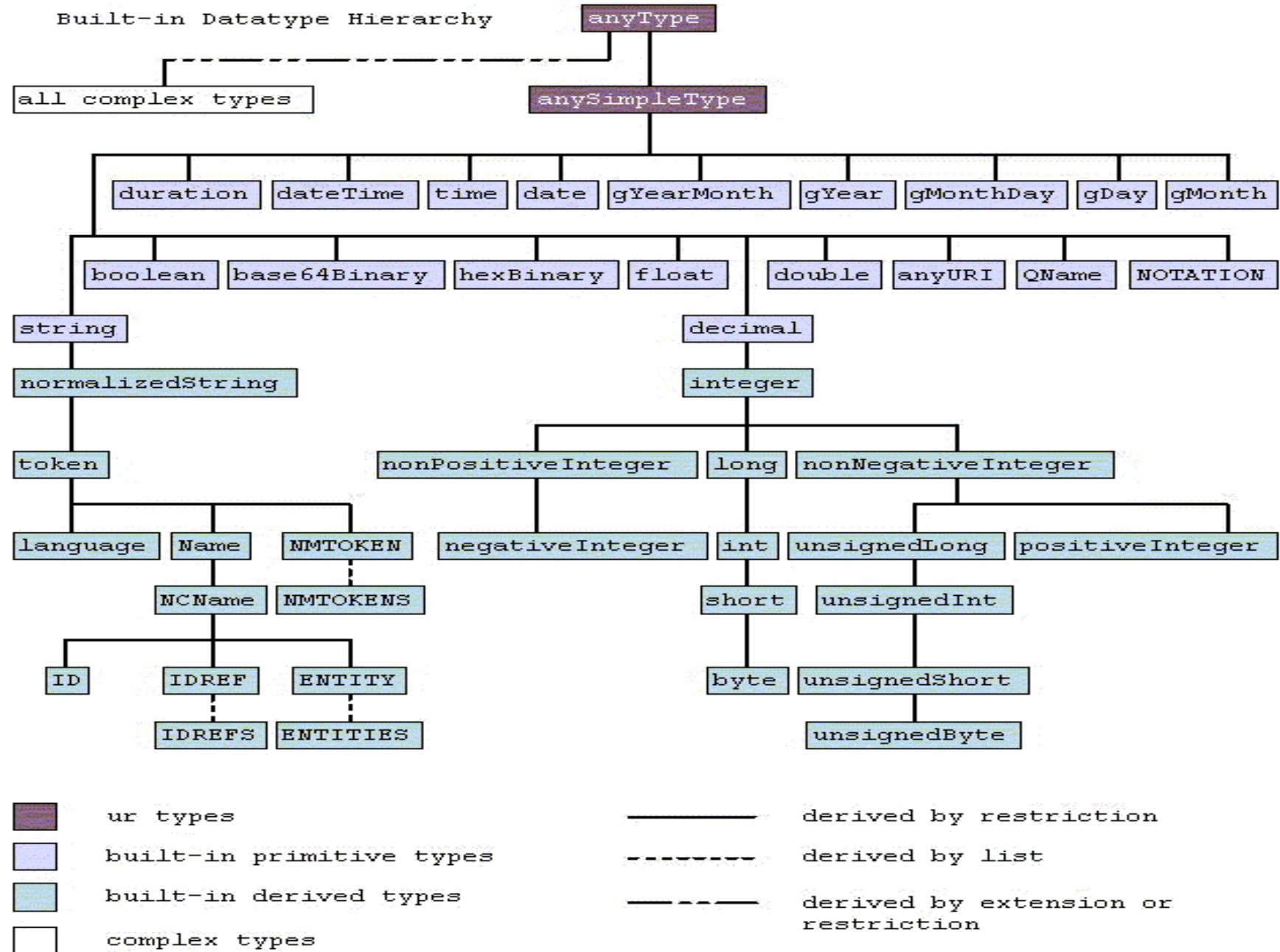
Um XML namespace permite criar espaço de nomes para os nomes dos elementos e atributos usados.

Um XML schema define os elementos que aparecerão num documento XML

```
<person id="123456789">
  <name>Smith</name>
  <place>London</place>
  <year>1934</year>
</person >
```

```
<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>
```

TIPOS XML



JSON (JAVASCRIPT OBJECT NOTATION)

JSON permite descrever estruturas de dados complexas em formato de texto

Tipos primitivos

Number

String

Boolean

Tipos complexos

Array

Object (mapa chave / valor)

JSON é uma alternativa ao XML

```
{ "Person": {  
    "name": "Smith",  
    "place": "London",  
    "year": 1934,  
}  
}
```

```
{ "Person": {  
    "name": "Smith",  
    "place": "London",  
    "year": 1934,  
    "phone": [99999999,  
              88888888],  
}  
}
```

PROTOBUF (GOOGLE PROTOCOL BUFFERS)

```
message Person {  
    required string name = 1;  
    required int32 id = 2;  
    optional string email = 3;  
  
    enum PhoneType {  
        MOBILE = 0;  
        HOME = 1;  
        WORK = 2;  
    }  
  
    message PhoneNumber {  
        required string number = 1;  
        optional PhoneType type = 2  
        [default = HOME];  
    }  
    repeated PhoneNumber phone = 4;  
}
```

```
person {  
    name: "John Doe"  
    id: 13  
    email: "jdoe@example.com"  
}
```

Dados passam na rede em formato binário

Menor dimensão, mais rápido a processar

E.g. protobuf: 28 bytes; 100 ns
XML: 69 bytes; 5000 ns

PROTOBUF (GOOGLE PROTOCOL BUFFERS)

Dados passam na rede em formato binário

Compilador cria código para serializar/deserializar dados estruturados

Resultado: menor dimensão, mais rápido a processar

E.g. protobuf: 28 bytes; 100-200 ns

XML: 69 bytes; 5000-10000 ns

... E MUITOS MAIS

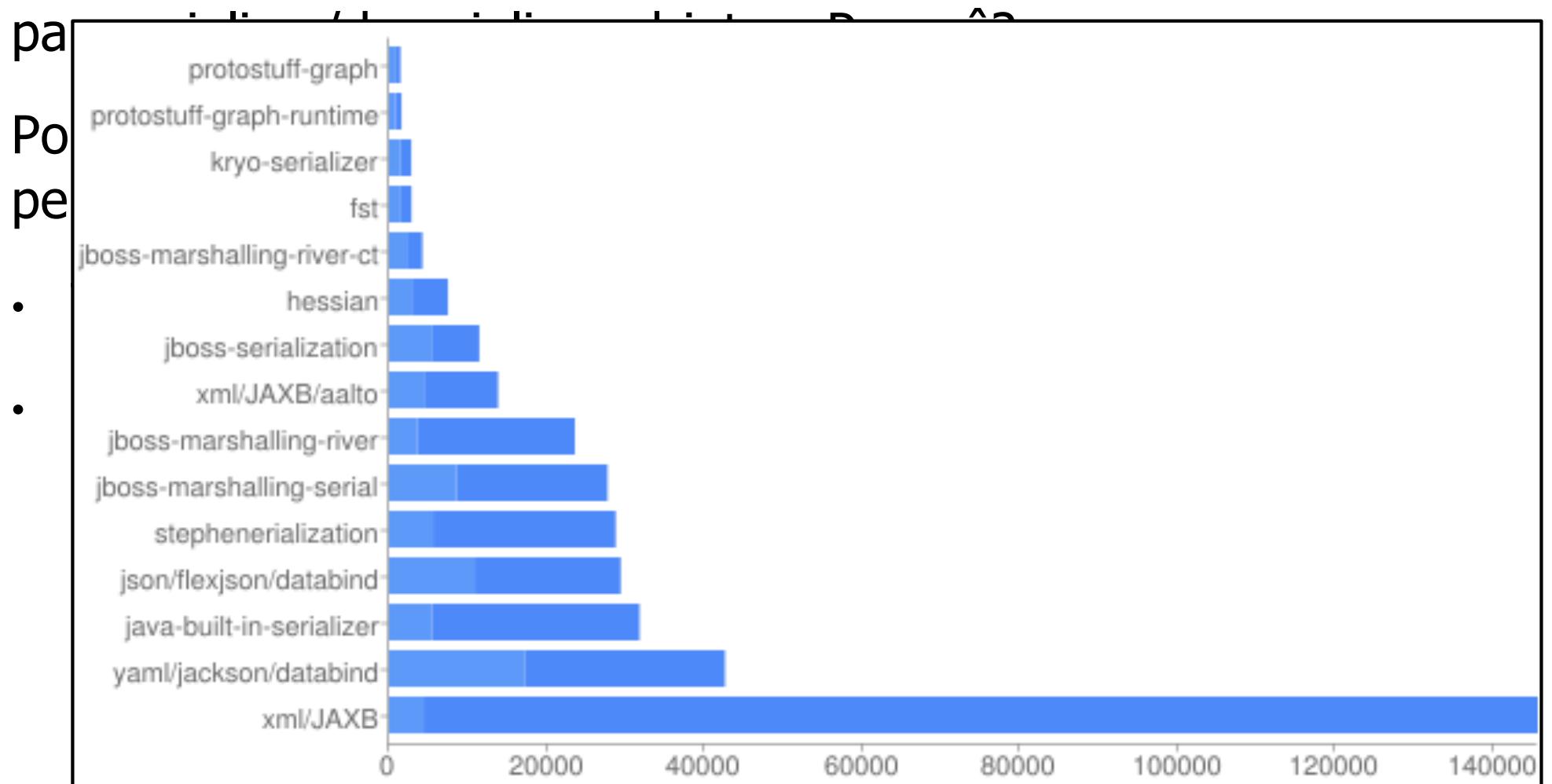
Existe um grande número de soluções para serializar/deserializar objetos. Porquê?

Porque a serialização/deserialização pode ter impacto na performance dos sistemas distribuídos:

- Tempo de serialização/deserialização
- Dimensão das mensagens => tempo de propagação das mensagens

... E MUITOS MAIS

Existe um grande número de soluções



Source: <https://github.com/eishay/jvm-serializers/wiki>

REPRESENTAÇÕES DOS DADOS: CLASSIFICAÇÃO

Conteúdo da representação

- Formato binário – Java, protobuf
- Formato de texto – XML, JSON

Integração com linguagem

- Independente – XML, JSON, protobuf
- Integrado – Java, JSON

Informação de tipos

- Incluída – Java, XML
- Não incluída – JSON, protobuf

MÉTODOS DE PASSAGEM DE PARÂMETROS

Numa linguagem de programação, independentemente dos tipos dos parâmetros, os mesmos podem ser:

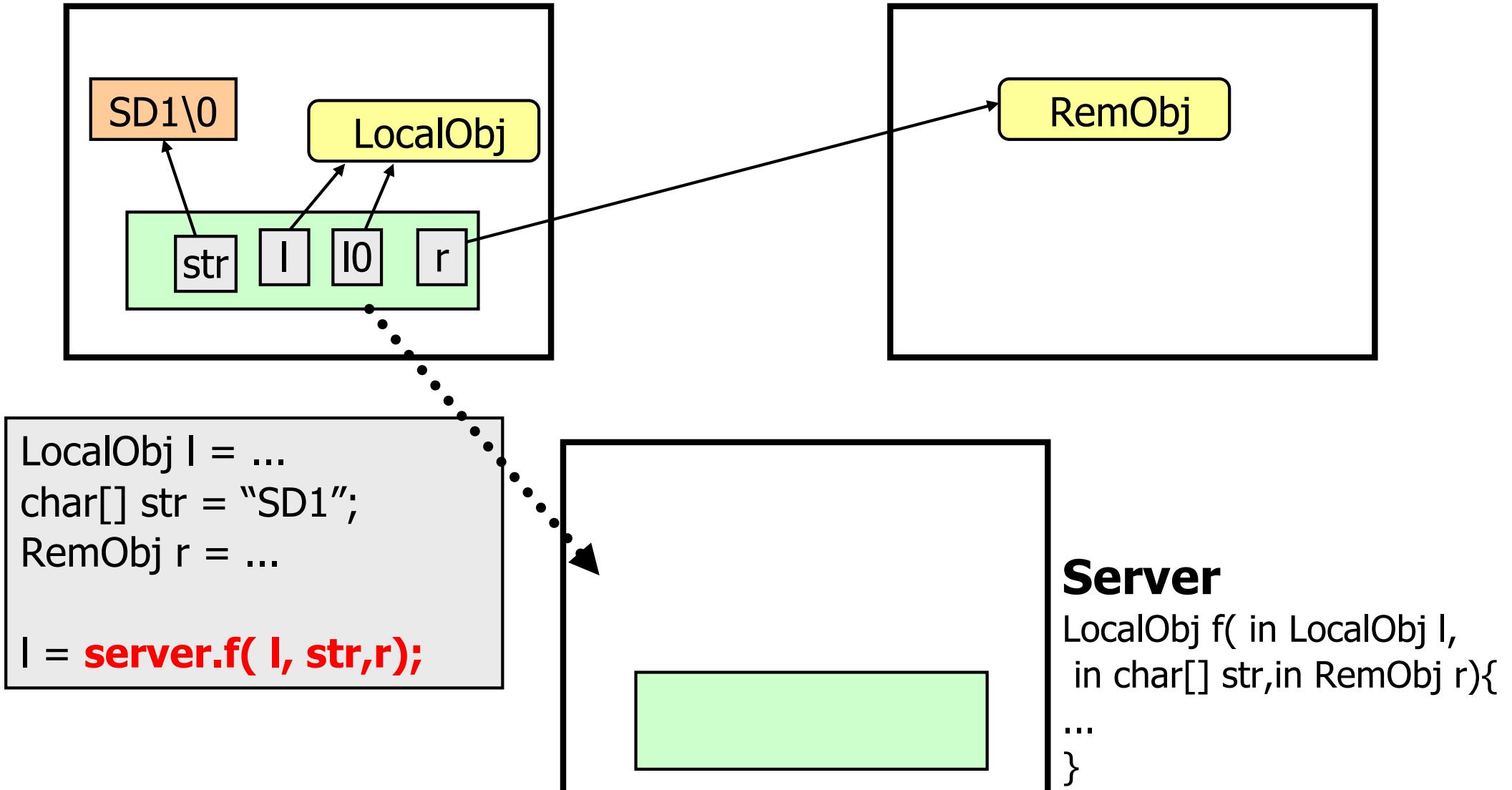
- Passados por valor
- Passados por referência

MÉTODOS DE PASSAGEM DE PARÂMETROS (CONT.)

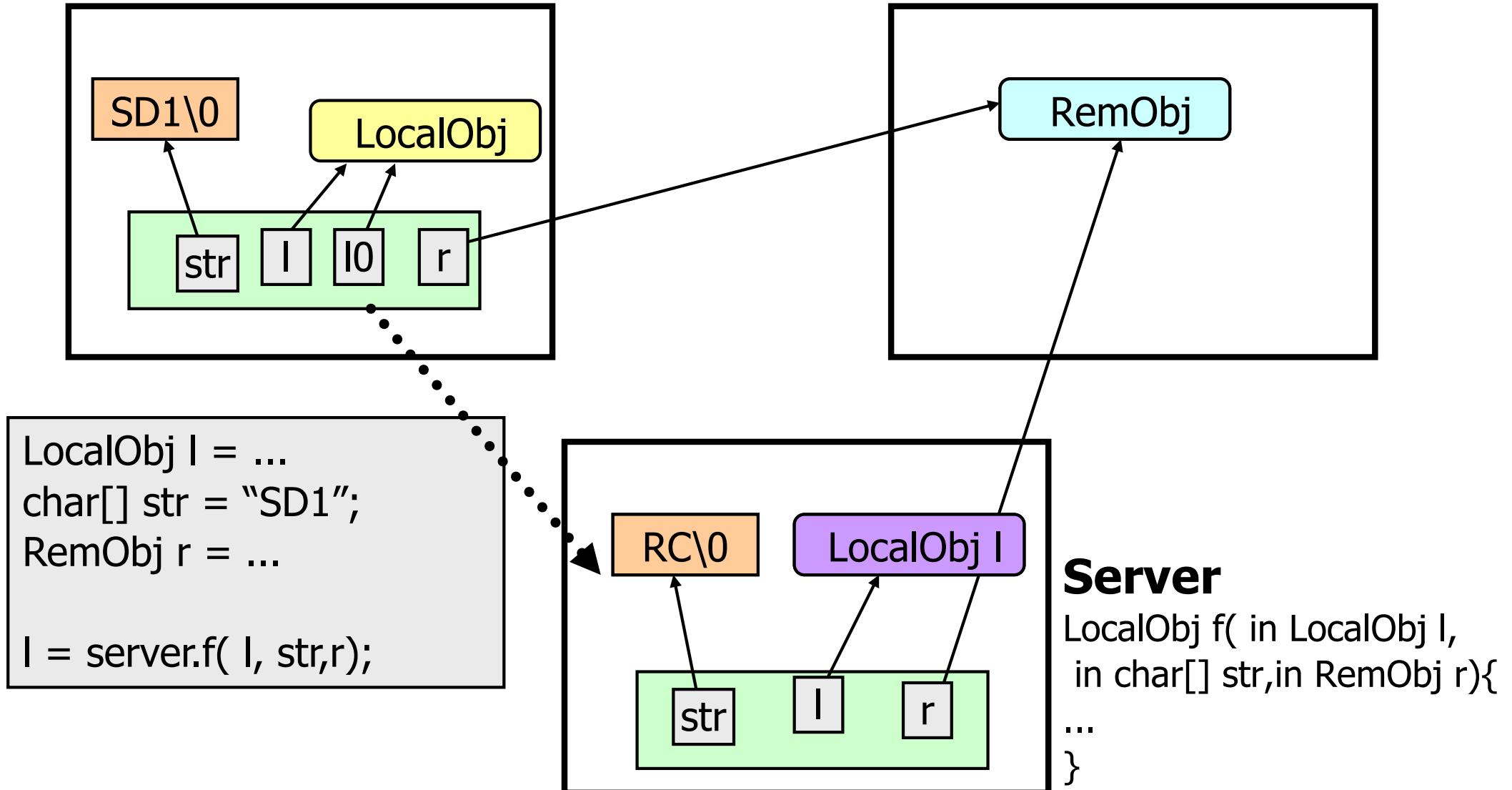
Aproximação comum nos sistemas de RPC/RMI:

- Passagem por valor para tipos primitivos, arrays, estruturas e objetos
 - Apontadores/referências para arrays, objetos, etc. são seguidas
 - Estado dos objetos é copiado (ex: Java RMI)
 - Porque não passar tipos básicos por referência?
- Passagem por referência para objetos remotos
 - quando o tipo de um parâmetro é um objeto remoto, uma referência para o objeto é transferida
 - Porque não passar objetos remotos por valor?

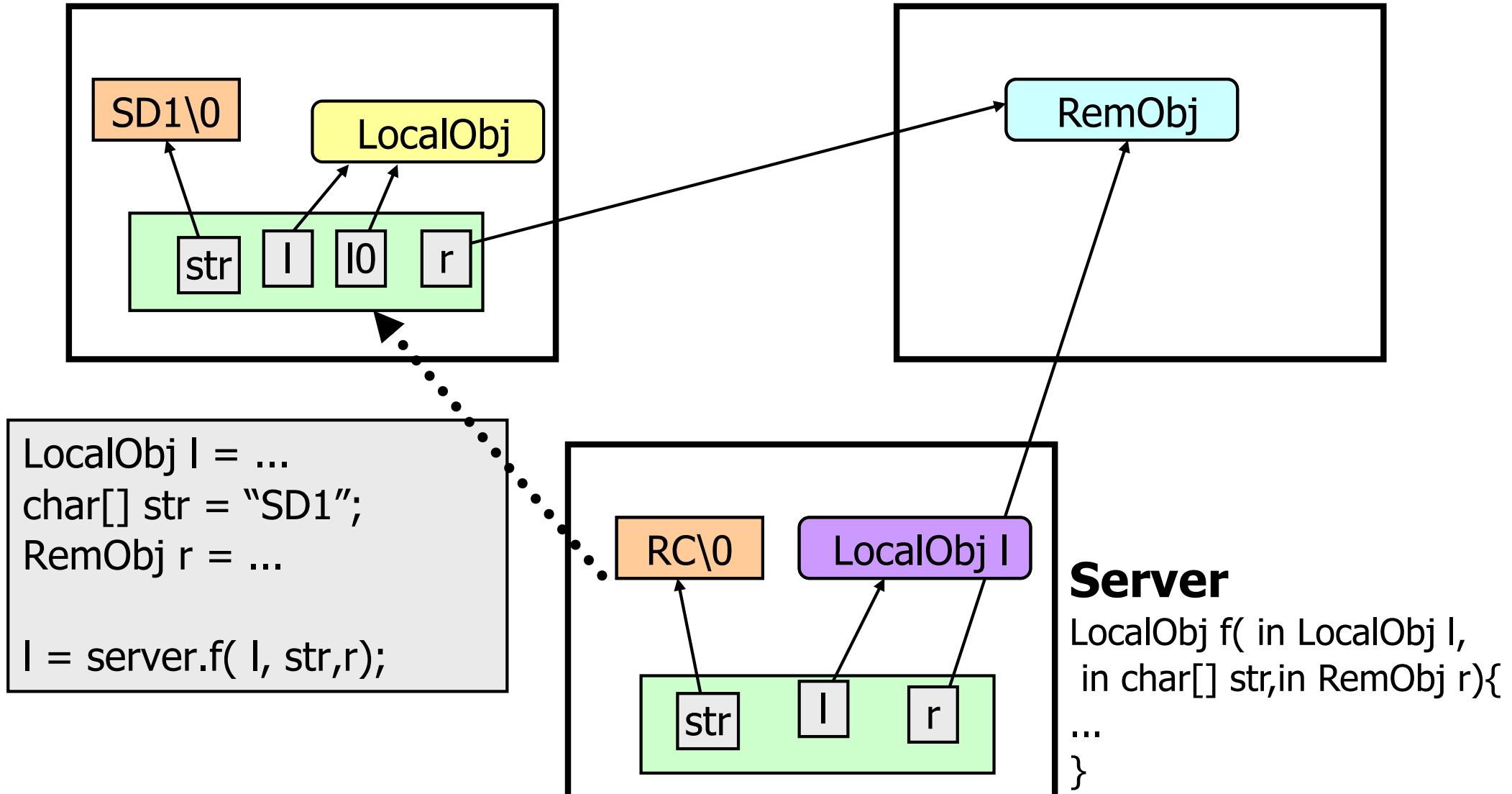
MÉTODOS DE PASSAGEM DE PARÂMETROS: EXEMPLO



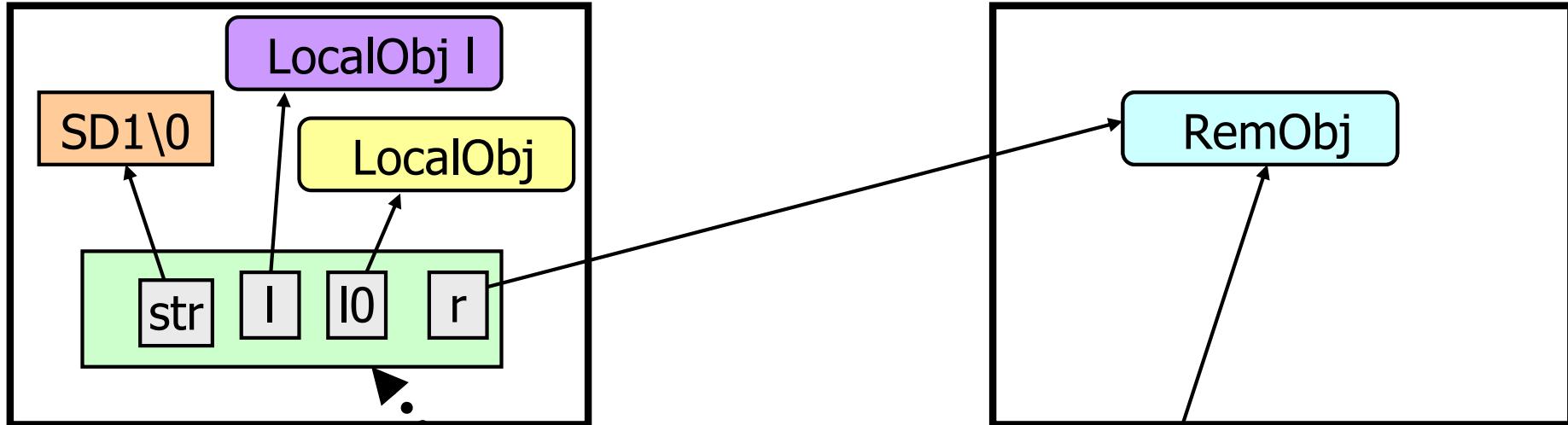
MÉTODOS DE PASSAGEM DE PARÂMETROS: EXEMPLO



MÉTODOS DE PASSAGEM DE PARÂMETROS: EXEMPLO

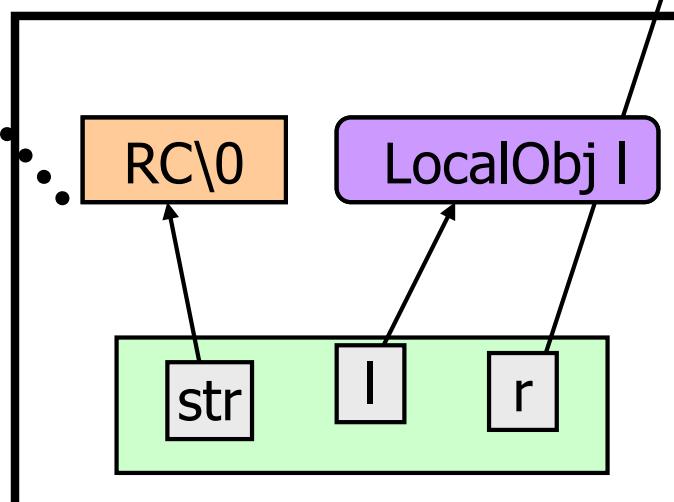


MÉTODOS DE PASSAGEM DE PARÂMETROS: EXEMPLO



```
LocalObj I = ...
char[] str = "SD1";
RemObj r = ...

I = server.f( I, str,r);
```



Server

```
LocalObj f( in LocalObj I,
in char[] str,in RemObj r){
...
}
```

PASSAGEM DE OBJECTOS REMOTOS EM PARÂMETRO

Nos sistemas de RMI é, em geral, possível passar (referências para) objetos remotos em parâmetro (ou como resultado de uma operação)

Em Java RMI pode-se enviar uma referência para um objecto remoto:

- Passando como parâmetro/resultado uma referência remota – neste caso, uma cópia da referência remota é enviada
- Passando como parâmetro/resultado o objecto servidor – neste caso, uma referência para o objecto remoto é enviada (e não o próprio objecto) – passagem por referência

Uma referência remota inclui, pelo menos, a seguinte informação:

- Endereço/porta do servidor
- Tipo do servidor
- Identificador Único

PASSAGEM DE OBJECTOS REMOTOS EM PARÂMETRO

Nos sistemas de RMI é, em geral, possível passar (referências para) objectos remotos em parâmetro (ou como resultado de uma operação)

Em Java RMI pode-se enviar uma re

- Passando como parâmetro/resu
uma cópia da referência remota
- Passando como parâmetro/resu
referência para o objecto remoto
é enviada (e não o próprio objecto)
passagem por referência

Com esta representação seria fácil mudar a localização do objecto?

Não. Para tal, a referência remota não deve incluir directamente a localização do objecto.

Uma referência remota inclui, pelo menos, a seguinte informação:

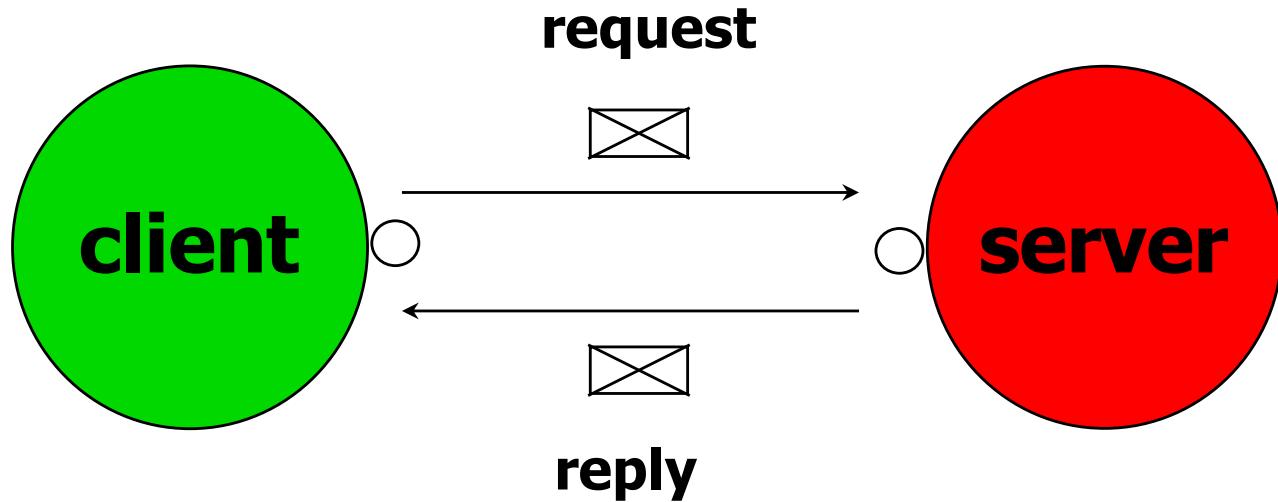
- Endereço/porta do servidor
- Tipo do servidor
- Identificador Único

NA AULA DE HOJE

Invocação remota de procedimentos/objects

- Motivação
- Modelo
- Definição de interfaces e método de passagem de parâmetros
- Codificação dos dados
- **Mecanismos de ligação (binding)**
- Protocolos de comunicação
- Sistemas de objetos distribuídos

LIGAÇÃO DO CLIENTE AO SERVIDOR (*BINDING*)



Para poder invocar o servidor, o cliente tem de obter uma referência para o servidor

Nos sistemas de RPC, uma referência corresponde ao endereço do servidor – endereço IP + porta + ...

Nos sistemas de RMI, a referência remota corresponde geralmente a um proxy com a mesma interface do servidor (que internamente inclui informação de localização do servidor)

Como obter essa referência?

COMO OBTER REFERÊNCIA PARA O SERVIDOR?

Por configuração directa

- Ex.: REST, Web services, .NET remoting

Servidor de nomes regista associação entre nome e referência remota

- Ex.: Java RMI

Servidor de nomes e directório regista informação sobre servidores

- Ex. Universal Directory and Discovery Service – UDDI (web services)
- Além de permitir obter servidor dado o nome, permite procurar servidor pelos seus atributos

Cliente procura servidor usando multicast/broadcast

- Alguns sistemas de objectos distribuídos usavam esta aproximação

POR CONFIGURAÇÃO DIRECTA

No código do cliente, para obter uma referência para o servidor indica-se explicitamente a sua localização

.NET remoting

```
ChannelServices.RegisterChannel(new TcpChannel());  
HelloServer obj = (HelloServer)Activator.GetObject(  
    typeof(Examples.HelloServer),  
    "tcp://localhost:8085/SayHello");
```

Web services

```
URL wsdlURL = new URL("http://localhost:8080/indexer?wsdl");  
Service service = Service.create(wsdlURL, IndexerService.QNAME);  
proxy = service.getPort(IndexerAPI.class);
```

Ao criar o código da referência remota a partir da descrição do serviço, a mesma inclui a localização do servidor

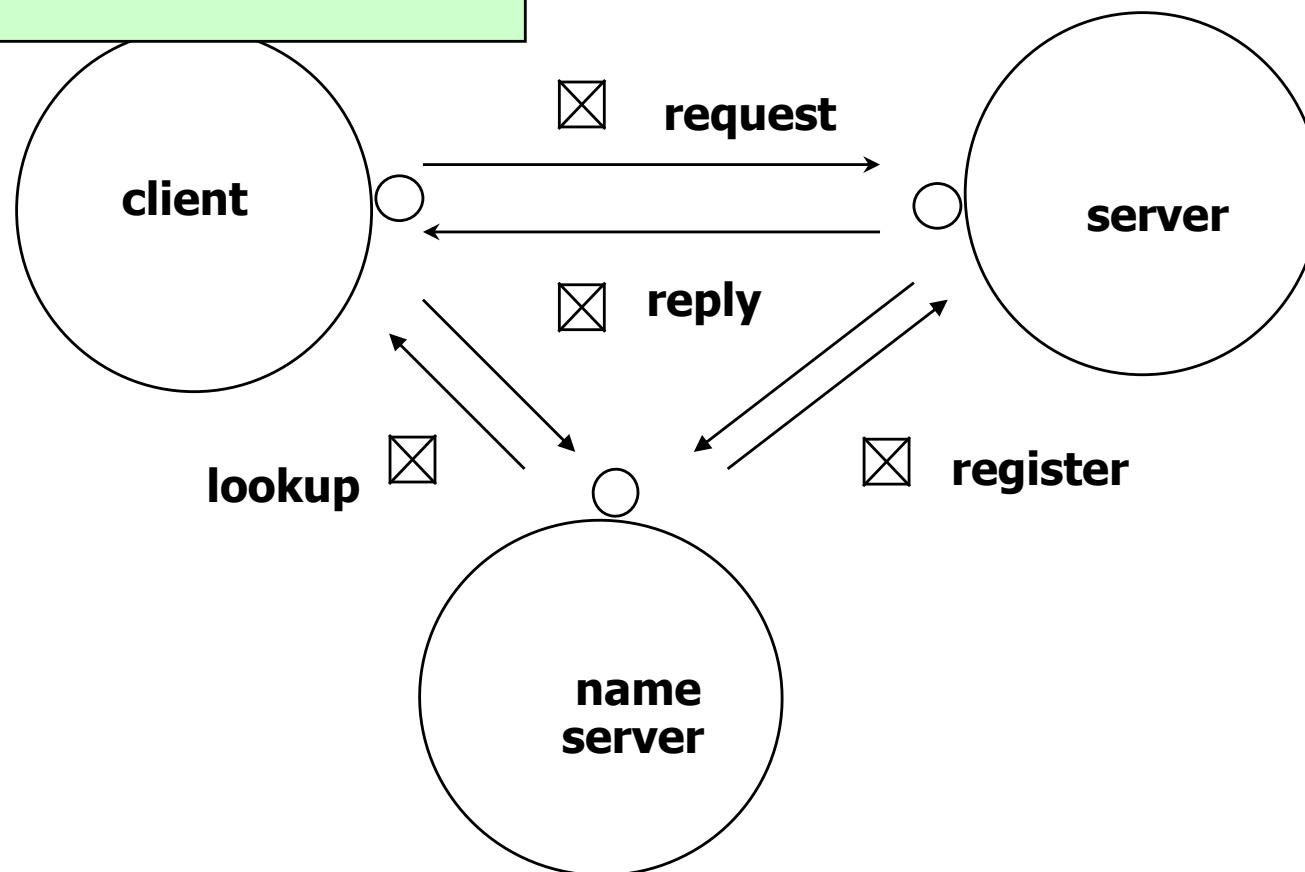
USAR UM SERVIÇO (*BINDING, NAMING OR TRADING*)

Java RMI

```
IServer s = Naming.lookup( "service")
res = s.fun( obj)
```

Java RMI

```
Naming.rebind( "service", this)
```



EXEMPLO: INTERFACE DA *REGISTRY* JAVA RMI

void rebind (String name, Remote obj)

- This method is used by a server to register the identifier of a remote object by name.

void bind (String name, Remote obj)

- This method can alternatively be used by a server to register a remote object by name, but if the name is already bound to a remote object reference an exception is thrown.

void unbind (String name, Remote obj)

- This method removes a binding.

Remote lookup(String name)

- This method is used by clients to look up a remote object by name. A remote object reference is returned. The code of the remote reference may be downloaded, if necessary. It encodes the protocol to be used.

String [] list()

- This method returns an array of Strings containing the names bound in the registry.

PROBLEMAS QUANDO SE USA UM SERVIDOR DE NOMES?

Como encontrar o servidor de nomes?

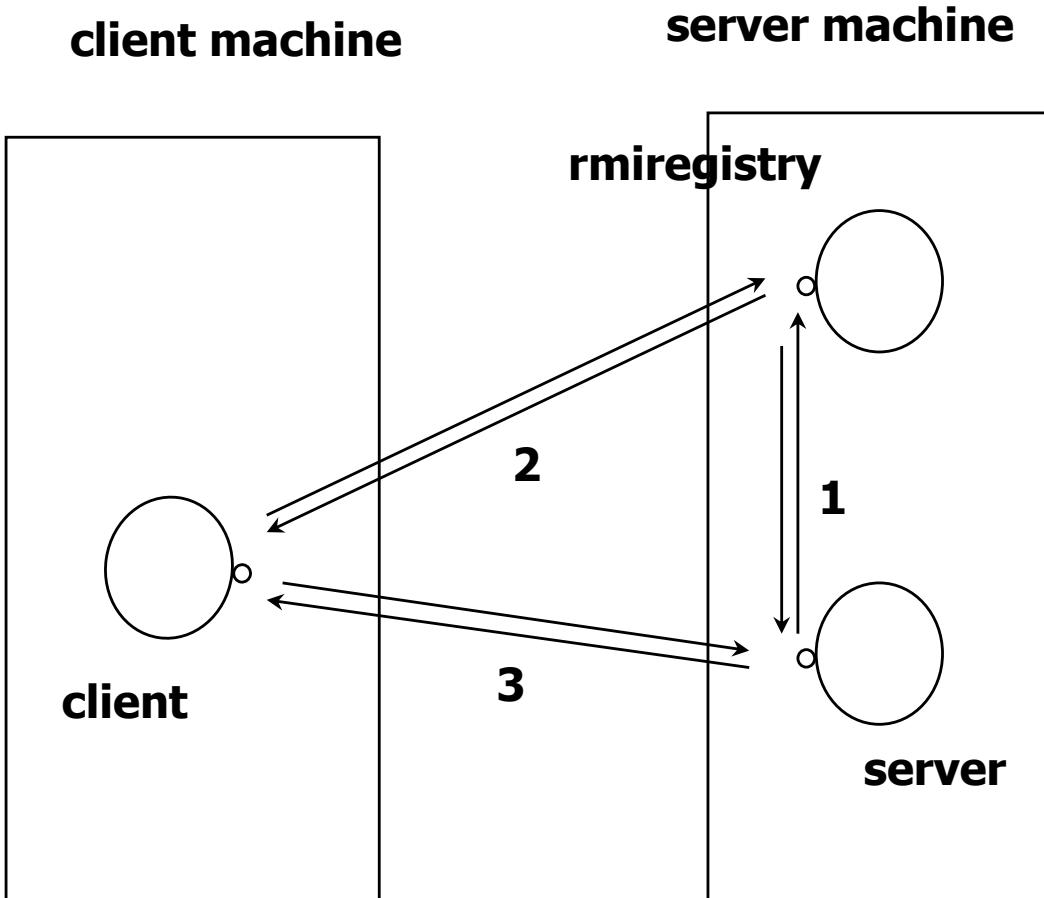
- Nome do objecto indica máquina em que está o servidor de nomes
- Servidor de nomes descoberto por multicast/broadcast

O servidor de nomes pode ser único ?

- No Java RMI não
- No SOAP, o espaço de nomes do UDDI é único e global

Problemas de segurança – como evitar que haja serviços / objectos impostores ou que atacantes impeçam o acesso ao serviço?

SOLUÇÃO PRAGMÁTICA DO JAVA RMI



Em cada máquina existe um RMI registry. O cliente tem de saber em que máquina está o serviço/objeto remoto em que está interessado.

Em cada máquina, só pode estar associado uma instância de uma interface/classe a cada nome

Pode existir mais do que uma instância da mesma interface/classe associados a nomes diferentes

Porque é que esta solução diminui os problemas de segurança?

SISTEMAS DISTRIBUÍDOS

Capítulo 5

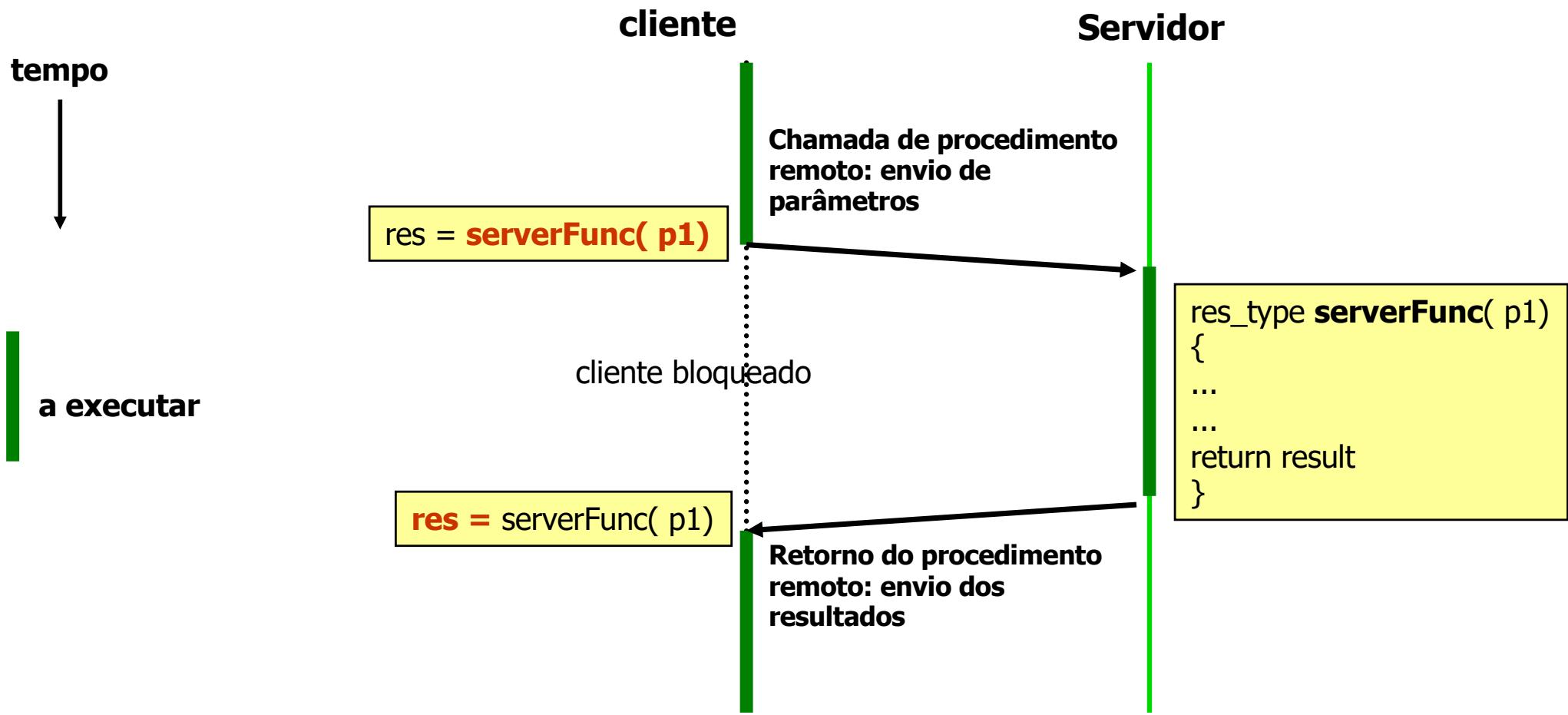
Invocação Remota na Web

NOTA PRÉVIA

A apresentação utiliza algumas das figuras do livro de base do curso

G. Coulouris, J. Dollimore and T. Kindberg,
Distributed Systems - Concepts and Design,
Addison-Wesley, 5th Edition, 2005

NAS ÚLTIMAS AULAS... INVOCAÇÃO REMOTA DE PROCEDIMENTOS



Modelo

Servidor exporta interface com operações que sabe executar
Cliente invoca operações que são executadas remotamente e (normalmente) aguarda pelo resultado

INVOCAÇÃO REMOTA DE PROCEDIMENTOS

Invocação remota de procedimentos/objectos

- Motivação
- Modelo
- Organização interna do servidor
- Definição de interfaces e método de passagem de parâmetros
- Codificação dos dados
- Mecanismos de ligação (binding)
- Semântica na presença de falhas
- Sistemas de objetos distribuídos

ORGANIZAÇÃO DO CAPÍTULO

Web services REST

Web services SOAP

INVOCAÇÃO REMOTA NA WEB

Os Web Services são uma forma de invocação remota orientada para a Web. O uso do termo Web advém de:

- Utilização do protocolo HTTP como suporte base à comunicação;
- Os próprios serviços poderem ser acessíveis ao nível da WWW (páginas e sites Web), podendo operar como componentes de aplicações web.

Em geral, tiram partido da implantação global do protocolo HTTP, da WEB e das suas tecnologias.

TIPOS DE WEB SERVICES

WebServices REST

A aplicação é estruturada em coleções de recursos, acedidos por HTTP através de um URL, onde a semântica das operações está mapeada para as várias operações HTTP, tais como GET, POST, DELETE, etc.

WebServices SOAP

Oferece uma forma de invocação remota cujo foco está na interoperabilidade. Por via da standardização dos seus componentes consegue ser independente da linguagem e da plataforma de sistema.

ORGANIZAÇÃO DO CAPÍTULO

Web services REST

Web services SOAP

REST: REPRESENTATIONAL STATE TRANSFER

Aproximação: uma aplicação é modelada como uma coleção de recursos

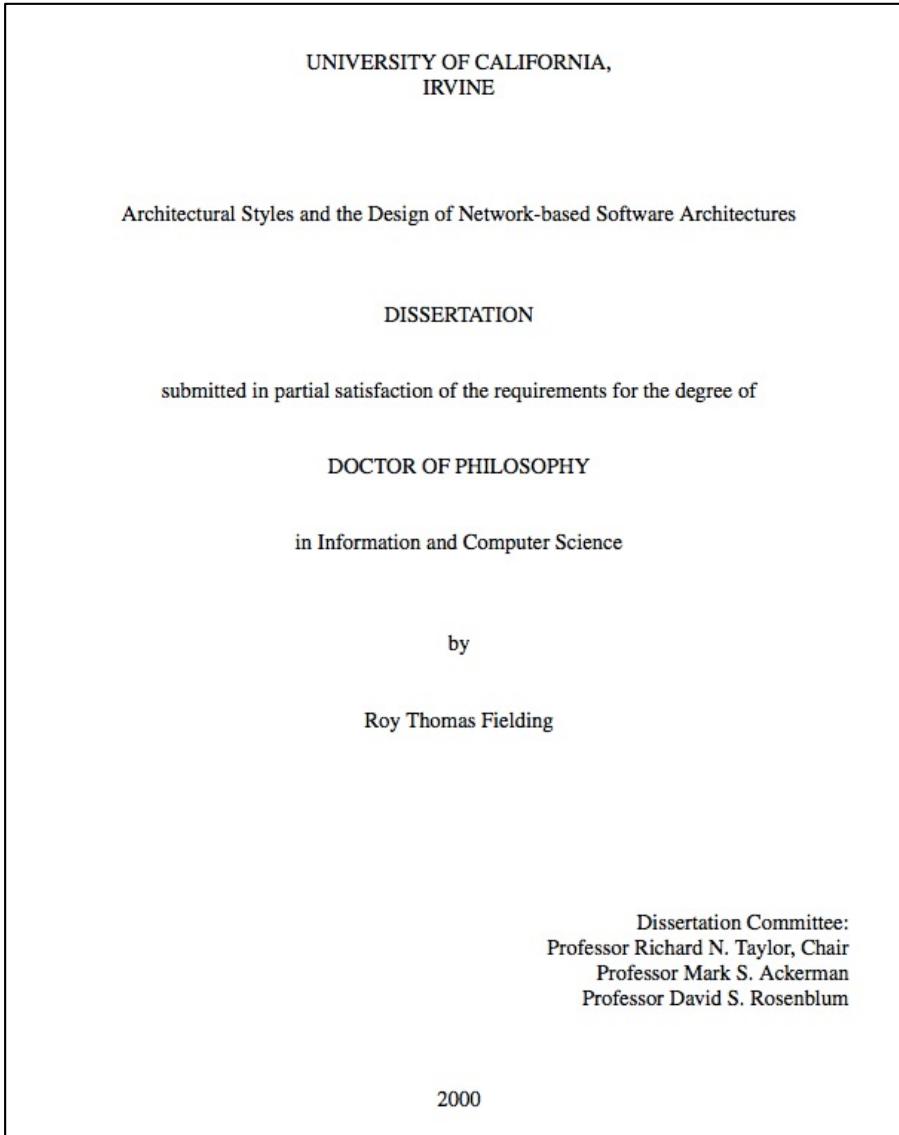
- Um recurso é identificado por um URI/URL
- Um URL devolve um documento com a representação do recurso
- Podem-se fazer referências a outros recursos usando *ligações (links)*

Estilo arquitetural, não um sistema de desenvolvimento

Aproximação proposta por Roy Fielding na sua tese de doutoramento

- Não como uma alternativa aos web services SOAP, mas como uma forma de aceder a informação

ROY FIELDING



- Author of HTTP specification
- Co-founder of Apache HTTP server

REST: PRINCÍPIOS DE DESENHO

Protocolo cliente/servidor stateless: cada pedido contém toda a informação necessária para ser processado – objetivo: tornar o sistema simples.

Recursos – no sistema tudo são recursos, identificados por um URI/URL.

- Recursos tipicamente armazenados num formato estruturado que suporta hiper-ligações (e.g. JSON, XML)

Interface uniforme: todos os recursos são acedidos por um conjunto de operações bem-definidas. Em HTTP: POST, GET, PUT, DELETE (equiv. criar, ler, actualizar, remover).

Caching: para melhorar desempenho, respostas podem ser etiquetadas como permitindo ou não *caching*, via o cabeçalho HTTP addequado.

REST: CONVENÇÕES E BOAS PRÁTICAS

As operações HTTP sobre um recurso, com um dado URL/URI têm a semântica pré-definida:

- **GET** lê o recurso, reportando 404 NOT FOUND se o recurso não existir;
- **POST** cria um novo recurso, reportando 409 CONFLICT, se o recurso já existir;
- **PUT** atualiza um recurso, reportando 404 NOT FOUND se o recurso não existir;
- **DELETE** apaga o recurso, reportando 404 NOT FOUND se o recurso não existir.

REST: CONVENÇÕES E BOAS PRÁTICAS (2)

Para atualizar parcialmente um recurso pode-se usar a operação:

- **PATCH** atualiza parcialmente um recurso, reportando 404 NOT FOUND se o recurso não existir.

Os dados envolvidos nas operações são bem-tipados, segundo um ***schema*** JSON ou XML .

EXEMPLO

Considere-se um serviço que mantém informação sobre servidores, disponibilizando as seguintes operações:

- Adicionar um servidor
- Remover um servidor
- Modificar um servidor
- Obter informação sobre um servidor, dado o seu id
- Pesquisar informação dum servidor

EXEMPLO: ADICIONAR SERVIDOR

- **URL:** `http://myserver.com/server/35345645`
- **Método:** POST
- **Body:**

```
{ id: "35345645",
  name : "server 1",
  props : ["a", "b", "c"]
}
```

Alternativa 1: usar o URL `http://myserver.com/server/` para a criação, sendo o id passado no objeto.

EXEMPLO: ADICIONAR SERVIDOR

- **URL:** `http://myserver.com/server/35345645`
- **Método:** POST
- **Body:**

```
{ id: "35345645",
name : "server 1",
props : ["a", "b", "c"]}
```

Alternativa 2: usar o URL `http://myserver.com/server/`, sendo o id criado no serviço e devolvido como resultado do método.

EXEMPLO: MODIFICAR SERVIDOR

- **URL:** `http://myserver.com/server/35345645`
- **Método:** PUT
- **Body:**

```
{ id: "35345645",
  name : "server 1",
  props : ["a", "b", "Z"]
}
```

A atualização tipicamente fornece uma cópia integral do recurso.

EXEMPLO: REMOVER SERVIDOR

- **URL:** `http://myserver.com/server/35345645`
- **Método:** `DELETE`
- **Body:** `<empty>`

EXEMPLO: OBTER INFORMAÇÃO DE SERVIDOR

- **URL:** `http://myserver.com/server/35345645`
- **Método:** GET
- **Body:**
- **Reply:**

```
{ id: "35345645",
  name : "server 1",
  props : ["a", "b", "z"]
}
```

EXEMPLO: LISTAR SERVIDORES

- **URL:** `http://myserver.com/server`
- **Método:** GET
- **Body:** <empty>
- **Reply:**

```
[{ id: "35345645",
  name : "server 1",
  props : ["a", "b", "z"]
}, ...]
```

EXEMPLO: PESQUISAR SERVIDORES

- **URL:** `http://myserver.com/server?query=a`

- **Método:** GET

- **Body:** <empty>

Pode-se usar um query parameter para filtrar os resultados que se pretendem obter

- **Reply:**

```
[{ id: "35345645",
  name : "server 1",
  props : ["a", "b", "z"]
}, ...
]
```

CODIFICAÇÃO DOS DADOS: XML vs. JSON

A informação transmitida nos pedidos e nas respostas é codificada tipicamente em JSON ou XML.

JSON é muito popular devido à facilidade de processamento em JavaScript e, por conseguinte, em páginas Web e aplicações móveis.

Dados binários são transferidos como HTTP octet-stream.

CODIFICAÇÃO DOS DADOS: XML VS. JSON

```
<Person firstName='John'  
lastName='Smith' age='25'>  
  <Address streetAddress='21 2nd  
Street' city='New York' state='NY'  
postalCode='10021' />  
  <PhoneNumbers home='212 555-  
1234' fax='646 555-4567' />  
</Person>
```

(Example from wikipedia)

```
{ "Person": {  
  "firstName": "John",  
  "lastName": "Smith",  
  "age": 25,  
  "Address": {  
    "streetAddress": "21 2nd Street",  
    "city": "New York",  
    "state": "NY",  
    "postalCode": "10021"  
  },  
  "PhoneNumbers": {  
    "home": "212 555-1234",  
    "fax": "646 555-4567"  
  }  
}
```

REST: SUMÁRIO

Baseado em protocolos standard

- Comunicação: HTTP / HTTPS
- Identificação de recursos: URL/URI
- Representação dos recursos: JSON, XML

O foco na simplicidade e no desempenho tornou o modelo REST muito popular.

Usado nas APIs públicas de serviços populares como: Twitter, Imgur, Facebook, Dropbox, etc.

- Alguns destes serviços forneceram anteriormente APIs em Web Services SOAP, que depois foram descontinuadas.

REST: TEORIA VS. PRÁTICA

Há muitos exemplos de serviços disponibilizados num modelo REST que nem sempre aderem completamente aos princípios REST.

Dropbox (apagar um ficheiro)

Operação: POST

URL: https://api.dropboxapi.com/2/files/delete_v2

Body: { "path": "/Homework/math/Prime_Numbers.txt" }

Na API da Dropbox, um ficheiro para efeitos da sua remoção não é um recurso a apagar com HTTP DELETE.

REST: SUPORTE JAVA

Definido em JAX-RS (JSR 311)

Suporte linguístico baseado na utilização de anotações

- Permite definir que um dado URL leva à execução dum dado método
- Permite definir o modo de codificação da resposta
 - JSON – mecanismo leve de codificação de tipos (RFC 4627)
 - XML – usando mecanismo standard de codificação de cobjectos java em XML fornecido pelo JAXB

JAX-RS: SERVIDOR

Do lado do servidor, o suporte Java para REST (JAX-RS / JSR 311) permite ao programador concentrar-se na lógica das operações e evitar muitos aspectos de baixo nível, ao nível do protocolo HTTP, conversão de dados, etc.

JAX-RS: SERVIDOR (EXEMPLO)

```
@Singleton  
@Path(ServerService.PATH)  
public class ServerService {  
    public static final String PATH = "/server";  
  
    @POST  
    @Consumes(MediaType.APPLICATION_JSON)  
    public Response addServer(Server server) { ..... }  
  
    @GET  
    @Path("/{id}")  
    @Produces(MediaType.APPLICATION_JSON)  
    public Server getServer(@PathParam("id") String id) {....}  
  
    @GET  
    public List<Server> listServer(@QueryParam("query") q)  
    {...}
```

JAX-RS: SERVIDOR (EXEMPLO)

@Singleton permite indicar que se quer apenas uma instância do recurso.

@Singleton

```
@Path(ServerService.PATH)
public class ServerService {
    public static final String PATH = "/server";

    @POST
    @Consumes(MediaType.APPLICATION_JSON)
    public Response addServer(Server server) { ..... }

    @GET
    @Path("/{id}")
    @Produces(MediaType.APPLICATION_JSON)
    public Server getServer(@PathParam("id") String id) {....}

    @GET
    public List<Server> listServer(@QueryParam("query") q)
    {...}
```

JAX-RS: SERVIDOR (EXEMPLO)

```
@Singleton
```

@Path permite definir a path, que será concatenada ao URL base do servidor

```
@Path(ServerService.PATH)
```

```
public class ServerService {
```

```
    public static final String PATH = "/server";
```

```
@POST
```

```
@Consumes(MediaType.APPLICATION_JSON)
```

```
public Response addServer(Server server) { ..... }
```

```
@GET
```

```
@Path("/{id}")
```

```
@Produces(MediaType.APPLICATION_JSON)
```

```
public Server getServer(@PathParam("id") String id) {....}
```

```
@GET
```

```
public List<Server> listServer(@QueryParam("query") q)
```

```
{...}
```

JAX-RS: SERVIDOR (EXEMPLO)

```
@Singleton  
@Path(ServerService.PATH)  
public class ServerService {  
    public static final String PATH = "/server";  
    ...  
    @POST / @GET / @PUT / @DELETE  
    Operação HTTP que leva à execução do  
    método  
  
    @POST  
    @Consumes(MediaType.APPLICATION_JSON)  
    public Response addServer(Server server) { ..... }  
  
    @GET  
    @Path("/{id}")  
    @Produces(MediaType.APPLICATION_JSON)  
    public Server getServer(@PathParam("id") String id) {....}  
  
    @GET  
    public List<Server> listServer(@QueryParam("query") q)  
    {...}
```

JAX-RS: SERVIDOR (EXEMPLO)

```
@Singleton  
@Path(ServerService.PATH)  
public class ServerService {  
    public static final String PATH = "/server";  
  
    @POST  
    @Consumes(MediaType.APPLICATION_JSON)  
    public Response addServer(Server server) { ..... }  
  
    @GET  
    @Path("/{id}")  
    @Produces(MediaType.APPLICATION_JSON)  
    public Server getServer(@PathParam("id") String id) {....}  
  
    @GET  
    public List<Server> listServer(@QueryParam("query") q)  
    {...}}
```

@Consumes usado para especificar o formato dos dados enviado no corpo do pedido (para o parâmetro server)

JAX-RS: SERVIDOR (EXEMPLO)

```
@Singleton  
@Path(ServerService.PATH)  
public class ServerService {  
    public static final String PATH = "/server";  
  
    @POST  
    @Consumes(MediaType.APPLICATION_JSON)  
    public Response addServer(Server server) { ..... }  
  
    @GET  
    @Path("/{id}")  
    @Produces(MediaType.APPLICATION_JSON)  
    public Server getServer(@PathParam("id") String id) {....}  
  
    @GET  
    public List<Server> listServer(@QueryParam("query") q)  
    {...}
```

@Producse usado para especificar o formato dos dados retornado no corpo da resposta

JAX-RS: SERVIDOR (EXEMPLO)

```
@Singleton  
@Path(ServerService.PATH)  
public class ServerService {  
    public static final String PATH = "/server";  
  
    @POST  
    @Consumes(MediaType.APPLICATION_JSON)  
    public Response addServer(...);  
  
    @GET  
    @Path("/{id}")  
    @Produces(MediaType.APPLICATION_JSON)  
    public Server getServer(@PathParam("id") String id) {...};  
  
    @GET  
    public List<Server> listServer(@QueryParam("query") q)  
    {...};
```

`@Path` permite definir a path a concatenar à define à path definida no recurso. `{id}` permite aceitar qualquer valor e atribui-lo a um parâmetro usando o `@PathParam("id")`

JAX-RS: SERVIDOR (EXEMPLO)

```
@Singleton  
@Path(ServerService.PATH)  
public class ServerService {  
    public static final String PATH = "/server";  
  
    @POST  
    @Consumes(MediaType.APPLICATION_JSON)  
    public Response addServer(Server server) { ..... }  
  
    @GET  
    @Path("/{id}")  
    @Produces(MediaType.APPLICATION_JSON)  
    public Server getServer(@PathParam("id") String id)  
        @QueryParam("query") String query) { ..... }  
  
    @GET  
    public List<Server> listServer(@QueryParam("query") String query) { ..... }
```

Observação:
@QueryParam permite obter um parâmetro opcional passado como um query parameter no URL

JAX-RS: SERVIDOR (EXEMPLO)

```
@PUT  
@Path("/{id}")  
@Consumes(MediaType.APPLICATION_JSON)  
public void updateServer(@PathParam("id") String id,  
                         Server srv) { ..... }  
  
@DELETE  
@Path("/{id}")  
@Produces(MediaType.APPLICATION_JSON)  
public Server delServer(@PathParam("id") String id) {...}  
}
```

JAX-RS: SERVIDOR (EXEMPLO)

Servidor por pedido

- `URI baseUri = UriBuilder.fromUri("http://0.0.0.0/").port(8080).build();`
- `ResourceConfig config = new ResourceConfig();`
- `config.register(ServerResource.class);`
- `JdkHttpServerFactory.createHttpServer(baseUri, config);`

Servidor único

- `URI baseUri = UriBuilder.fromUri("http://0.0.0.0/").port(8080).build();`
- `ResourceConfig config = new ResourceConfig();`
- `config.register(new ServerResource());`
- `JdkHttpServerFactory.createHttpServer(baseUri, config);`

JAX-RS: SERVIDOR : OBSERVAÇÕES

O programador define o mapeamento entre as operações REST sobre os recursos e os métodos Java correspondentes; especifica a codificação dos dados a utilizar JSON ou XML, mas não precisa de fornecer código para codificar/descodificar os dados.

A capacidade de introspeção da linguagem Java permite gerar o resto do código do servidor automaticamente. O programador não se preocupa com sockets, canais de entrada ou saída; HTTP 1.0/1.1 ou 2.0, etc.

JAX-RS: CLIENTE

O suporte linguistico é muito limitado. A invocação é concisa, mas não se assemelha a uma invocação de um procedimento local.

O programador tem que construir um pedido explicitamente, em vez de invocar uma única função com parâmetros e um resultado.

NOTA: Existem várias bibliotecas que automatizam o processo de criar clientes para servidores REST.

JAX-RS: CLIENTE (EXEMPLO)

```
ClientConfig config = new ClientConfig();
Client client = ClientBuilder.newClient(config);
WebTarget target = client.target( serverUrl )
    .path( RestMediaResources.PATH );
Response r = target.request()
    .accept(MediaType.APPLICATION_JSON)
    .post(Entity.entity(srv,
        MediaType.APPLICATION_JSON));
if( r.getStatus() == Status.OK.getStatusCode()
    && r.hasEntity() )
    System.out.println("Response: "
        +r.readEntity(String.class) );
else
    System.out.println("Status: " + r.getStatus());
```

JAX-RS: CLIENTE (EXEMPLO)

```
ClientConfig config = new ClientConfig();
Client client = ClientBuilder.newClient(config);
WebTarget target = client.target( serverUrl )
    .path( RestMediaResources.PATH );
Response r = target.request()
    .accept(MediaType.APPLICATION_JSON)
    .post(Entity.entity(srv,
        MediaType.APPLICATION_JSON));
if( r.getStatus() == Status.OK.getStatusCode()
    && r.hasEntity() )
```

“Clients are heavy-weight objects that manage the **client-side communication** infrastructure. Initialization as well as disposal of a Client instance may be a rather expensive operation. It is therefore **advised to construct only a small number of Client instances** in the application.”

Por exemplo, no trabalho não devem estar a criar clientes para o serviço Users de cada vez que precisam de verificar uma password... devem reusar o cliente existente.

ORGANIZAÇÃO DO CAPÍTULO

Web services REST

Web services SOAP

WEB SERVICES SOAP: MOTIVAÇÃO

*A Web service is a software system designed to support **interoperable machine-to-machine interaction** over a network.*

*It has an **interface described in a machine-processable format** (specifically WSDL).*

*Other systems interact with the Web service in a manner prescribed by its description using **SOAP messages**, typically **conveyed using HTTP** with an **XML serialization** in conjunction with other Web-related standards*

WEB SERVICES SOAP: CARACTERÍSTICAS PRINCIPAIS...

Modelo para acesso a servidores: **invocação remota**

Desenhado para garantir **inter-operabilidade**

Protocolo: **HTTP** e **HTTPS** (eventualmente SMTP)

Referências: URL/URI

Representação dos dados: **XML**

COMPONENTES BÁSICOS

SOAP: protocolo de invocação remota

Oneway

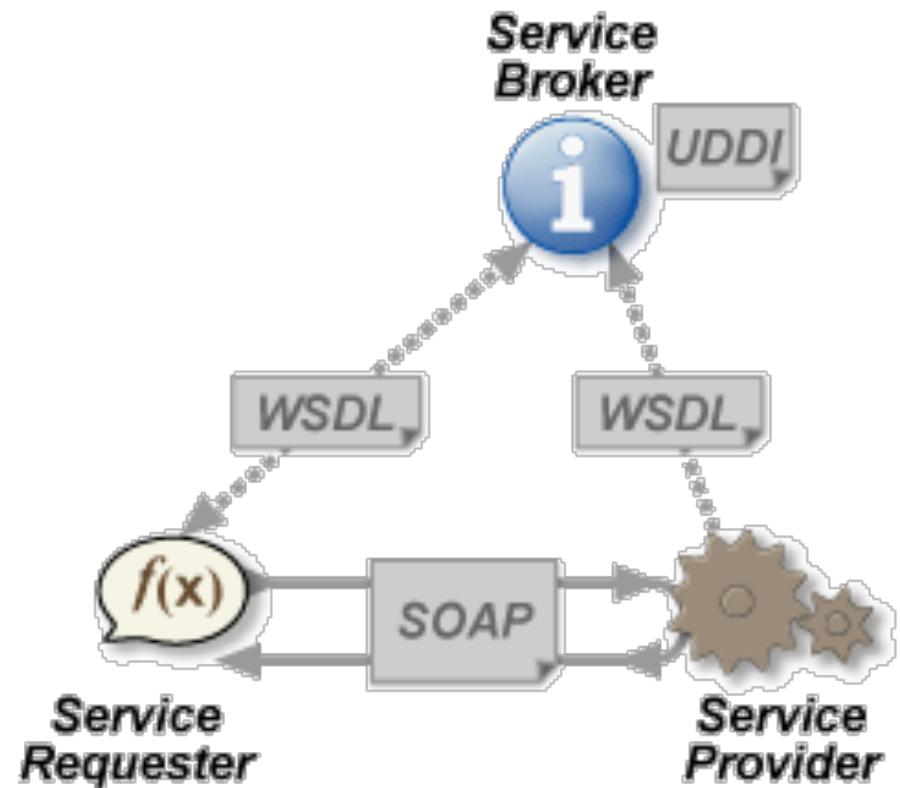
Pedido-resposta

Notificação

Notificação-resposta

WSDL: linguagem de especificação de serviços

UDDI: serviço de registo



SOAP (SIMPLE OBJECT ACCESS PROTOCOL)

Protocolo de comunicação visando a troca de informação estruturada na implementação de WebServices.

Estabelece as regras e as mensagens trocadas, usando o formato XML.

Procura ser neutro e independente da plataforma de sistema para garantir a inter-operabilidade.

Implementado sobre protocolos applicacionais tais como HTTP ou SMTP (Email).

WSDL (WEB SERVICES DESCRIPTION LANGUAGE)

Usado para definir um documento XML que descreve a interface de um Web Service, a vários níveis:

- Define a interface do serviço, indicando quais as operações disponíveis;
- Define as mensagens trocadas na interacção (e.g. na invocação duma operação, quais as mensagens trocadas);
- Permite definir a forma de representação dos dados;
- Descreve como e onde aceder ao serviço
 - Inclui o URL de uma instância do serviço.

WSDL (WEB SERVICES DESCRIPTION LANGUAGE)

Especificação WSDL bastante verbosa – normalmente criada a partir de interface ou código do servidor

- Em Java e .NET existem ferramentas para criar especificação a partir de interfaces Java
- Sistemas de desenvolvimento possuem *wizards* que simplificam tarefa

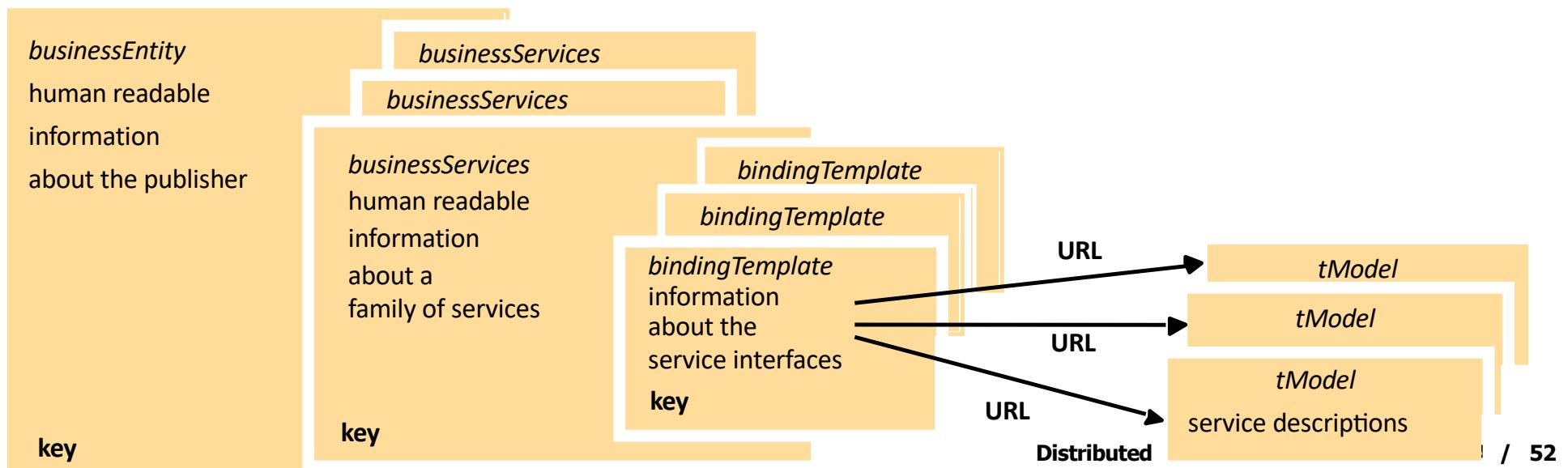
O documento WSDL pode tipicamente ser obtido diretamente a partir de uma instância do serviço em execução.

`http://mywebservice.org/my-web-service?wsdl`

UDDI (UNIVERSAL DESCRIPTION, DISCOVERY AND INTEGRATION)

Diretório de WebServices com a possibilidade de pesquisa com base em atributos.

- Protocolo de ligação ou binding entre o cliente e o servidor
- Os fornecedores dos serviços publicam a respectiva interface
- O protocolo de inspeção permite verificar se uma dado serviço existe baseado na sua identificação
- O UDDI permite encontrar o serviço baseado na sua definição – capability lookup



WEBSERVICES: CLIENTES

O coódigo de ligação ao web service do cliente é tipicamente gerado de forma automática.

Com base neste código, o cliente invoca uma função/método, passando os parâmetros e recebendo eventualmente o resultado do retorno.

A geração deste código tem como base o documento WSDL associado ao web service a invocar.

A geração pode ser estática, em tempo de compilação; Dinâmica, sendo o código gerado em tempo de execução.

Tal é possível porque o WSDL está pensado para ser processado por máquinas e é exaustivo na descrição do serviço.

WEB SERVICES: EXTENSÕES WS-*

Para além do mecanismo de invocação remota de base, existem diversas extensões que oferecem *features* e semânticas mais sofisticadas:

WS-Reliability: especificação que permite introduzir fiabilidade nas invocações remotas

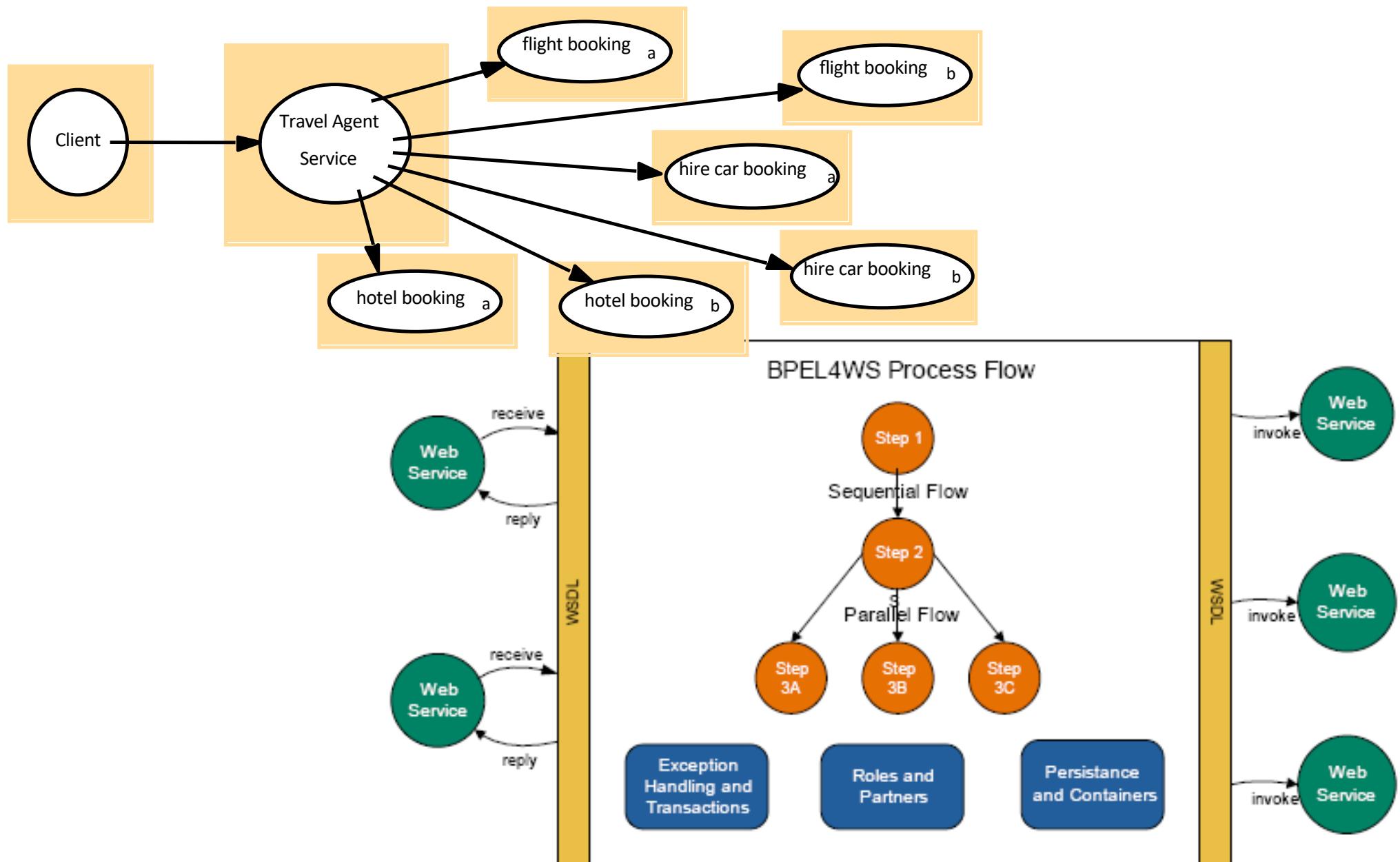
WS-Security: define como efectuar invocações seguras

WS-Coordination: fornece enquadramento para coordenar as ações de aplicações distribuídas (coreografia de web services)

WS-AtomicTransaction: forncece coordenação transaccional (distribuída) entre múltiplos web services

...

WS-COORDINATION



WEB SERVICES: RESUMO

Standard na indústria porque apresenta uma solução para integrar diferentes aplicações

Permite:

- Utilização de standards
 - HTTP, XML
- Reutilização de serviços (arquiteturas orientadas para os serviços)
 - WS-Coordination
- Modificação parcial/evolução independente dos serviços
 - WSDL

WEB SERVICES: PROBLEMAS

Desempenho:

- Complexidade do XML
- Difícil fazer *caching*: noção de operação + estado não permite explorar o *caching* da infraestrutura HTTP

Complexidade

- A normalização gerou especificações complexas que tornam necessário a utilização de ferramentas de suporte.

WEB SERVICES: SUPORTE EM JAVA

Os Web Services são suportados através da extensão Java API for XML Web Services (JAX-WS). Fez parte da distribuição standard até à versão 8 (1.8) da linguagem.

A partir da versão Java 9, passou a ser uma dependência externa.

JAX-WS: SERVIDOR (EXEMPLO)

```
@WebService(serviceName=SoapServers.NAME,
targetNamespace=SoapServers.NAMESPACE,
endpointInterface=SoapServers.INTERFACE)
public class SoapServersWebService implements SoapServer
{
    static final String NAME = "servers";
    static final String NAMESPACE = "http://sd2019";
    static final String INTERFACE =
"server.soap.SoapServers";

    @WebFault
    public class ServersException extends Exception
    {... }
```

JAX-WS: SERVIDOR (EXEMPLO)

@WebService permite indicar que se trata dum servidor de Web Services.

```
@WebService(serviceName=SoapServers.NAME,  
targetNamespace=SoapServers.NAMESPACE,  
endpointInterface=SoapServers.INTERFACE)
```

```
public class SoapServersWebService implements SoapServer  
{  
    static final String NAME = "servers";  
    static final String NAMESPACE = "http://sd2019";  
    static final String INTERFACE =  
"server.soap.SoapServers";
```

```
@WebFault
```

```
public class ServersException extends Exception  
{... }
```

JAX-WS: SERVIDOR (EXEMPLO)

```
@WebService(serviceName=SoapServers.NAME,  
targetNamespace=SoapServers.NAMESPACE,  
endpointInterface=SoapServers.INTERFACE)  
public class SoapServersWebService implements SoapServer  
{  
    static final String NAME = "servers";  
    static final String NAMESPACE = "http://sd2019";  
    static final String INTERFACE =  
"server.soap.SoapServers";
```

@WebFault permite definir uma exceção a ser lançada pelos métodos do servidor

```
@WebFault
```

```
public class ServersException extends Exception  
{...}
```

JAX-WS: SERVIDOR (EXEMPLO)

```
@WebMethod  
void createServer( Server srv) throws ServersException  
{... }  
  
@WebMethod  
Server getServer (String id) throws ServersException  
{... }  
...  
  
public static void main(String[] args) {  
    ...  
    Endpoint.publish(serverURI,  
        new SoapServersWebService());  
    ...  
}  
}
```

JAX-WS: SERVIDOR (EXEMPLO)

@WebMethod especifica que é um método do servidor de Web Services

@WebMethod

```
void createServer( Server srv) throws ServersException  
{... }
```

@WebMethod

```
Server getServer (String id) throws ServersException  
{... }
```

...

```
public static void main(String[] args) {
```

...

```
Endpoint.publish(serverURI,  
    new SoapServersWebService());
```

...

```
}
```

```
}
```

JAX-WS: SERVIDOR (EXEMPLO)

```
@WebMethod  
void createServer( Server srv) throws ServersException  
{... }
```

```
@WebMethod  
Server getServer (String id) throws ServersException  
{... }  
...
```

```
public static void main( ... )
```

Para publicar um servidor de Web Services
usando o suporte nativo do JAX-WS.

```
    Endpoint.publish(serverURI,  
                     new SoapServersWebService());
```

```
    ...  
}
```

JAX-WS: CLIENTE (EXEMPLO)

```
QName QNAME = new QName(SoapServersWebService.NAMESPACE,  
                        SoapServersWebService.NAME );  
  
Service service = Service.create(  
    "http://<ip>:<port>/path?wsdl", QNAME);  
SoapServers servers = service.getPort(  
    servers.soap.SoapServers.class );  
  
Server srv = ...  
  
servers.createServer( srv );  
  
srv = servers.getServer( "someid" );
```

JAX-WS: CLIENTE (EXEMPLO)

```
QName QNAME = new QName(SoapServersWebService.NAMESPACE,  
    SoapServersWebService.NAME );
```

```
Service service = Service.create(
```

"http://<ip>:<port>/SoapServersWebServices?wsdl");

```
    Para criar um cliente de Web Services  
    usando o suporte nativo do JAX-WS.
```

```
SoapServers servers = service.getPort(  
    servers.soap.SoapServers.class );
```

```
Server srv = ...
```

```
servers.createServer( srv );
```

```
srv = servers.getServer( "someid" );
```

JAX-WS: CLIENTE (EXEMPLO)

Obtém uma referência para o servidor por configuração direta.

```
QName QNAME = new QName(SoapServersWebService.NAMESPACE,  
    SoapServersWebService.NAME );
```

```
Service service = Service.create(  
    "http://<ip>:<port>/path?wsdl", QNAME);  
SoapServers servers = service.getPort(  
    servers.soap.SoapServers.class );
```

```
Server srv = ...  
  
servers.createServer( srv );  
  
srv = servers.getServer( "someid" );
```

JAX-WS: CLIENTE (EXEMPLO)

```
QName QNAME = new QName(SoapServersWebService.NAMESPACE,  
    SoapServersWebService.NAME );
```

```
Service service = Service.create(  
    "http://<ip>:<port>/path?wsdl", QNAME);  
SoapServers servers = service.getPort(  
    servers.soap.Servers.class );
```

```
Server srv = ...
```

Com a referência para o servidor, invocar
um método remoto é idêntico a invocar
um método local.

```
servers.createServer( srv );  
  
srv = servers.getServer( "someid" );
```

WEBSERVICES: SOAP vs REST

SOAP:

getServer(id)
addServer(id, srv)
removeServer(id)
updateServer (id, srv)
listServers()
findServer(query)

REST:

<http://myserver.com/server/{id}>
(GET, POST, DELETE, PUT)

<http://myserver.com/server>
<http://myserver.com/server?query=a+b>

REST vs. RPCs/WEB SERVICES

Nos sistemas de RPCs/Web Services a ênfase é nas operações que podem ser invocadas.

Nos sistemas REST, a ênfase é nos recusos, na sua representação e em como estes são afectados por um conjunto de métodos standard.

WEBSERVICES: SOAP vs REST

REST:

- Mais simples
- Mais eficiente
- Complicado implementar serviços complexos (numa abordagem purista)
- Uso generalizado para disponibilização de serviços na Internet

Web services

- Mais complexo
- Grande suporte da indústria para serviços “internos”

E.g.:

- <http://www.oreillynet.com/pub/wlg/3005>

PARA SABER MAIS

G. Coulouris, J. Dollimore and T. Kindberg, Gordon Blair,
Distributed Systems - Concepts and Design, Addison-Wesley,
5th Edition

- Web services – capítulo 9.1-9.4

REST:

http://en.wikipedia.org/wiki/Representational_State_Transfer

SISTEMAS DISTRIBUÍDOS

Capítulo 6

(Introdução ao Teste e) Avaliação de Sistemas
Distribuídos

TESTE E AVALIAÇÃO DE SISTEMAS DISTRIBUÍDOS

Dois objetivos:

- Avaliação de propriedades funcionais
 - O sistema implementa a funcionalidade pretendida?
- Avaliação de propriedades não funcionais
 - “Rapidez”, escalabilidade, etc.

TESTE E AVALIAÇÃO DE SISTEMAS DSITRIBUÍDOS

Dois objetivos:

- **Avaliação de propriedades funcionais**
 - O sistema implementa a funcionalidade pretendida?
- Avaliação de propriedades não funcionais
 - “Rapidez”, tolerância a falhas, etc.

TESTES FUNCIONAIS

Objetivo: verificar que o sistema distribuído efetua as funcionalidades definidas na sua especificação.

Qual a diferença face a sistemas não distribuídos?

É necessário ter em atenção a existência de múltiplos componente independentes, e as falhas que podem surgir na comunicação e nos próprios componentes.

TESTES FUNCIONAIS: PASSO 0

Todo o código desenvolvido deve passar pelos mecanismos normais de teste, independentemente de ser parte dum sistema distribuído: e.g. *unit tests*, etc.

- ***Unit test***: método de teste de software em que os módulos dum programa são testados de forma independente para verificar se estão corretos. Um módulo pode ser uma classe, por exemplo.
- Alguns sistemas para implementar *unit tests* em Java: JUnit, Mockito

JUNIT : EXEMPLO

```
class UserResourceTest {  
    private final User u = new User( "nuno", "Nuno", "nuno@fct", "pwd");  
  
    @Test  
    void simpleInsert() {  
        try {  
            UserResource resource = new UserResource("fct",  
                new URI("http://localhost:8080/rest"));  
            String result = resource.postUser(u);  
            assertEquals("nuno@dummydomain", result);  
  
            User u0 = resource.getUser(u.getName(), u.getPwd());  
            assertEquals(u.getName(), u0.getName());  
            assertEquals(u.getPassword(), u0.getPassword());  
            assertEquals(u.getFullName(), u0.getFullName());  
            assertEquals(u.getEmail(), u0.getEmail());  
            assertEquals(u.toString(), u0.toString());  
        } catch (URISyntaxException e) {  
            fail(e.getMessage());  
        }  
    }  
}
```

TESTES FUNCIONAIS: *TESTES UNITÁRIOS*

Testes unitários de componentes distribuídos: testar a funcionalidade dos componentes distribuídos quando funcionam isoladamente (se possível).

Como fazer: definir um conjunto de cenários que testem as diferentes possibilidades dos inputs, não só para utilizações que terminam em sucesso mas também para utilizações que terminam em erros.

EXEMPLO: TESTES UNITÁRIOS NO TRABALHO PRÁTICO

Objetivo: testar cada componente isoladamente, considerando situações com e sem concorrência.

Não completamente possível devido à forte ligação entre os serviços Users e Diretório.

Cenários definidos – e.g.:

- Cria novo utilizador; verifica que utilizador existe.
- Cria novo utilizador; remove utilizador; verifica que utilizador não existe.
- Cria novo utilizador; altera dados do utilizador; verifica que dados foram atualizados.
- Verifica que devolve erro apropriado ao aceder a utilizador não existente.
- ...

EXEMPLO: TESTES UNITÁRIOS COM CONCORRÊNCIA

Objetivo: testar cada componente isoladamente, considerando situações com concorrência.

Cenários definidos – e.g.:

- Cria múltiplos utilizadores concorrentemente; verifica que todos os utilizadores existem.
- Cria e remove utilizadores concorrentemente; verifica que todos os utilizadores que devem existir existem.
- ...

TESTES FUNCIONAIS: *TESTES DE INTEGRAÇÃO*

Testes de integração de componentes distribuídos: testar a funcionalidade dos componentes distribuídos quando se integram vários componentes, ainda sem considerar a parte da distribuição e as falhas (se possível).

Como fazer: definir um conjunto de cenários que testem as diferentes possibilidades dos inputs que levem à interação dos componentes, não só para utilizações que terminam em sucesso mas também para utilizações que terminam em erros.

EXEMPLO: TESTES DE INTEGRAÇÃO NO TRABALHO PRÁTICO

Objetivo: testar a integração de componentes no sistema, ainda sem considerar a parte da distribuição e as falhas.

No trabalho, os componentes são distribuídos, pelo que se testava a integração de componentes distribuídos sem falhas.

Cenários definidos – e.g.:

- Criar um post; verifica que o post foi criado ou não consoante os parâmetros.
- Criar um post e removê-lo; verifica que o post não existe.
- Remove um utilizador; verifica que os posts/feed desse utilizador foram removidos tentando aceder ao feed/post do utilizador..

EXEMPLO: TESTES DE INTEGRAÇÃO COM CONCORRÊNCIA NO TRABALHO PRÁTICO

Objetivo: testar a integração de componentes no sistema, considerando situações de concorrência mas ainda sem considerar a parte da distribuição e as falhas.

No trabalho, os componentes são distribuídos, pelo que se testava a integração de componentes distribuídos sem falhas.

Cenários definidos – e.g.:

- Concorrentemente cria um post e remove utilizadores.

TESTES FUNCIONAIS: *TESTES FUNCIONAIS*

Testes funcionais de sistemas distribuídos: testar a funcionalidade dos diversos componentes/serviços quando interagem entre si.

Neste momento é necessário considerar o modelo de falhas, testando o comportamento quando não ocorrem falhas e na presença de falhas.

Como fazer: definir um conjunto de cenários que testem as diferentes possibilidades dos inputs que levem à interação dos componentes, não só para utilizações que terminam em sucesso mas também para utilizações que terminam em erros.

Definir um conjunto de cenários que considerem falhas.

EXEMPLO: TESTES FUNCIONAIS NO TRABALHO PRÁTICO

Objetivo: testar o funcionamento do sistemas como um todo, considerando também cenários de falhas.

Cenários definidos – e.g.:

- Criação e acesso a um post quando existe uma falha de comunicação de curta duração entre os servidores.
- Criação e acesso a um post quando existe uma falha de comunicação de longa duração entre os servidores.
- ...

TESTES: ALGUNS DESAFIOS

O resultado de algumas operações não é determinista – e.g. timestamp do post.

- Estes valores não devem ser verificados.

Alguns resultados são não-deterministas devido à ordem dos resultados, mas é importante verificar os valores – e.g. o resultado duma pesquisa, um feed dum utilizador.

- Verificar os valores independentemente da ordem.

TESTES: ALGUNS DESAFIOS (2)

A execução concorrente de múltiplas operações pode levar a múltiplos estados possíveis.

Exemplos: Não é possível prever o valor final quando se executam concurrentemente múltiplas operações de atualização do displayName dum utilizador.

O teste dos resultados deve aceitar como válido qualquer um dos resultados/estados possíveis. Nem sempre é fácil de prever.

TOOLS PARA TESTES DE SISTEMAS DISTRIBUÍDOS

TLA+ [<https://lamport.azurewebsites.net/tla/tla.html>]

- Linguagem para modelar sistemas (especialmente concorrentes e distribuídos)
- Ferramentas para verificar a correção do modelo

Jepsen [<https://jepsen.io/>]

- Biblioteca para criar programas de teste de sistemas distribuídos.

Functional testing

- Existem algumas ferramentas de *load testing* que também podem ser usadas para fazer testes funcionais (mais no fim do capítulo).

TESTER

Programa para testar trabalho prático.

- 11K+ LOC

Arquitetura base:

- Inicia os servidores em “containers” independentes usando o sistema Docker;
- Mantém internamente uma “cópia” do estado do sistema, i.e., para cada servidor/serviço mantém o estado do serviço e disponibiliza as operações.

TESTER: OPERAÇÕES

Para cada operação do sistema, define-se método que executa os seguintes passos:

1. Executa operação no serviço “real”;
 - Usa resultado da execução para lidar com o não determinismo – e.g. identificador do post retornado pelo serviço real é adicionado à representação interna do post: serviço simulado não gera novo identificador.
2. Executa operação no serviço “simulado”;
3. Verifica que resultado é o mesmo (sucesso ou falha).

TESTER: OPERAÇÕES (CONT.)

Como lidar com invocações REST / SOAP ?

1. Estado simulado é igual;
2. Diferente objeto para invocação em REST e SOAP, mas com a mesma interface – execução da operação permanece idêntica, apenas mudando a inicialização;
3. Tratamento de falhas uniformizado – e.g. exceções SOAP transformadas em códigos HTTP REST.

TESTER: OPERAÇÕES (CONT.)

Como lidar com replicação (no serviço de feeds)?

1. Estado simulado é igual e não é necessário manter as várias réplicas – apenas uma;
2. Diferentes objetos para invocação para cada uma das réplicas;
3. Pode-se dirigir pedido a uma réplica específica ou pedir para executar em todas.

TESTER: TESTES COM FALHAS DE REDE

Falhas de rede simuladas usando iptables – programa que permite configurar as regras de filtragem de pacotes no Linux (sistema usado nos *containers*).

Desafios

- Como garantir que falhas levam a exceções nos programas? Tempo de falhas configurado com base nos timeouts definidos nos programas a testar, de forma a ser superior ao tempo de timeout – parâmetro **-timeout**
- Como se tenta inferir se programa está a tratar dos pedidos assincronamente? Se a resposta ao pedido é anterior ao fim da falha.

TESTER: TESTES COM FALHAS DOS SERVIDORES

As falhas dos servidores são simuladas terminando e reiniciando os containers em que os servidores estão a executar.

TESTER: TESTES

Cada teste consiste numa sequência de operações.

As operações podem executar sequencialmente ou concorrentemente.

TESTE E AVALIAÇÃO DE SISTEMAS DISTRIBUÍDOS

Dois objetivos:

- Testes funcionais
- **Avaliação de propriedades não funcionais**

QUE PROPRIEDADE NÃO FUNCIONAIS

Num sistema distribuído, as propriedades não funcionais tipicamente interessantes são:

- **Latência:** tempo para executar um pedido
- **Performance (*throughput*):** número de pedidos que o sistema consegue executar (por unidade de tempo)
- **Escalabilidade:** como o número de pedido que o sistemas consegue tratar aumenta quando o número de servidores aumenta

MUITO IMPORTANTE

A **avaliação** que se faz a um sistema **deve servir para responder a uma pergunta** – não apenas para obter um número ou um gráfico para decorar um relatório ☺ !!!

De seguida apresentam-se exemplos de questões que é comum colocar.

PERGUNTA 1:

Num sistema com um servidor, **qual a latência observada pelos clientes em função da carga do servidor?**

Como testar?

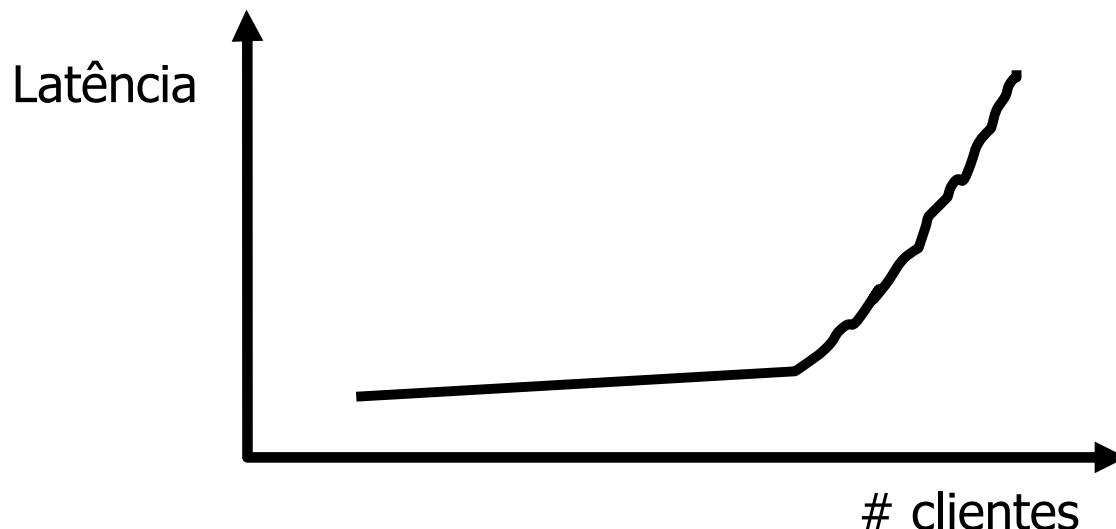
Fazer uma sequência de experiências, aumentando o número de clientes entre cada experiência até ao ponto que a latência começa a aumentar muito.

Repetir cada experiência K vezes (e.g. K = 3).

PERGUNTA 1: REPORTAR RESULTADOS

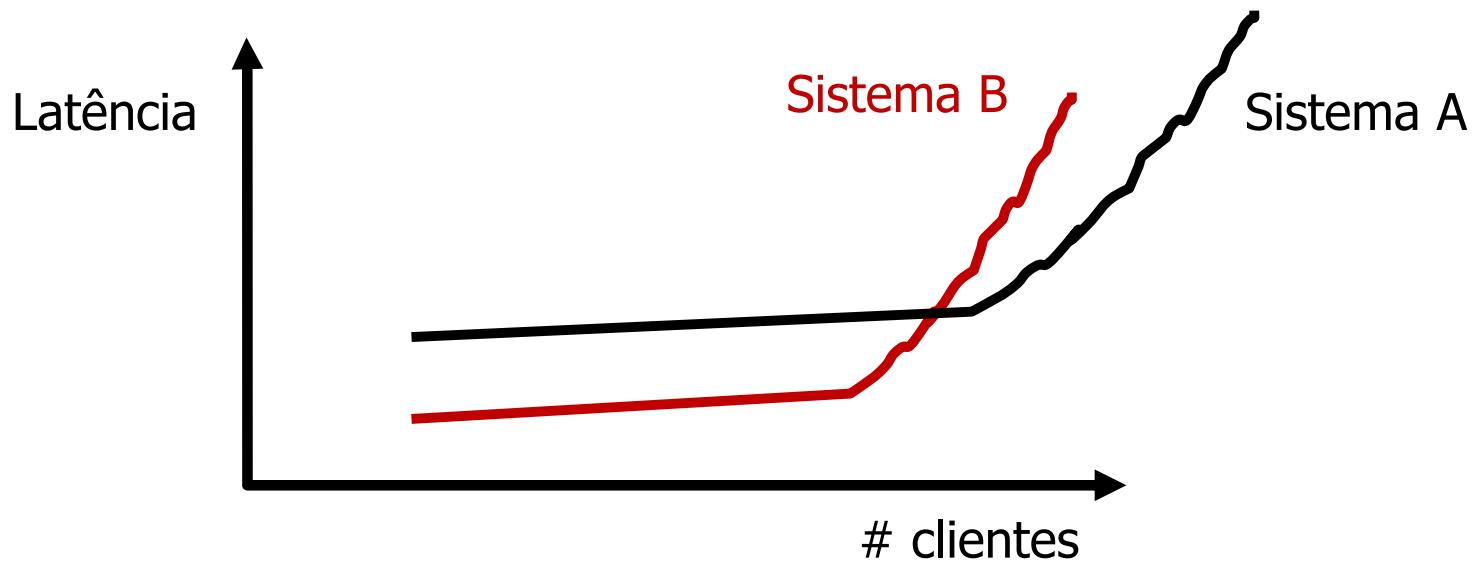
Quais são os resultados esperados quando o servidor executa operações concorrentemente?

- latência permanece constante enquanto o servidor não está em sobrecarga;
- latência aumenta quando o servidor se aproxima da sobrecarga.



PERGUNTA 1: COMPARATIVO

Muitas vezes, o importante na resposta não são os valores absolutos mas a comparação com os valores de outro sistema. Nesse caso, devemos executar as experiências com os dois sistemas e apresentar os resultados. [isto aplica-se também a todas as perguntas seguintes]



PERGUNTA 2:

Num sistema com um servidor (ou N servidores), **qual o número máximo de pedidos que o servidor pode suportar?**

Como testar?

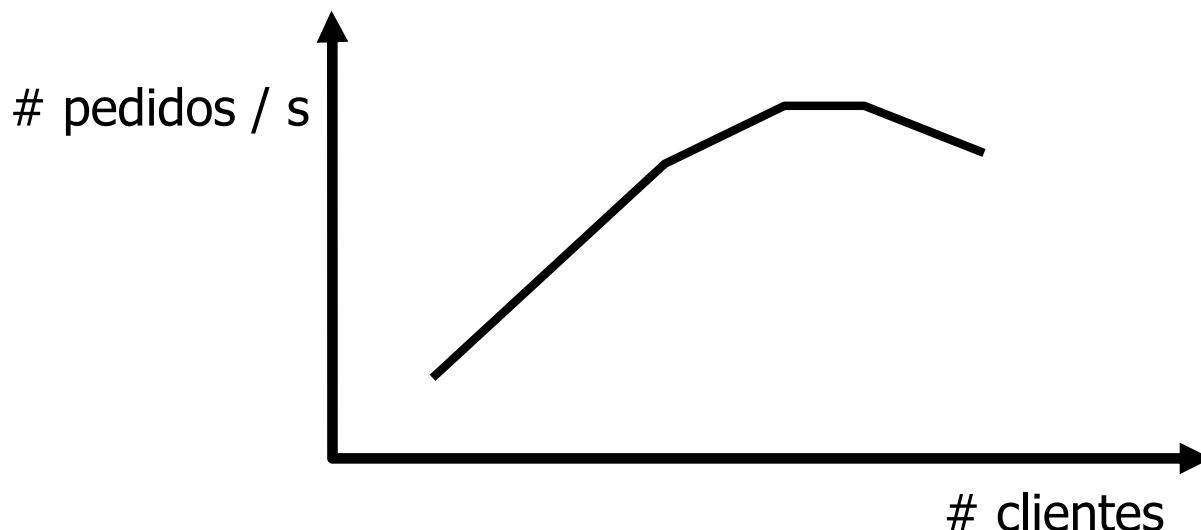
Fazer uma sequência de experiências, aumentando o número de clientes entre cada experiência, até ao ponto em que o número de pedidos deixa de aumentar.

Repetir cada experiência K vezes (e.g. $K = 3$).

PERGUNTA 2: REPORTAR RESULTADOS

Quais são os resultados esperados quando o servidor executa operações concorrentemente?

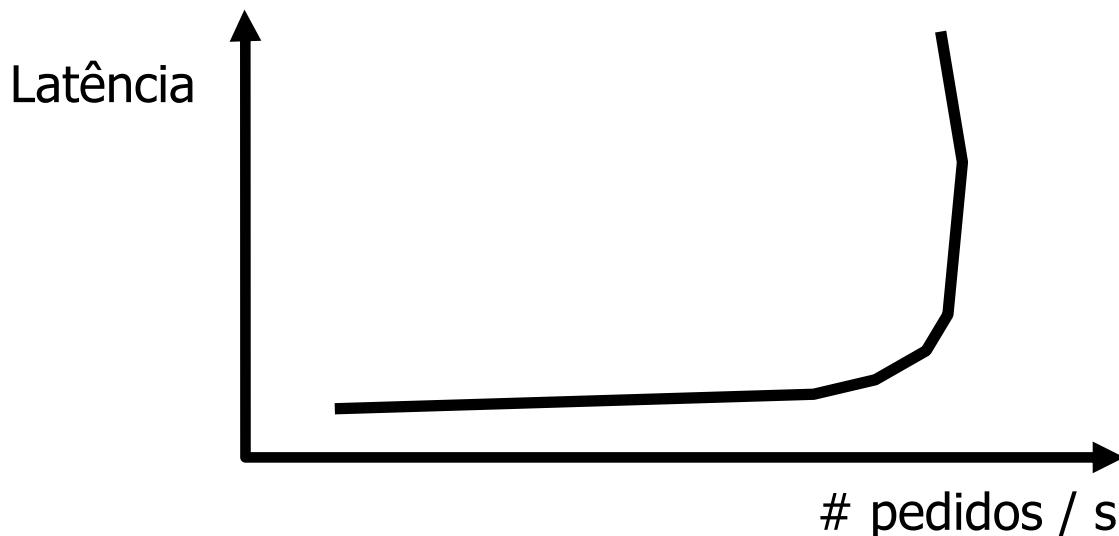
- Número de pedidos aumenta inicialmente linearmente e depois mais lentamente até ao ponto em que deixa de aumentar; após esse ponto é comum o número de pedidos diminuir.



PERGUNTA 2: REPORTAR RESULTADOS

Podemos reportar a latência e o throughput num só gráfico?

Sim: cada ponto corresponderá aos resultados com um dado número de clientes e fazemos uma linha ligando os vários pontos.



PERGUNTA 3:

Como varia a performance/latência quando se aumenta o número de servidores?

Como testar?

Fazer uma sequência de experiências, aumentando o número de clientes entre cada experiência, até ao ponto em que o número de pedidos deixa de aumentar.

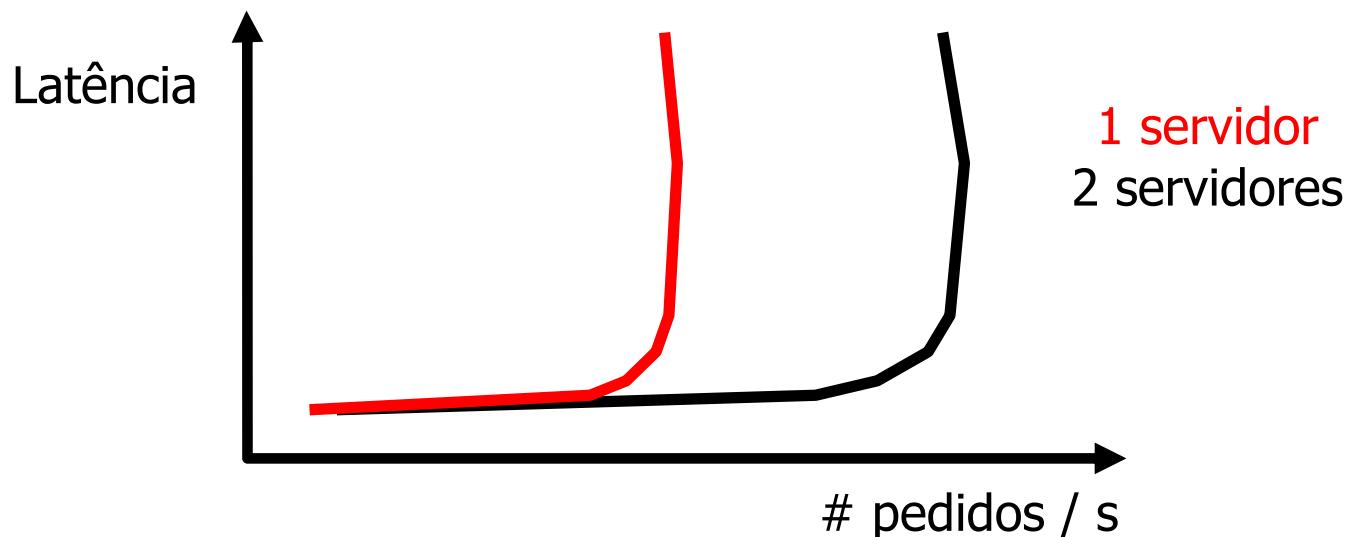
Repetir cada experiência K vezes (e.g. K = 3).

Repetir o passo anterior aumentando o número de servidores entre cada passo.

PERGUNTA 3: REPORTAR RESULTADOS

Quais são os resultados esperados quando os servidores executam operações concorrentemente?

- Latência permanece constante enquanto os servidores não estão em sobrecarga;
- Ponto de sobrecarga é maior com mais servidores.



PERGUNTA 4:

No contexto X (um dos anteriores), qual a latência/throughput das diferentes operações do sistema?

Como varia a latência/throughput do sistema com diferentes workloads?

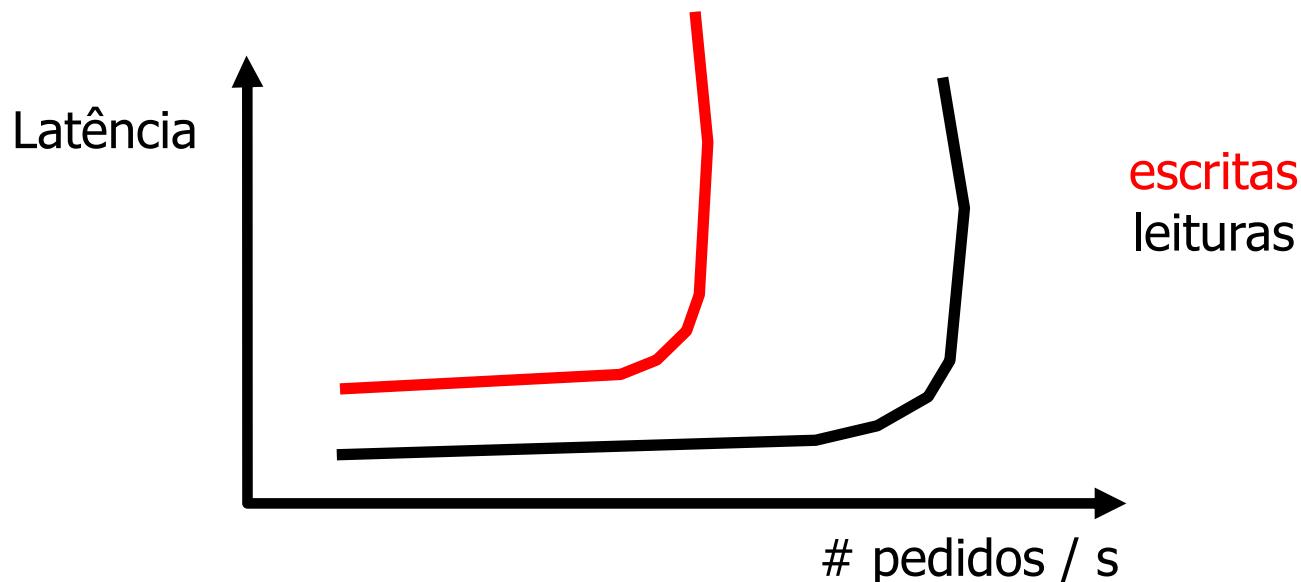
Como testar?

Como nas perguntas anteriores, mas comparando diferentes operações/workloads.

PERGUNTA 4: REPORTAR RESULTADOS

Quais são os resultados esperados?

- A performance de uma operação tende a ser pior quanto mais mensagens e leituras/escritas origina; as escrita são normalmente mais lentas que as leituras.



PERGUNTA 5:

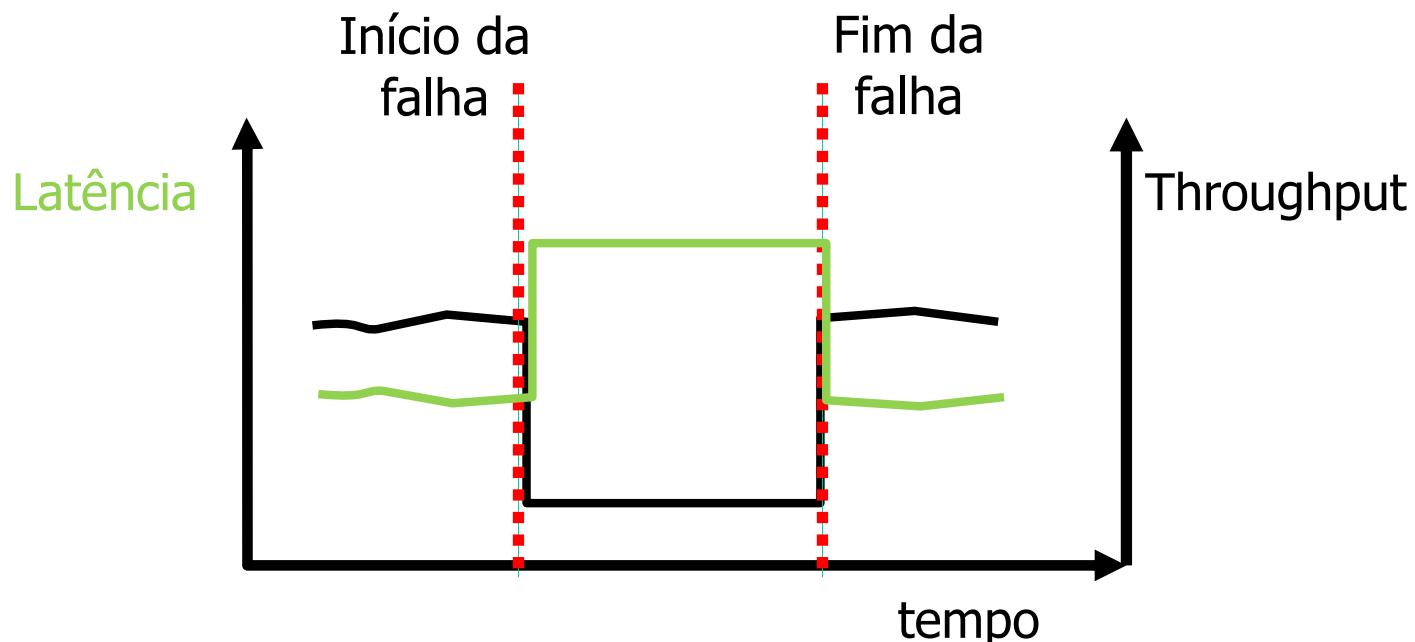
Qual o comportamento do sistema, latência/throughput, na presença de falhas?

Como testar: para um dado setup, simular falhas durante um período limitado da experiência. Obter resultados em janelas de tempo antes, durante e depois das falhas.

PERGUNTA 5: REPORTAR RESULTADOS

Quais são os resultados esperados?

- A latência do sistema aumenta durante o período de falhas e o throughput diminui. Após a falha voltam ao normal, podendo haver alguma oscilação.



TOOLS PARA AVALIAÇÃO DE SISTEMAS DISTRIBUÍDOS

“Load testing” é o teste de sistemas colocando carga nos sistemas e medindo o seu comportamento. Existem muitas ferramentas de “load testing” de sistemas distribuídos.

Exemplos

JMeter [<https://jmeter.apache.org/>]

- Aplicação para “Load testing”. Inicialmente de serviços web, mas atualmente de outros serviços.
- Relativamente complexo de definir testes.

Artillery [<https://artillery.io/>]

- Ferramenta de load testing e functional testing de web services REST.

TOOLS PARA AVALIAÇÃO DE SISTEMAS DISTRIBUÍDOS (2)

As ferramentas de “load testing” permitem muitas vezes fazer “functional testing”, mas o suporte é tipicamente bastante primitivo – quem define o teste tem de definir qual o resultado esperado para cada operação, o que tipicamente não é simples em testes que sejam feitas muitas operações ou operações aleatórias.