SISTEMAS DISTRIBUÍDOS

João Leitão, Sérgio Duarte, Pedro Camponês

(baseado nos slides de Nuno Preguiça)

Aula 7: Capítulo 4

Invocação remota (continuação)

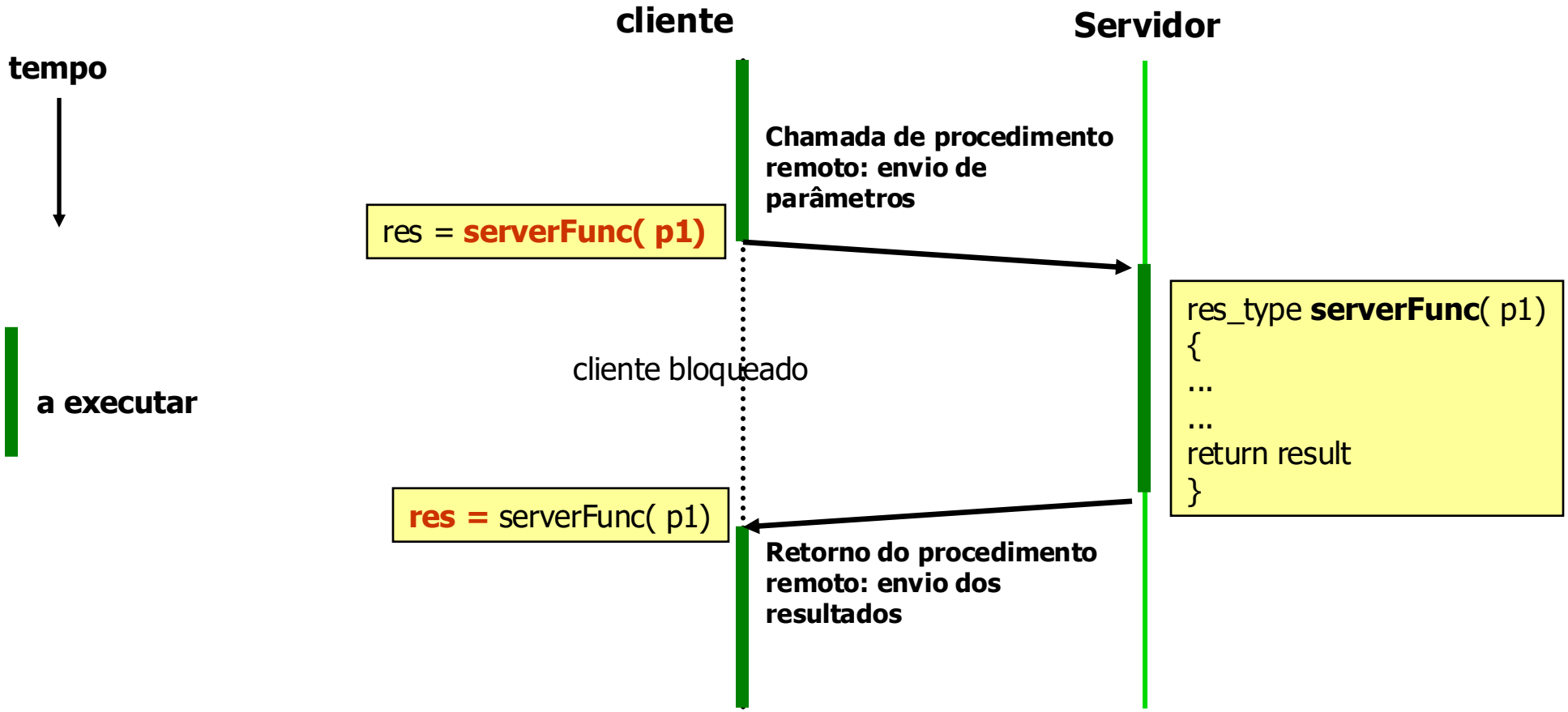
Primeiro Projecto

NA ÚLTIMA AULA

Invocação remota de procedimentos/objectos

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

MODELO DE FALHAS



Admitindo que o canal não introduz falhas arbitrárias, podemos ter:

Canal: falhas de omissão e temporização

Cliente e servidor: crash-failures e falhas de temporização

ANOMALIAS POSSÍVEIS DURANTE UMA INVOCAÇÃO REMOTA

Anomalias a considerar:

- a mensagem com o pedido pode perder-se
- a mensagem com a resposta pode perder-se
- o servidor está muito lento e aparentemente não responde
- o servidor falha e não responde
  - o servidor falha e recupera mais tarde (quando ?)
- o cliente falha e recupera

Algumas destas situações podem ser facilmente detectáveis, outras não.

DIMENSÕES DO PROBLEMA

Protocolo de transporte

Semântica da invocação

PROTOCOLO UDP VS. TCP/HTTTP

Motivações para o uso do UDP:

- Estabelecer uma conexão tem um peso que se pode revelar demasiado pesado (para pedidos pontuais e de pequenas dimensões)
- Resposta à invocação remota funciona como ACK (não é necessário suportar peso dos mecanismos de fiabilidade incluídos no TCP)

Motivações para o uso de TCP (ou HTTP):

- Dimensão dos parâmetros/resultados não influencia complexidade da implementação
- No caso de usar UDP pode ser necessário enviar um pedido/resposta em mais do que uma mensagem

PROTOCOLO TCP / HTTP

O cliente usa conexão TCP (ou HTTP) para contactar o servidor e enquanto não receber resposta não avança com outro pedido

1) O cliente fez um pedido e recebeu a resposta:

- O cliente tem a certeza que a operação executou uma e uma só vez

2) O cliente fez o pedido e não recebeu resposta nenhuma até um certo *timeout*. Decidiu então abandonar o pedido:

- Não se sabe se a operação foi executada ou não.

3) O cliente fez o pedido e recebeu uma notificação de que a conexão foi quebrada (por falha na comunicação ou por *crash* no servidor)

- Não se sabe se a operação foi executada ou não.

PROTOCOLO UDP

O cliente usa o protocolo UDP para contactar o servidor e enquanto não receber resposta a um pedido não avança com outro pedido

1) O cliente enviou uma só vez o pedido e recebeu a resposta
   - O cliente tem a certeza que o procedimento executou (uma e uma só vez… assumindo que não existe duplicação de pacotes)

2) O cliente fez o pedido e não recebeu resposta nenhuma até um certo timeout
   - Não se sabe se a operação foi executada ou não.

DIMENSÕES DO PROBLEMA

Protocolo de transporte

Semântica da invocação

- Maybe
- At Least once
- Operações idempotentes
- At most once
- Exactly once

SEMÂNTICA DA INVOCAÇÃO REMOTA

Maybe (talvez): método pode ter sido executado uma vez ou nenhuma vez

- usa-se quando não se esperam resultados; não tem garantias. O interesse é muito limitado.

Protocolo?

- Envio simples sem retransmissões

clt: s.send(msg)

srv: msg = s.receive()
exec(msg)

SEMÂNTICA DA INVOCAÇÃO REMOTA

At least once (uma ou mais vezes ou "pelo menos uma vez"):

- 1. se cliente recebeu a resposta, o procedimento foi executado uma ou mais vezes
- 2. se o cliente não recebeu a resposta, o procedimento foi executado zero ou mais vezes

Protocolo?

- Implementado por protocolo com re-emissões e sem filtragem de duplicados

```
clt: forever
        s.send(msg)
        try
            [id,res] = s.receive()
            if( id == msg.id)
                // executou 1+ vezes
                break
        catch InterruptException
            continue
```

```
srv: msg = s.receive()
     res = exec(msg)
     s.send( [msg.id,res])
```

EM CASO DE ANOMALIA, APÓS RETRANSMISSÃO, O QUE SE PASSOU?

Caso o cliente não receba resposta ao seu pedido até um timeout após, fez, pelo menos, uma retransmissão, e deseje abandonar, podem ter acontecido duas situações:

- A anomalia ocorreu antes de executar a operação em todas as retransmissões
- A anomalia ocorreu após a execução da operação em uma ou mais retransmissões

Sabe-se que: a operação foi executada zero, uma ou mais vezes.

Caso o cliente receba resposta ao seu pedido após, pelo menos, uma retransmissão, podem ter acontecido duas situações:

- Na tentativa de transmissão anterior, a anomalia ocorreu após a execução da operação
- Na tentativa de transmissão anterior, a anomalia ocorreu antes de executar a operação

Sabe-se que: a operação foi executada uma vez ou mais vezes.

SEMÂNTICA DA INVOCAÇÃO REMOTA

At most once (zero ou uma vez ou "no máximo uma vez"):

- 1. Se cliente recebeu a resposta, o procedimento foi executado uma só vez
- 2. Se o cliente não recebeu a resposta, o procedimento foi executado zero ou uma vezes

Protocolo?

- Protocolo sem re-emissões …

```
clt: s.send(msg)
     try
          [id,res] = s.receive()
          if( id == msg.id)
              // executou 1 vez
     catch InterruptException
          // executou 0 ou 1 vez
```

```
srv: msg = s.receive()
     res = exec(msg)
     s.send( [msg.id,res])
```

SEMÂNTICA DA INVOCAÇÃO REMOTA

At most once (zero ou uma vez ou "no máximo uma vez"):

- 1. Se cliente recebeu a resposta, o procedimento foi executado uma só vez
- 2. Se o cliente não recebeu a resposta, o procedimento foi executado zero ou uma vezes

Protocolo?

- Protocolo sem re-emissões ou protocolo com re-emissões e filtragem de duplicados (código assume: servidor com apenas um thread)

```
clt: forever    ou    for n = 1 to K
          s.send(msg)
          try
              [id,res] = s.receive()
              if( id == msg.id)
                  // executou 1 vez
                  break
          catch InterruptException
              continue
```

```
srv: msg = s.receive()
     if( ! cache.contains( msg.id))
          res = exec(msg)
          cache.put( msg.id, res)
     res = cache.get( msg.id)
     s.send( [msg.id,res])
```

SEMÂNTICA DA INVOCAÇÃO REMOTA

At most once (zero ou uma vez ou "no máximo uma vez"):

- 1. Se cliente recebeu a resposta, o procedimento foi executado uma só vez
- 2. Se o cliente não recebeu a resposta, o procedimento foi executado zero ou uma vezes

Protocolo?

- Protocolo sem re-emissões ou protocolo com re-emissões e filtragem de duplicados (código assume: servidor com apenas um thread). O que acontece quando servidor falha?

```
clt: forever    ou    for n = 1 to K
        s.send(msg)
        try
            [id,res] = s.receive()
            if( id == msg.id)
                // executou 1 vez
                break
        catch InterruptException
            continue
```

```
srv: forever
        msg = s.receive()
        if( ! cache.contains( msg.id))
            res = exec(msg)
            cache.put( msg.id, res)
        res = cache.get( msg.id)
        s.send( [msg.id,res])
```

SEMÂNTICA DA INVOCAÇÃO REMOTA

Exactly once (exactamente uma vez):

- Garante-se a execução exatamente uma vez (a menos que existam falhas permanentes)

Protocolo?

- Protocolo com re-emissões, filtragem de duplicados, estado gravado em memória estável e re-execução quando cliente reinicia (código assume: servidor com apenas um thread)

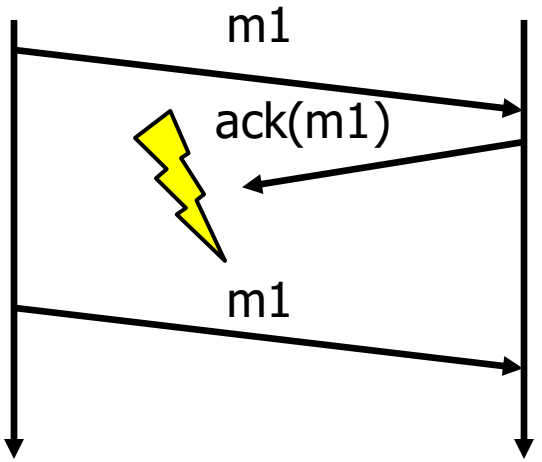
```
clt: log.log( [msg])
     forever
          s.send(msg)
          try
              [id,res] = s.receive()
              if( id == msg.id)
                  log.log( [msg,res])
                  //executou 1 vez
                  break
          catch InterruptException
              continue
```

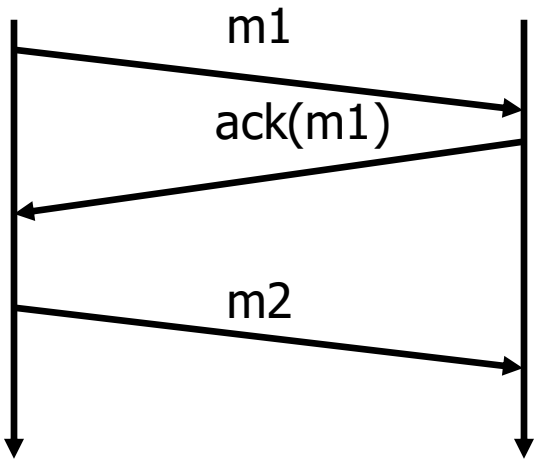
```
srv: msg = s.receive()
     atomic
          if( ! log.contains( msg.id))
               res = exec(msg)
               log.put( msg.id, res)

     res = log.get( msg.id)
     s.send( [msg.id,res])
```

FILTRAGEM DE DUPLICADOS (1)

Problema: Como é que se identifica que uma mensagem recebida é um duplicado?

As mensagens devem ter um identificador. Quem recebe a mensagem guarda o identificador.





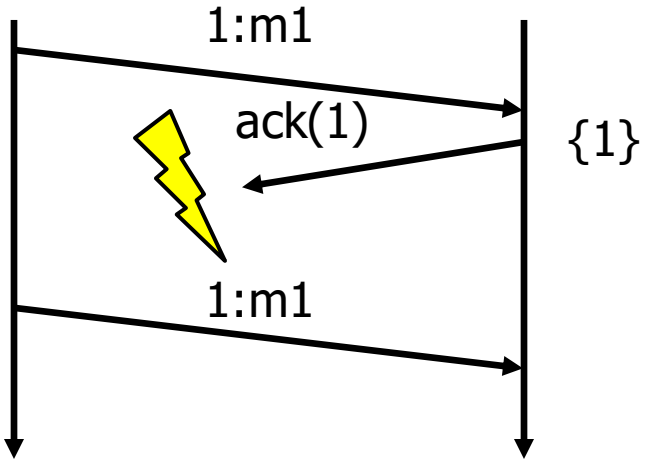
Para o servidor estas situações são indistinguíveis, se m1 == m2.
Como resolver?

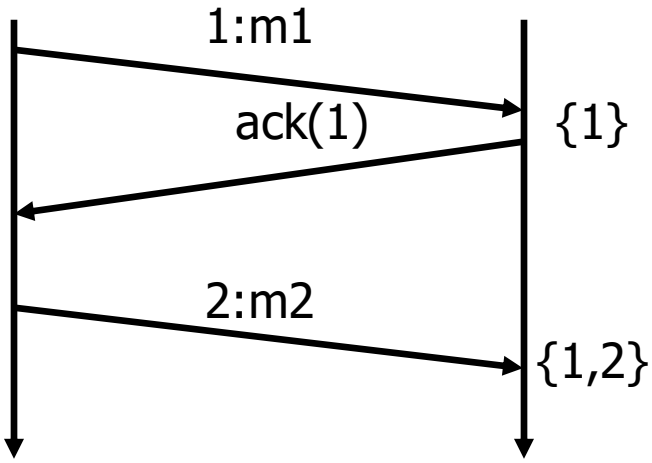
FILTRAGEM DE DUPLICADOS (2)

Problema: É necessário guardar todos os identificadores?

Se os identificadores forem sequenciais basta guardar o último (de cada emissor). Porquê?

Se o emissor não voltar a enviar uma mensagem, pode-se remover a informação após um período alargado, ao fim do qual se saiba que a mensagem não pode ser enviada novamente.





OPERAÇÕES IDEMPOTENTES

Uma operação é **idempotente** se a sua execução repetida não altera o efeito produzido (deixando o servidor no mesmo estado ou num estado aplicacionalmente aceitável como equivalente e produzindo o mesmo resultado).

Exemplos de operações idempotentes:

- em geral todas as operações que não mudam o estado
- reescrever os primeiros 512 bytes de um ficheiro se se ignorar o problema da concorrência de acessos ao ficheiro

Exemplos de operações não idempotentes:

- acrescentar 512 bytes a um ficheiro
- transferir dinheiro entre contas

As operações idempotentes podem ser usadas com um protocolo de invocação remota que faça re-emissões sem filtrar duplicados.

Pode ser usado imediatamente com semântica ”pelo menos uma vez”

OPERAÇÕES IDEMPOT

Nota: Podem existir operações que deixem o servidor no mesmo estado e não sejam idempotentes.
Exemplo ???
Exemplo: uma operação de criação de um ficheiro que devolva como resultado um booleano que indique se o ficheiro foi criado ou não (caso já existisse).

Uma operação é **idempo**
efeito produzido (deixand
aplicacionalmente aceitável como equivalente e produzindo o mesmo resultado).

Exemplos de operações idempotentes:

- em geral todas as operações que não mudam o estado
- reescrever os primeiros 512 bytes de um ficheiro se se ignorar o problema da concorrência de acessos ao ficheiro

Exemplos de operações não idempotentes:

- acrescentar 512 bytes a um ficheiro
- transferir dinheiro entre contas

As operações idempotentes podem ser usadas com um protocolo de invocação remota que faça re-emissões sem filtrar duplicados.

Pode ser usado imediatamente com semântica ”pelo menos uma vez”

SOLUÇÕES GERALMENTE ADOPTADAS

Java RMI

- "At most once" sobre TCP

.NET Remoting

- "At most once" sobre TCP, HTTP, pipes

Web Services

- "At most once" sobre HTTP (SMTP, etc.)
- WS-Reliability: suporte para "at least once", "exactly-once"

REST

- "At most once" sobre HTTP

AGENDA

Invocação remota de procedimentos/objectos

- Motivação
- Modelo
- Concorrência no 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**

SISTEMAS DE OBJETOS DISTRIBUÍDOS

Extensão do modelo de uma aplicação composta por múltiplos objetos para um ambiente distribuído, em que os objetos executam em diferentes máquinas

Problemas adicionais:

- Garbage collection
- Carregamento dinâmico de código

GARGABE-COLLECTION: PROBLEMA

Numa aplicação distribuída composta por objetos a executar em diferentes máquina, como saber que um objeto já não é necessário?

Usar abordagem normal de **garbage collection**:

> se **nenhuma** referência para o objeto **existir**, este pode ser **removido**

Problema: como é que se sabe se existem referências ou não?

GARBAGE-COLLECTION: JAVA RMI

Entre máquinas virtuais, o sistema Java executa um algoritmo de *garbage-collection* distribuído para remover (apagar) objetos remotos para os quais não existem referências

- Cada máquina virtual (VM) contabiliza as referências que existem para cada objecto remoto e informa a VM do objecto
  - Quando aparece a primeira referência, a VM do objecto remoto é informada
  - Quando é removida a última referência, a VM do objecto remoto é informada
  - A VM reenvia a informação periodicamente
- Um objecto remoto pode ser removido (apagado) quando não existem referências em nenhuma VM

QUESTÃO DUM EXAME

Como sabe, o sistema Java RMI usa um mecanismo de garbage collection distribuído para recolher (remover) os objectos remotos que já não são necessários.

Explique o que acontece a um objecto remoto no caso de existir uma falha temporária da rede que impeça a comunicação entre a máquina virtual em que executa um objecto remoto e a máquina virtual do único programa que mantém uma referência para esse objecto remoto.

QUESTÃO DUM EXAME

Como sabe, o sistema Java RMI usa um mecanismo de garbage collection distribuído para recolher (remover) os objectos remotos que já não são necessários.

Explique o que acontece a um objecto remoto no caso de existir uma falha temporária da rede que impeça a comunicação entre a máquina virtual em que executa um objecto remoto e a máquina virtual do único programa que mantém uma referência para esse objecto remoto.

O Java RMI usa um mecanismo baseado em leases, e a lease mantida pelo cliente tem de ser periódicamente renovada (antes que essa lease expire).

Se a lease expirar, a referência remota será esquecida e como tal o objeto remoto pode ser garbage collected na máquina que o executa.

CARREGAMENTO DE CÓDIGO: PROBLEMA

Numa aplicação com objetos, dado um método

```
void function( T t)
```

é possível invocar o método com qualquer objeto cujo tipo estenda T.

Num sistema distribuído, se considerarmos um método disponível remotamente, seria possível passar um objeto dum tipo cujo código não é conhecido no servidor.

Solução: impedir esta situação ou carregar o código dinamicamente.

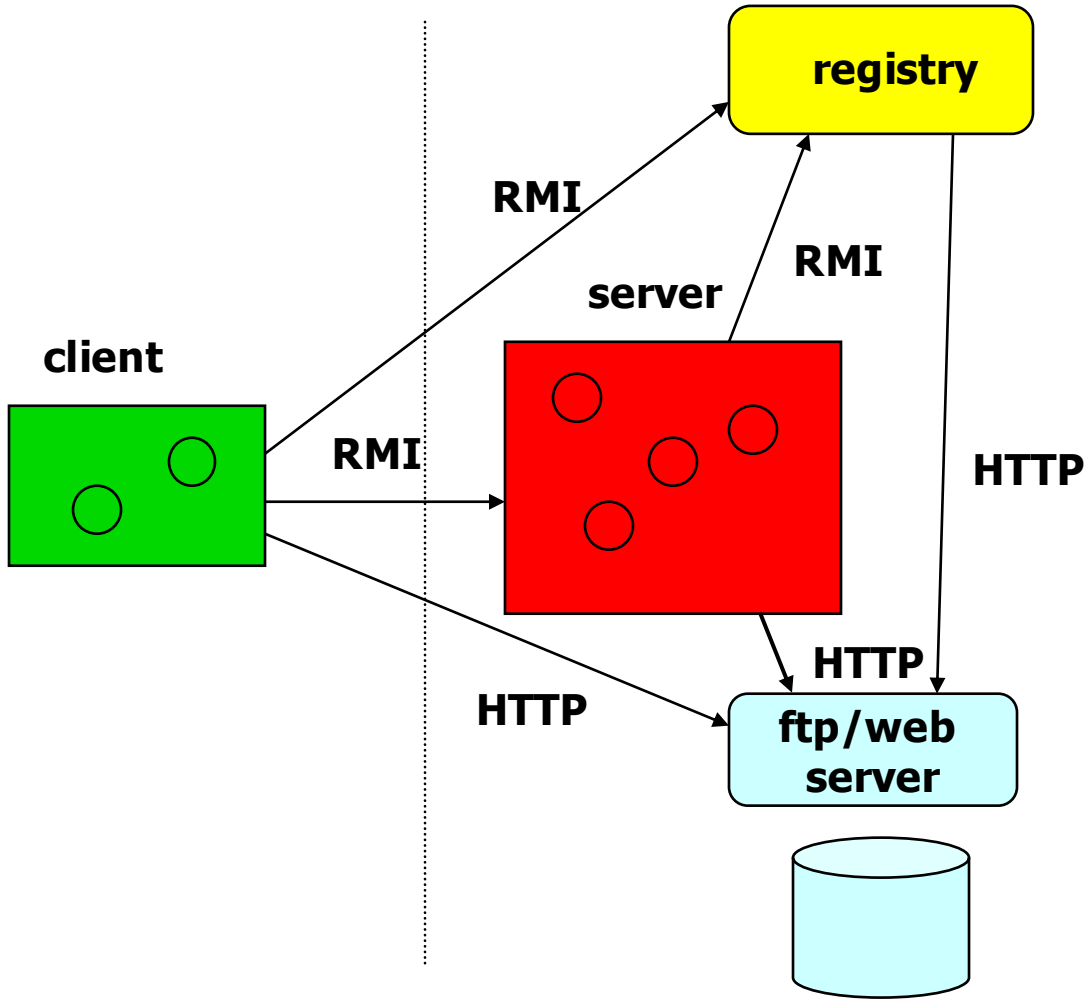
CARREGAMENTO DE CLASSES: SOLUÇÃO JAVA

RMI carrega o código das classes se o mesmo não estiver disponível

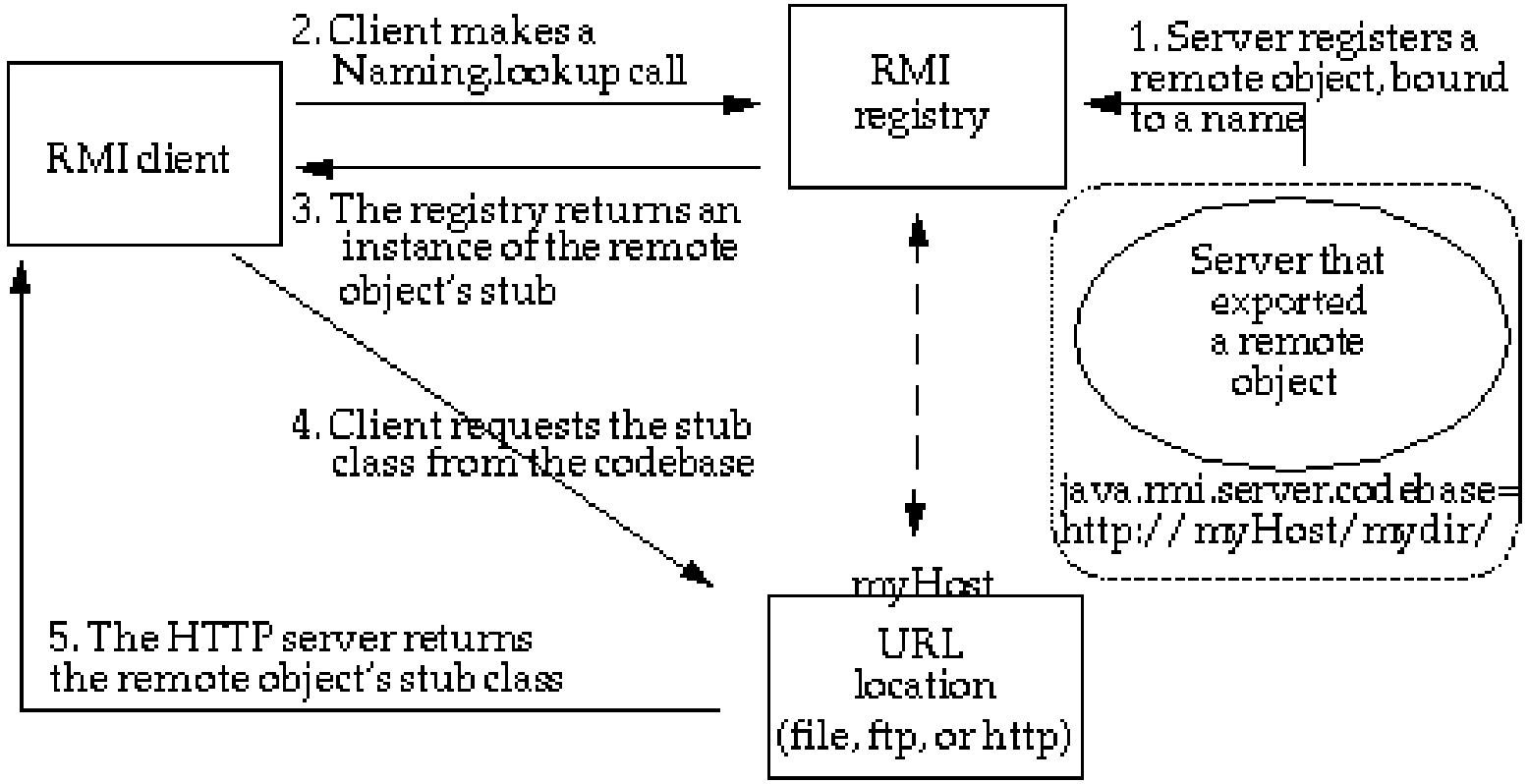
Isto aplica-se às classes do objecto remoto, do *stub*, do esqueleto, dos parâmetros e do valor de retorno dos métodos

O carregamento faz-se remotamente se necessário

Os URLs das classes ficam anotados nas referências remotas para se poderem localizar as mesmas

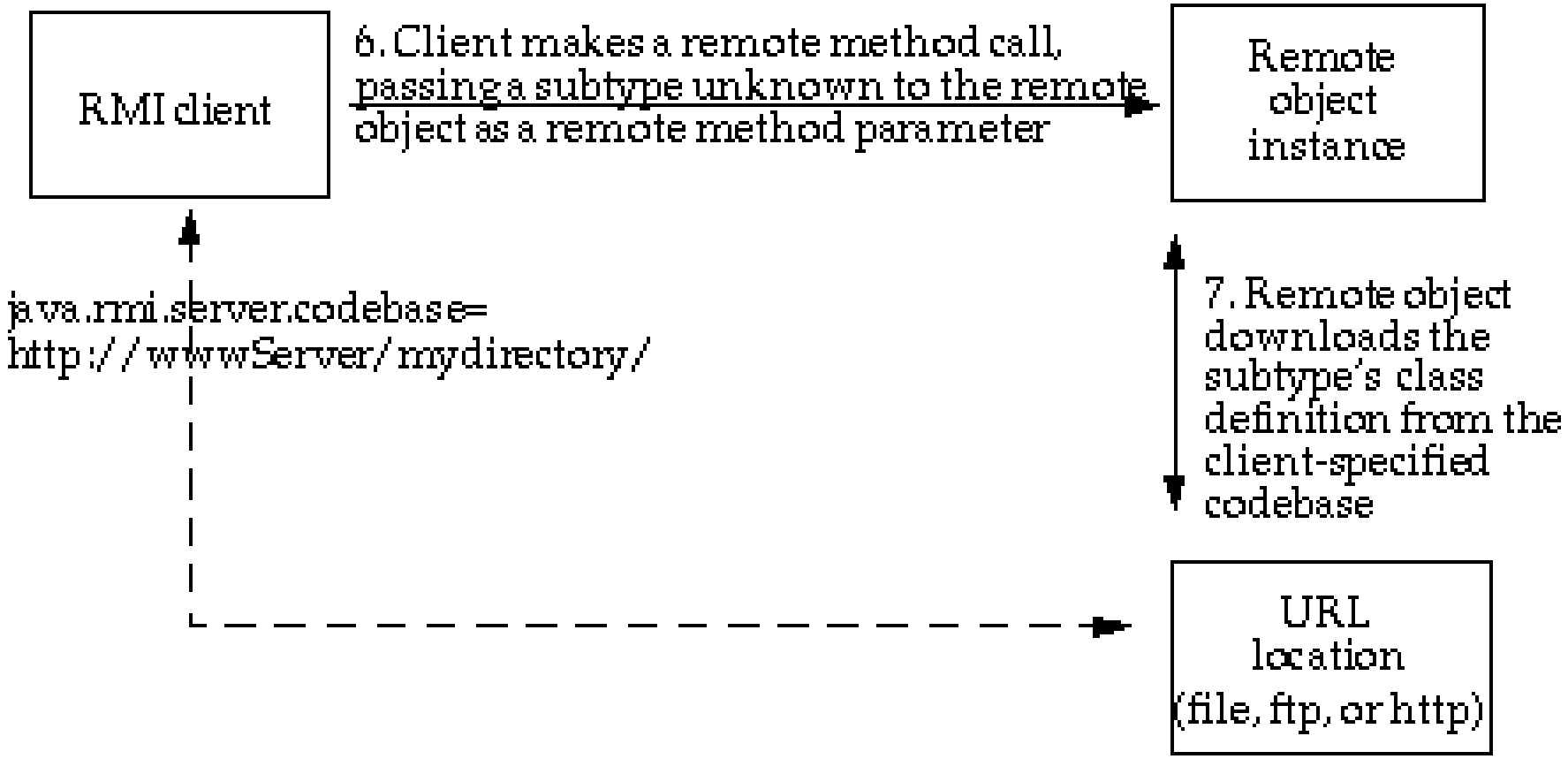


CARREGAMENTO DE CÓDIGO NO CLIENTE



Os objetos anotam o seu ***codebase***, o que permite obter o código das classes remotamente sem configuração adicional.

CARREGAMENTO DE CÓDIGO NO SERVIDOR



SEGURANÇA DO CARREGAMENTO DO CÓDIGO

Carregar código de fontes desconhecidas é um potencial problema de segurança

O RMI usa os mecanismos de segurança da linguagem Java

Qualquer código a carregar tem de o ser através de um carregador de classes (*class loader*)

Um gestor de segurança (*security manager*) tem de estar presente para que o carregamento de código seja possível

As aplicações podem fornecer o seu próprio gestor de segurança

PARA SABER MAIS

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

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

PRIMEIRO PROJECTO

Neste projeto vão desenvolver um Sistema distribuído composto por vários serviços que pretende modelar uma versão simplicada (e diferente) do Sistema Reddit!

REDDIT

O Reddit é uma Plataforma online que permite a utilizadores registados no Sistema adicionarem contéudo sobre a forma de *posts* textuais ou outros contéudos multimedia a que outros utiliadores podem reagir com comentários e fazer *upvote* e *downvote*.

Na nossa versão do Reddit temos uma visão simplicada visto que não existem sub-reddits (todo o contéudo aparece numa única *thread* de discussão)

ARQUITETURA DO SISTEMA

Users Service

Manipula entidades do tipo **User**

Operações:

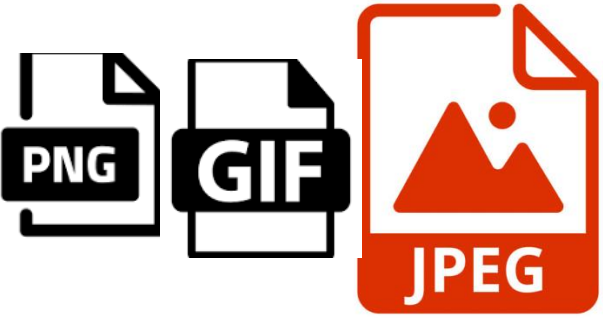
- Criar utilizador
- Obter informação de um utilizador
- Atualizar um utilizador
- Apagar um utilizador
- Procurar um utilizador

Notas relevantes:

- Tem a sua propria base de dados a que os outros serviços não têm acesso.
- A operação de apagar um utilizador têm implicações nos restantes serviços.

ARQUITETURA DO SISTEMA

Image Service



Manipula **imagens (e a sua meta informação)**

Operações:

- Criar uma imagem (associada a um utilizador)
- Obter uma imagem
- Apagar uma imagem (apenas o dono da imagem pode apagar)

Notas relevantes:

- As imagens devem ser persistidas (mas talvez não numa base de dados).
- É preciso de alguma forma manter informação relativa aos donos de cada imagem.

ARQUITETURA DO SISTEMA

Content Service

Manipula entidades do tipo **Post**

Notas relevantes:

- Posts quando são criados podem ou não estar associado a outro post (dependendo se é um post de topo ou uma resposta a um post ou outra reposta)

Operações:

- Criar um post.
- Obter um post.
- Listar posts (com potenciais ordens diferentes).
- Listar posts que respondem a um post especifico.
- Apagar um post.
- Adicionar/Remover upvotes/downvotes
- Consultar upvotes/downvotes

Arquitetura do Sistema

Content Service

Manipula entidades do tipo **Post**

Notas relevantes:

- Upvotes e Downvotes não podem ser repetidos (por utilizador) pelo que um Contador apenas não consegue lidar com esta semântica.

Operações:

- Criar um post.
- Obter um post.
- Listar posts (com potenciais ordens diferentes).
- Listar posts que respondem a um post especifico.
- Apagar um post.
- Adicionar/Remover upvotes/downvotes
- Consultar upvotes/downvotes

Arquitetura do Sistema

Content Service

Manipula entidades do tipo **Post**

Notas relevantes:

- A edição de um post (por um utilizador) nunca deve manipular certos atributos do post: e.g., postId, authorId, creationTimestamp, parentUrl, upVote, downVote.

Operações:

- Criar um post.
- Obter um post.
- Listar posts (com potenciais ordens diferentes).
- Listar posts que respondem a um post especifico.
- Apagar um post.
- Adicionar/Remover upvotes/downvotes
- Consultar upvotes/downvotes

Arquitetura do Sistema

Content Service

Manipula entidades do tipo **Post**

Notas relevantes:

- Um post não pode ser editado (nem pelo seu autor) quando alguma outra entidade o referencia (outro post, upVote ou downVote)

Operações:

- Criar um post.
- Obter um post.
- Listar posts (com potenciais ordens diferentes).
- Listar posts que respondem a um post especifico.
- Apagar um post.
- Adicionar/Remover upvotes/downvotes
- Consultar upvotes/downvotes

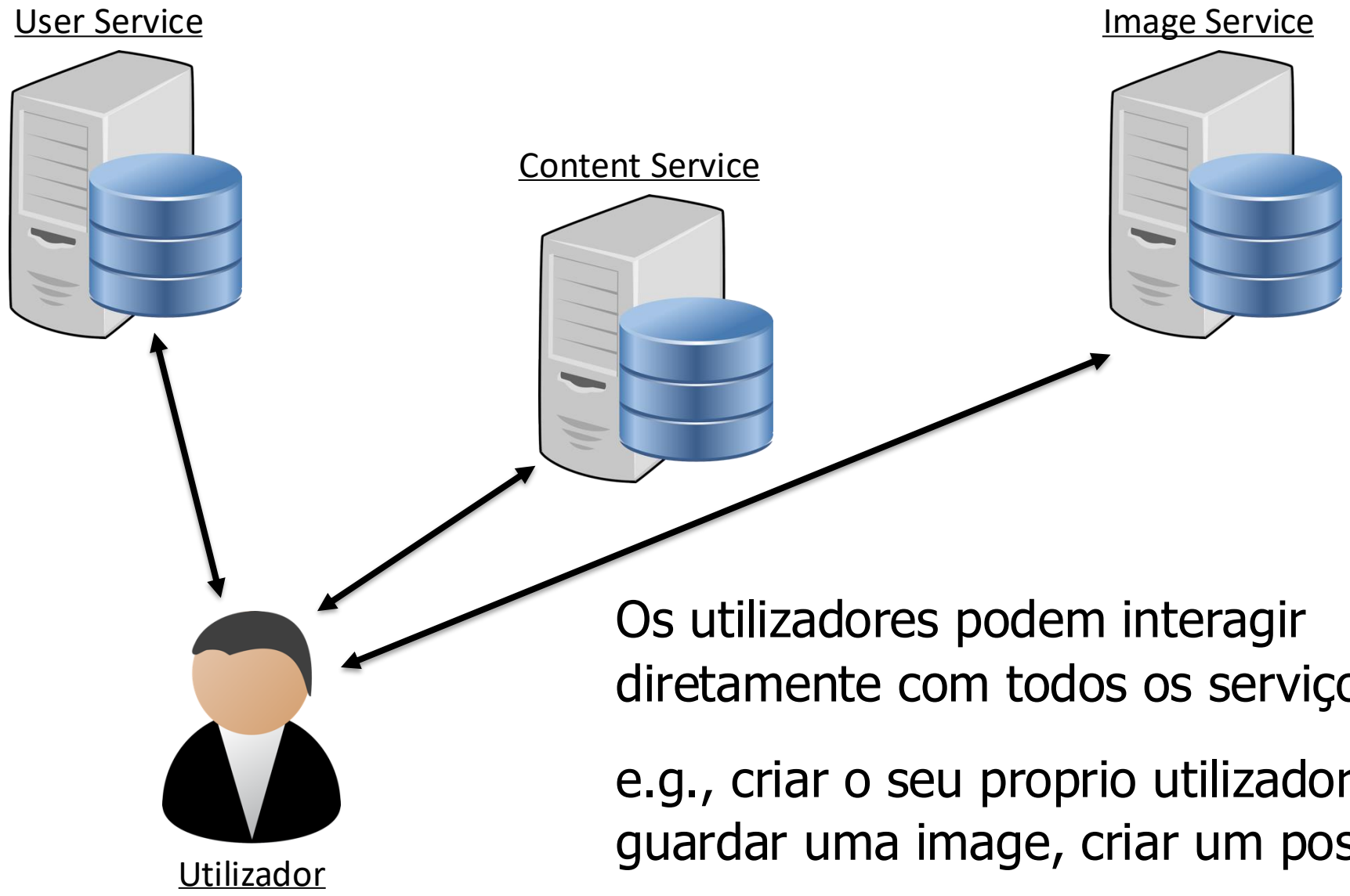
ARQUITETURA DO SISTEMA

User Service

Content Service

Image Service

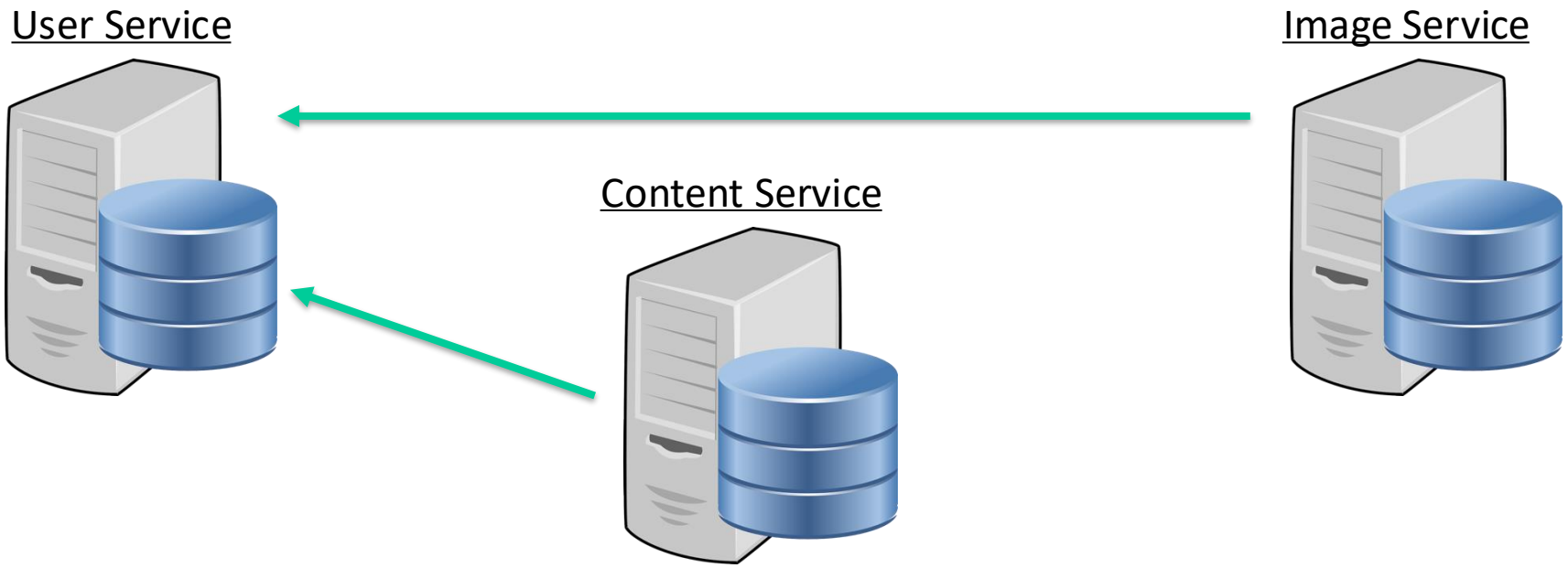
Interações no Sistema: Clientes



Os utilizadores podem interagir diretamente com todos os serviços:

e.g., criar o seu proprio utilizador, guardar uma image, criar um post.

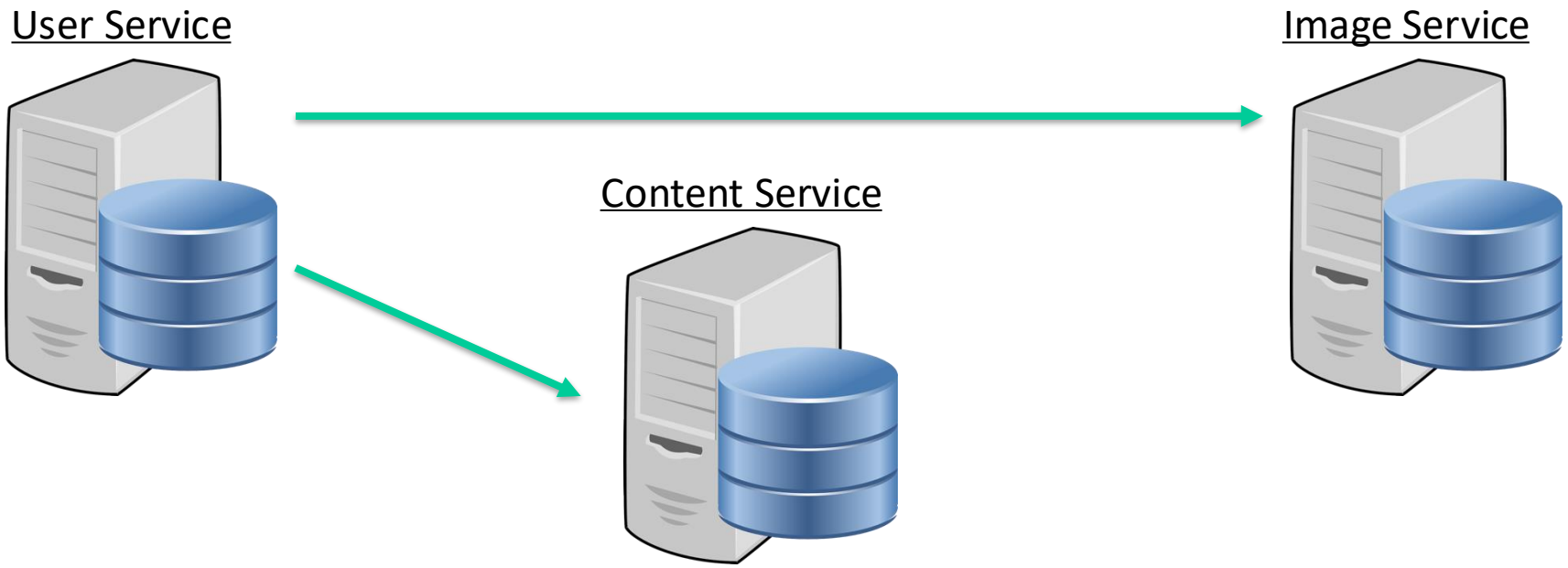
Interações no Sistema: Entre serviços



Os servidores também vão ter de interagir diretamente entre si:

O Image Service e o Content Service vão, por exemplo, ter de autenticar utilizadores, sendo o Users Service o único que pode suportar essa operação.

Interações no Sistema: Entre serviços

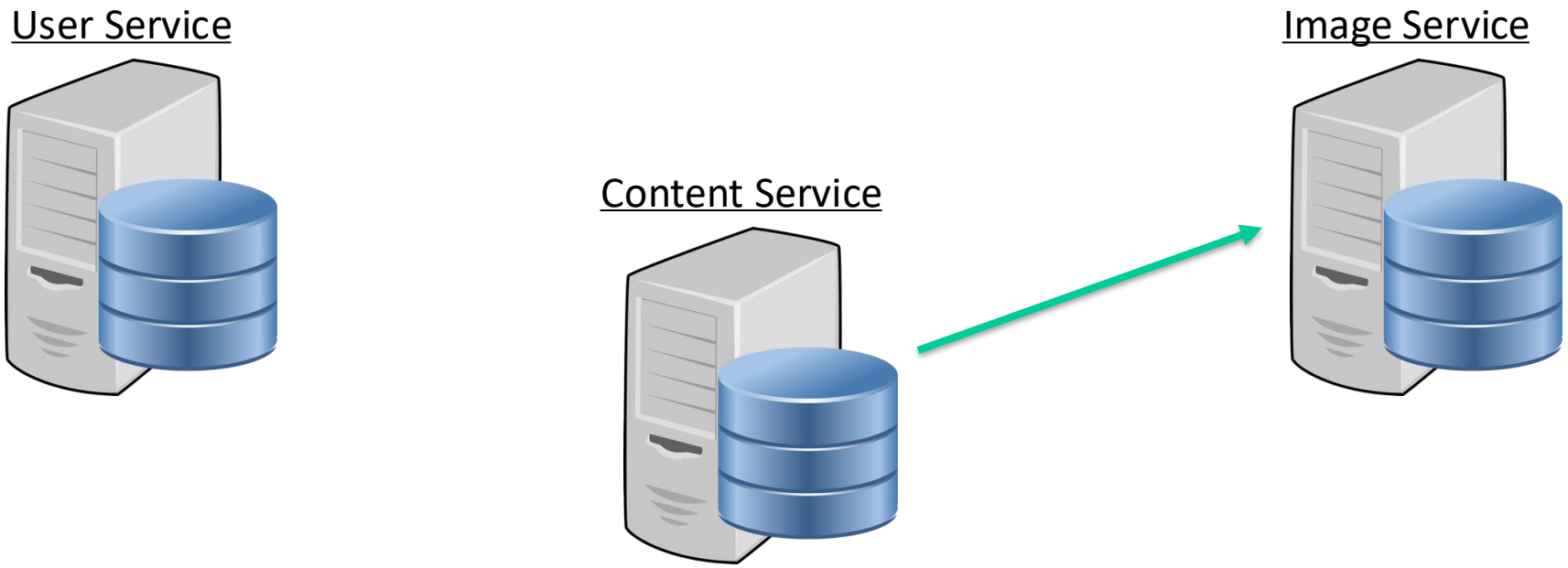


Os servidores também vão ter de interagir diretamente entre si:

O User Service vai ter de interagir com o Content Service e o Image Service quando um utilizadore é apagado:

- Os posts desse utilizador continuam a existir mas deixam de estar associados a um utilizador.
- Os upvotes/downvotes desse utilizador devem deixar de existir.
- Se o utilizador tinha uma imagem como avatar, essa imagem deve ser apagada.

INTERAÇÕES NO SISTEMA: ENTRE SERVIÇOS



Os servidores também vão ter de interagir diretamente entre si:

O Content Service deve interagir com o Image Service:

- Quando um post que refere um conteudo multimedia no Image Service é apagado (de forma a apagar essa imagem também).

Interações Com Base de Dados

- A API que vos foi fornecida do Hibernate executa todas as operações individualmente numa transação.
- Isso pode não ser util visto que permite accessos concorrentes disruptivos.
- E.g.; duas operações de edição de um User concorrentes que modificam o mesmo objeto (mas campos diferentes)
- Op1-Read, Op2-Read, Op1-Update(password), Op2-Update(e-mail).
- N…rder.
- P…s de u…

Para conseguirem isto vão ter de adicionar novas operações à classe de suporte do Hibernate, nomeadamente operações para começar uma transação (e obter a referência para a mesma), ler, modificar e apagar entidades no contexto dessa transação, e fazer commit a uma transação.

REGRAS

- Interfaces REST/gRPC fornecidas no Código base no clip.
- Não podem modificar as Interfaces excepto para:
  - Adicionar parametros opcionias a funções.
  - Adicionar novos endpoints para suportar funcionalidades pedidas.
- Podem utilizar **e adaptar** todo o código que foi fornecido ao longo das aulas práticas.
- Os diferentes serviços vão ser executados em containers distintos, pelo que não podem assumir, por exemplo, que a lógica do serviço Content pode consultar diretamente a base de dados do serviço de Users.

REQUISITOS MÍNIMOS (MAX 9 VALORES)

- API REST: Existem implementações corretas de todos os serviços em REST (com excepção de operações marcadas como opcionais).

- A auto-configuração (descoberta de servidores) funciona e permite a qualquer servidor obter a informação necessária para interagir com os restantes serviços.

- Para os requisitos mínimos não têm de lidar com falhas transientes de comunicação.

Requisitos Base (Max 15 Valores)

- Todos os requisitos mínimo e:
- Conseguem lidar com falhas transientes de comunicação.
- Todos os serviços implementados com a API REST conseguem interagir corretamente com os restantes serviços com API REST.
- Todos os serviços implementados corretamente utilizando uma API gRPC.
- Todos os serviços implementados com a API gRPC conseguem interagir corretamente com os restantes serviços com API gRPC.

ELEMANTOS VALORATIVOS (MAX 20 VALORES)

Todos os requisitos base e (não necessáriamente todos):

- Operações que contabilizam o numero de upVotes e downVotes no serviço de Content (API REST e gRPC) funcionam corretamente e eficientemente. (1 valor)
- Solução suporta transparentemente qualquer combinação de serviços com interfaces gRPC e REST (2 valores)
- Operação que permite, no serviço Content) a consulta dos identificadores de post (i.e., sem parent) por ordens diferentes (decrescente upVote e decrescente por Respostas agregadas) funciona corretamente (API REST e gRPC) (2 valores)
- Servidores de Content (REST e gRPC) disponibilizam uma operação de leitura bloqueante que espera pr novas respostas para uma entrada de topo específica (3 valores).

ASPETOS CONSIDERADOS NA AVALIAÇÃO:

Para além da funcionalidade descrita anteriormente na avaliação vão ser considerados os seguintes aspetos:

- Repetição de Código desnecessário, má estruturação do cógido, uso incorreto de estruturas de dados, constantes mágicas. (Penalização máxima de 2 valores).

- Falta de robustez, comportamento errático, ou soluções profundamente ineficientes (Penalização variável de acordo com a gravidade).

TESTER:

Brevemente vão ter acesso a uma primeira versão do tester:

- Executa bateria de testes que são usadas para averiguar a correta funcionalidade da vossa implementação.
- A bateria de testes depende de garantirem que não há mudanças nas APIs disponibilizadas que as tornem incompatíveis com os pedidos de cliente realizados pelo Tester.
- O tester depende de um ficheiro "fctreddit.propos" corretamente preenchido.
- Existem scripts no Código fornecido para executar o tester (Vão ser atualizados quando a primeira versão for publicada).

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

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 doutroamento

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

ROY FIELDING

UNIVERSITY OF CALIFORNIA,
IRVINE

Architectural Styles and the Design of Network-based Software Architectures

DISSERTATION

submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in Information and Computer Science

by

Roy Thomas Fielding

Dissertation Committee:
Professor Richard N. Taylor, Chair
Professor Mark S. Ackerman
Professor David S. Rosenblum

2000

- 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 adequado.

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 exisitir;
- **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-tipificados, 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**: <empty>

- **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
- Identifcação 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 nivel, 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)

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

```
@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(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)
   {...}
```

@POST / @GET / @PUT / @DELETE
Operação HTTP que leva à execução do método

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)
   {...}
```

@Producesusado 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_J
   public Response addServer(Server serve

   @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

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

@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: Instanciação da Lógica do Servidor

Intância 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);`

Intância única

- `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() )
el
```

"***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**

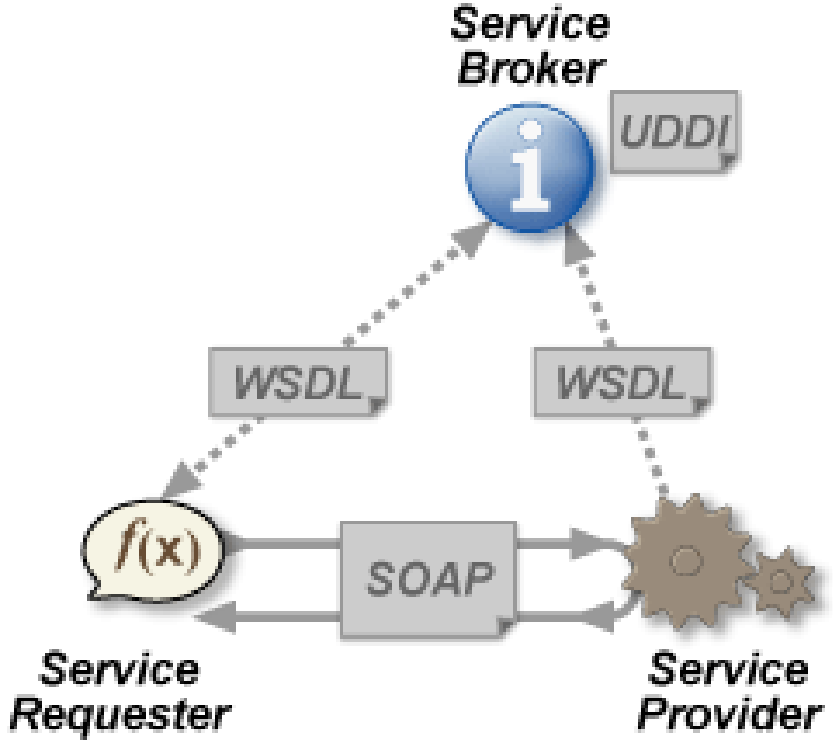
COMPONENTES BÁSICOS

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

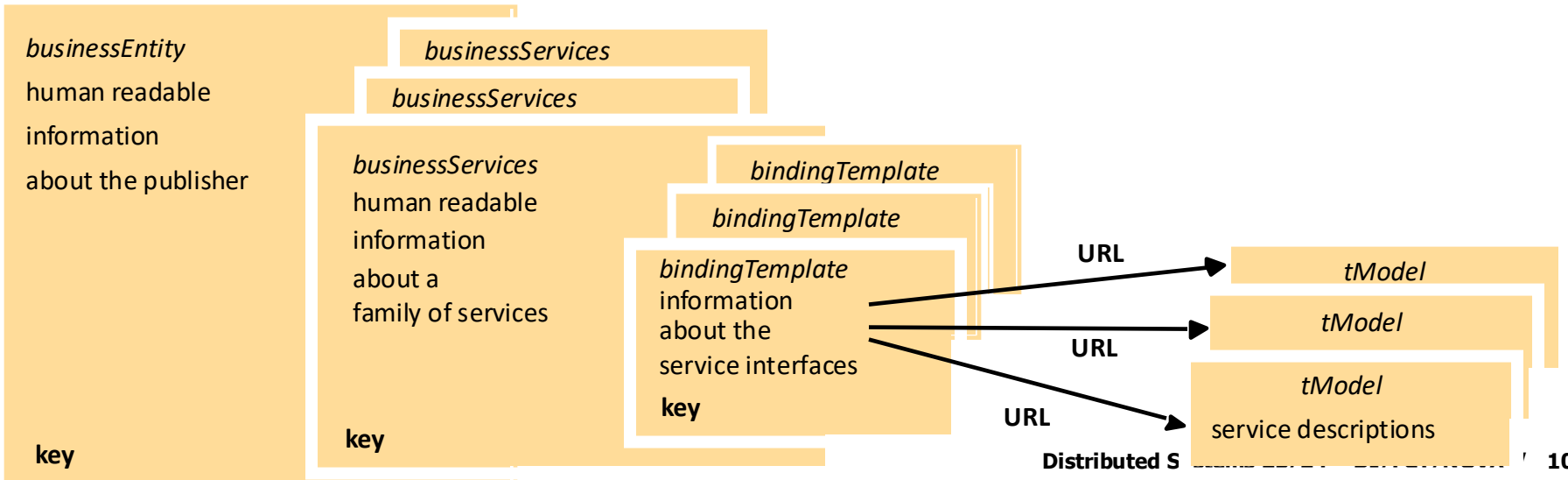
O documento WSDL pode tipicamente ser obtido diretamente a partir de uma instância do serviço em execução.

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 código 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

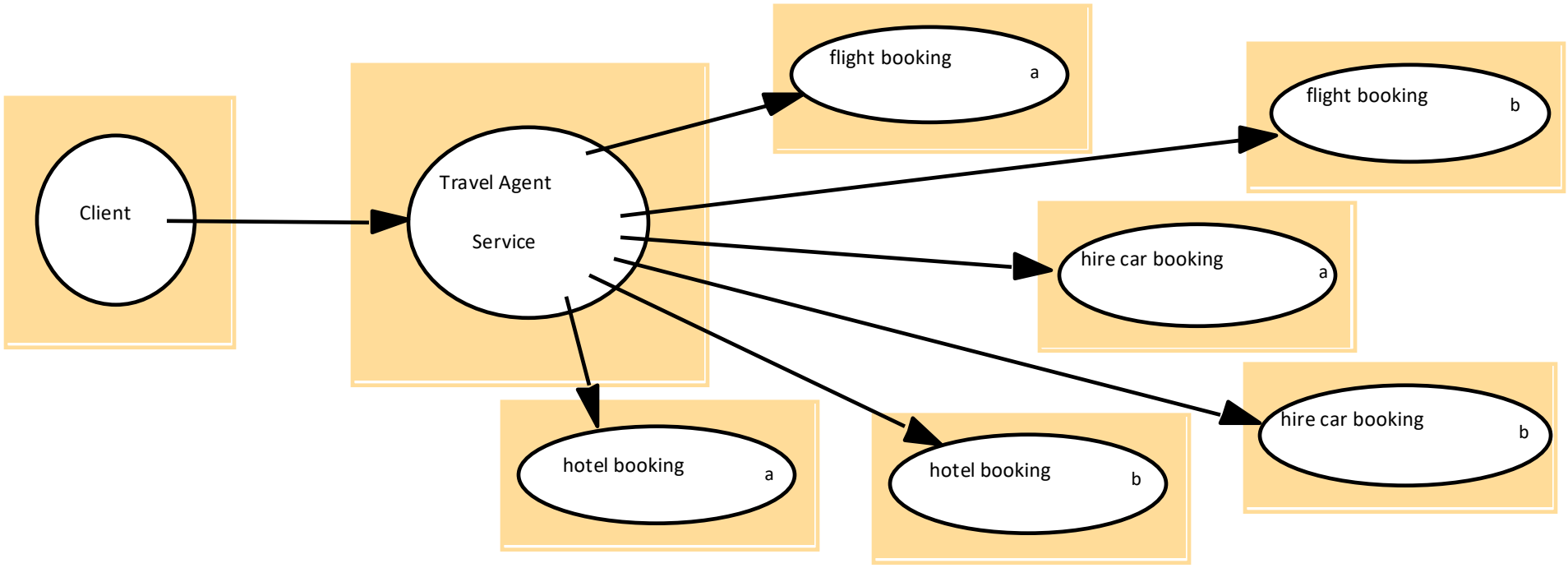
WS-Security: define como efectuar invocações seguras

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

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

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(String[] args) {
        ...
        Endpoint.publish(serverURI,
          new SoapServersWebService());
        ...
    }
}
```

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

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>/path?wsdl",
SoapServers servers = service.getPort(
      servers.soap.SoapServers.class );

Server srv = ...

servers.createServer( srv );

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

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

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.SoapServers.class );

Server srv = ...

servers.createServer( srv );

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

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

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 recursos, na sua representação e em como estes são afectados por um conjunto de métodos (i.e., operações) 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