**Web Services IV**

**Objectivos**

* Configurar o limite de tempo de espera do cliente
* Invocar uma operação unidireccional *(one-way)*
* Fazer chamadas assíncronas *(async)*

**Índice:**

* [Limite de tempo](http://disciplinas.tecnico.ulisboa.pt/leic-sod/2016-2017/labs/11-ws4/index.html#timeout)
* [Operações unidireccionais](http://disciplinas.tecnico.ulisboa.pt/leic-sod/2016-2017/labs/11-ws4/index.html#oneway)
* [Operações assíncronas](http://disciplinas.tecnico.ulisboa.pt/leic-sod/2016-2017/labs/11-ws4/index.html#async)

**Limite de tempo**



O exemplo abaixo demonstra como configurar os tempos de espera pelas respostas de Web Services. No fim do tempo, caso a resposta não tenha sido recebida, é atirada uma excepção.

O JAX-WS distingue dois *timeouts* distintos:

* *connection* - quando tempo deve esperar até se estabelecer uma ligação com o servidor;
* *receive* - quando tempo deve esperar até receber a resposta a um pedido feito ao servidor.

Exemplo:

* [**Web Service client with timeouts** [ZIP](http://disciplinas.tecnico.ulisboa.pt/leic-sod/2016-2017/labs/11-ws4/hello-ws-cli_timeout.zip)](http://disciplinas.tecnico.ulisboa.pt/leic-sod/2016-2017/labs/11-ws4/hello-ws-cli_timeout.zip)
* Para verificar o comportamento do timeout do cliente, pode experimentar [este servidor](http://disciplinas.tecnico.ulisboa.pt/leic-sod/2016-2017/labs/11-ws4/hello-ws_sleep.zip) que introduz um atraso artificial na execução das suas operações.

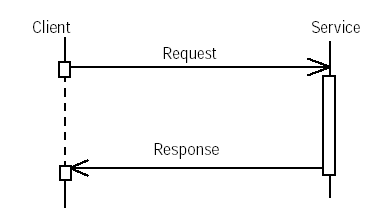
**Operações unidireccionais**



Este exemplo demonstra como definir operações unidireccionais, ou seja, operações de Web Service que não enviam resposta.

* [Implementation-first Web Service with one-way operation that sleeps [ZIP](http://disciplinas.tecnico.ulisboa.pt/leic-sod/2016-2017/labs/11-ws4/bye-ws_oneway_sleep.zip)](http://disciplinas.tecnico.ulisboa.pt/leic-sod/2016-2017/labs/11-ws4/bye-ws_oneway_sleep.zip)  
  Lançar servidor, abrir o WSDL gerado e confirmar o que é diferente na operação one-way. No caso *contract-first*, deverão escrever um WSDL semelhante ao que é gerado neste exemplo, ou seja, a operação *one-way* não declara output no portType e no binding.
* [Web Service client of one-way operation [ZIP](http://disciplinas.tecnico.ulisboa.pt/leic-sod/2016-2017/labs/11-ws4/bye-ws-cli_oneway.zip)](http://disciplinas.tecnico.ulisboa.pt/leic-sod/2016-2017/labs/11-ws4/bye-ws-cli_oneway.zip)  
  - o cliente é idêntico ao habitual, mas podem confirmar que não espera pela conclusão da execução da operação *one-way* no servidor

**Operações assíncronas**



Em situações em que o cliente não pretenda ficar bloqueado à espera da resposta do servidor, é possível fazê-lo através de uma invocação assíncrona.

* [Web Service for asynchronous invocations [ZIP](http://disciplinas.tecnico.ulisboa.pt/leic-sod/2016-2017/labs/11-ws4/echo-ws_async.zip)](http://disciplinas.tecnico.ulisboa.pt/leic-sod/2016-2017/labs/11-ws4/echo-ws_async.zip)  
  - Lançar servidor da forma habitual
* [Web Service client of asynchronous operations [ZIP](http://disciplinas.tecnico.ulisboa.pt/leic-sod/2016-2017/labs/11-ws4/echo-ws-cli_async.zip)](http://disciplinas.tecnico.ulisboa.pt/leic-sod/2016-2017/labs/11-ws4/echo-ws-cli_async.zip)  
  - O cliente é diferente do habitual, porque não fica bloqueado à espera da resposta do servidor.

Uma possível maneira de fazer invocações assíncronas é através dos métodos com sufixo Async. Para que estes métodos sejam gerados é necessário do lado do cliente indicar um ficheiro de *binding*. Os *stubs* assim gerados passam a incluir tanto os métodos para invocação sincrona como assíncrona.

<bindings

xmlns:xsd="http://www.w3.org/2001/XMLSchema"

xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"

xmlns="http://java.sun.com/xml/ns/jaxws" >

<bindings node="wsdl:definitions">

<enableAsyncMapping>true</enableAsyncMapping>

</bindings>

</bindings>

Para uma invocação assíncrona, o cliente deve executar um método com o sufixo Async e de seguida usar o método Response.isDone() para verificar se a resposta já chegou. Nesta solução o cliente invoca o método remoto sem ficar bloqueado, ficando responsável por verificar quando o servidor já respondeu através do objecto Response. Só depois da resposta ter chegado, pode então obter o seu resultado através do objecto Response.

// asynchronous call with polling

Response response = port.echoAsync(name);

while (!response.isDone()) {

Thread.sleep(100 /\* milliseconds \*/);

/\* while waiting for response do other calls... \*/

String result = port.fastEcho(name);

System.out.print("Synchronous call result: ");

System.out.println(result);

}

System.out.println("Async->" + response.get().getReturn());

Um outro modelo de funcionamento é o registo de um objecto de *callback* do tipo AsyncHandler aquando da execução da chamada assíncrona. Quando a resposta chega, um método desse objecto é invocado.

static boolean finished = false;

...

// asynchronous call with callback

port.echoAsync(name, new AsyncHandler() {

@Override

public void handleResponse(Response response) {

try {

System.out.println();

System.out.print("Asynchronous call result arrived: ");

System.out.println(response.get().getReturn());

finished = true;

} catch (InterruptedException e) {

System.out.println("Caught interrupted exception.");

System.out.print("Cause: ");

System.out.println(e.getCause());

} catch (ExecutionException e) {

System.out.println("Caught execution exception.");

System.out.print("Cause: ");

System.out.println(e.getCause());

}

}

});

while (!finished) {

Thread.sleep(100);

System.out.print(".");

System.out.flush();

}

Em ambos os casos, a resposta é obtida invocando o método response.get() que lança uma excepção caso esta tenha sido retornada pelo método remoto. Caso o método remoto retorne Void, este método lancará uma NullPointerException. Caso contrário, o objecto retornado pode ser obtido com o método getReturn().

Repare que não é preciso alterar o servidor para que o cliente possa fazer invocações assíncronas.

**Exercício**

**Quarta parte do projeto**

O objetivo deste exercício é replicar o mediador e implementar o mecanismo de provas de vida entre mediador primário e secundário.

1) Replicar o mediator-ws:

1. A forma mais simples de permitir múltiplas instâncias do mediador é parametrizar as propriedades de configuração com um número de instância, tal como se fez para o supplier-ws.
2. Adicionar as seguintes definições ao pom.xml (substituir CXX pelo identificador do grupo):
3. ...
4. <group.id>CXX</group.id>
5. <ws.i>1</ws.i>
6. <ws.host>localhost</ws.host>
7. <ws.port>807${ws.i}</ws.port>
8. <ws.url>http://${ws.host}:${ws.port}/mediator-ws/endpoint</ws.url>
10. <ws.name>${group.id}\_Mediator</ws.name>
11. <supplier.ws.name>${group.id}\_Supplier</supplier.ws.name>

...

* + O porto do URL do serviço foi trocado para 8071, por omissão.
  + O nome do serviço não inclui o número de instância porque o mediador secundário não se regista no UDDI. O secundário "não existe" para os clientes.

1. Para lançar o Mediador Primário:
   * mvn compile
   * mvn exec:java  
     Por omissão, será a instância 1, que fica à escuta de pedidos no porto 8071 e que se regista com o nome CXX\_Mediator.
2. Para lançar o Mediador Secundário:
   * mvn exec:java **-Dws.i=2**  
     A instância **2** fica à escuta de pedidos no porto 807**2** e não se regista no UDDI.

Cada instância, ao arrancar, deve imprimir para a consola uma mensagem que indique se está a funcionar como primário ou como secundário

A partir deste momento já existem duas instâncias do mediador a correr, mas ambas vão ter estados diferentes, pelo que não são ainda réplicas. A replicação de estado será tratada depois.

2) Pretende-se agora modificar o contrato WSDL do mediador para permitir uma operação unidireccional de prova de vida: imAlive.

1. Criar nova versão do WSDL. Pretende-se manter a compatibilidade com as operações existentes, pelo a modificação seguinte deve apenas **acrescentar** elementos e operações.
2. Sem alterar as operações já definidas no WSDL, adicionar uma nova operação imAlive.
3. Fazer mvn generate-sources
4. Implementar método imAlive() no mediador onde:
   * Se for mediador primário, retorna sem executar qualquer instrução.
   * Se for mediador secundário, guarda a data da hora da chegada dessa prova de vida (*timestamp* do instante em que o método foi executado).
5. Adicionar método imAlive() no MediatorClient que executa a chamada do método sobre o *stub*.

3) Pretende-se agora criar uma classe LifeProof.java que faz o lançamento de provas de vida do mediador primário para secundário.

1. Para as provas de vida, será necessário ter uma atividade que se executa de forma independente.  
   Esta atividade deve estender a classe *Thread* ou *TimerTask* do Java:
   * [thread [ZIP](http://disciplinas.tecnico.ulisboa.pt/leic-sod/2016-2017/labs/11-ws4/thread.zip)](http://disciplinas.tecnico.ulisboa.pt/leic-sod/2016-2017/labs/11-ws4/thread.zip) - criação de novas tarefas
   * [timer [ZIP](http://disciplinas.tecnico.ulisboa.pt/leic-sod/2016-2017/labs/11-ws4/timer.zip)](http://disciplinas.tecnico.ulisboa.pt/leic-sod/2016-2017/labs/11-ws4/timer.zip) - criação de novas tarefas temporizadas
2. Uma vez escolhida *Thread* ou *TimerTask*,  
   criar uma classe LifeProof.java no mediator-ws.
3. Implementar o método run da classe para que, se for o mediador primário:
   * Cria um cliente de mediador que aponta para o mediador secundário  
     (através de convenção estabelecida ou argumento explícito - não usa UDDI porque se assume que o primário conhece o secundário)
   * De 5 em 5 segundos (valor configurável) chama o método imAlive() no servidor através do cliente
4. Modificar a classe MediatorApp de modo a lançar a classe LifeProof antes de ficar à escuta de pedidos.

**Próximos passos:**

4) Acrescentar à classe LifeProof.java o comportamento para o caso de ser o mediador secundário

5) Modificar novamente o contrato WSDL para conter as operações auxiliares de atualização de estado do mediador secundário.  
Estas operações devem ser unidireccionais.

* void updateShopHistory(...)
* void updateCart(...)

6) Implementar a lógica de atualizações entre os dois mediadores, tendo em atenção a execução de pedidos repetidos avaliando a idempotência das operações

7) Utilizar as invocações de operações com *timeout* para alterar o MediatorClient de modo a garantir um *front-end* com a semântica pretendida no enunciado, tendo em atenção os vários tipos de exceções de comunicação.

* Ou seja, tornar o cliente tolerante a faltas de quebra de ligação TCP dos pedidos-respostas HTTP
* Os intervalos de tempo devem ser configuráveis
* Mais informação sobre [tratamento de exceções](http://disciplinas.tecnico.ulisboa.pt/leic-sod/2016-2017/labs/01-tools/exceptions/index.html) e sobre [exceções de Web Services](http://disciplinas.tecnico.ulisboa.pt/leic-sod/2016-2017/labs/05-ws1/junit-it/ws-exceptions.html).

**Continuação de bom trabalho!**