FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Departamento de Informática

Mestrado Integrado em Engenharia Informática
Sistemas Distribuídos – 1º teste, 28 de Abril de 2017
2º Semestre, 2016/2017

NOTAS: Leia as questões atentamente antes de responder. **O teste é sem consulta. A duração do teste é 1h30min**. O teste contém **10** páginas.

Nome: _____ **Número**: _____

1) Pretende-se desenvolver um serviço web para gerir um fórum de discussão (*message board*). Para esse fim, um serviço REST e um serviço SOAP irão disponibilizar as seguintes operações: i) listar todas as mensagens presentes no *message board*; ii) criar uma nova mensagem no *message board*, sendo que o identificador dessa mensagem é retornado ao cliente; iii) substituir o conteúdo de uma mensagem no *message board*, dado o seu identificador; e finalmente, iv) eliminar uma mensagem do *message board,* dado o seu identificador. O código (incompleto) destes serviços em REST e SOAP encontra-se apresentado respetivamente no anexo A e B. As mensagens são modeladas por recurso à classe *Message*. Operações descritas devem retornar um erro para o cliente se a mensagem não existir (código HTTP 404 no caso do serviço REST e uma exceção `NoSuchMessageException` no caso do serviço SOAP, também apresentada no Anexo B).

   a) Complete o Anexo A, como achar necessário, sabendo que as operações listadas acima utilizam os seguintes URLs para um servidor a executar na máquina local no porto 8080.

      i) http://localhost:8080/messageboard/messages

      ii) http://localhost:8080/messageboard/messages/

      iii) http://localhost:8080/messageboard/messages/{id}

      iv) http://localhost:8080/messageboard/messages/{id}

   b) Complete o Anexo B, como achar necessário.

2) No Anexo C, apresenta-se o código java de dois processos que usam *multicast* para comunicarem entre si. O processo, denominado servidor, espera permanentemente por mensagens *multicast* com qualquer conteúdo, e responde a estas mensagens (através de *unicast* UDP) para o emissor da mensagem com um URL. O processo cliente emite uma mensagem *multicast* para o endereço e porto usados pelo servidor, e espera receber uma resposta com um URL que, seguidamente, imprime no terminal.

   É possível que o processo cliente nunca gere nenhum *output*, assumindo que o servidor nunca falha e que se encontra a executar? Se sim, explique como resolveria o problema. Se não, explique o que garante que um *output* é gerado (nas condições dadas).

Sim / Não ...

As falhas em sistemas distribuídos não ocorrem apenas nos componentes de servidor ou cliente, também podem acontecer nos canais de comunicação. Se o cliente enviar a sua mensagem, ela chegar ao servidor (ou não), mas a mensagem do servidor não conseguir chegar ao cliente, nenhum output é gerado pois nunca recebe nada. Para garantir que obtemos um output, podemos utilizar um timeout após do qual podemos dizer que a conexão falhou. Alternativamente, para além do timeout poderiamos ter um certo número de vezes em que tentavamos comunicar novamente com o servidor, ao fim das quais diriamos que a conexão falhou.

3) Considere um serviço REST que expõe uma única operação que permite obter a Data e hora correntes no servidor. O código do servidor e do cliente (recorrendo à linguagem Java e à biblioteca Jersey) encontram-se listados no Anexo D.

   Observou-se que quando se executa o servidor numa máquina, e de seguida na mesma máquina se executa o cliente apresentado, o cliente termina com a exceção com o seguinte *stack trace*.

```
Exception in thread "main" javax.ws.rs.InternalServerErrorException: HTTP 500 Internal Server Error
  at org.glassfish.jersey.client.JerseyInvocation.convertToException(JerseyInvocation.java:1032)
  at org.glassfish.jersey.client.JerseyInvocation.translate(JerseyInvocation.java:819)
  at org.glassfish.jersey.client.JerseyInvocation.access$700(JerseyInvocation.java:92)
  at org.glassfish.jersey.client.JerseyInvocation$2.call(JerseyInvocation.java:701)
  at org.glassfish.jersey.internal.Errors.process(Errors.java:315)
  at org.glassfish.jersey.internal.Errors.process(Errors.java:297)
  at org.glassfish.jersey.internal.Errors.process(Errors.java:228)
  at org.glassfish.jersey.process.internal.RequestScope.runInScope(RequestScope.java:444)
  at org.glassfish.jersey.client.JerseyInvocation.invoke(JerseyInvocation.java:697)
  at org.glassfish.jersey.client.JerseyInvocation$Builder.method(JerseyInvocation.java:420)
  at org.glassfish.jersey.client.JerseyInvocation$Builder.get(JerseyInvocation.java:316)
  at question2.TimeClient.main(TimeClient.java:27)
```

   Consulte o código no Anexo D para identificar a origem do erro. Indique como este erro foi detetado e processado no servidor, bem como este foi comunicado ao cliente.

Ao recebermos 500 Internal Server Error, sabemos que a o código começou a executar. Desta forma, o erro foi um null pointer no dtf.format(now). Como foi lançada uma exceção null pointer, o servidor enviou um erro HTTP 500 ao cliente.

4) Indique se cada afirmação é **[V]erdadeira** ou **[F]alsa** (**nota: respostas incorretas descontam**):

V ___ Um *container* permite executar aplicações de forma mais leve que uma máquina virtual (do tipo sistema).

V ___ Os vários *containers docker* a executar numa máquina têm IPs diferentes.

V ___ Uma mensagem corrompida é um exemplo de falha arbitrária. arbirtrária - Aconteceu algo que não devia ter acontecido

F ___ Num sistema cliente/servidor particionado cada servidor recebe em média um maior número de pedidos que num sistema cliente/servidor replicado. Particionado - w/n + r/n Replicado - w + r/n

F ___ No sistema BitTorrent, um peer A obtém de outro peer B os blocos de forma ordenada (começando no primeiro bloco conhecido por B e terminando no último) tit for tat não tem ordem de blocos, é aleatório

F ___ Um *edge server* é tipicamente usado para armazenar todos os conteúdos de um site web. apenas conteúdos estáticos

F ___ Um ISP não tem vantagem nenhuma em ter um *edge server* a funcionar nos seus centros de dados.

F ___ As invocações HTTP assíncronas são usadas principalmente para permitir que uma aplicação web faça dois pedidos concorrentemente a dois servidores. são usados para manter a página responsiva

No contexto da invocação remota de métodos/procedimentos:

V ___ Um dos objetivos dos mecanismos de invocação remota é tornar parcialmente transparente a distribuição, fazendo a invocação dum método num servidor remoto semelhante a invocar um método num objeto local.

F ___ o RMI registry é usado para manter os nomes e atributos de servidores implementados usando o sistema Java RMI. não é suposto saber atributos

V ___ Nos web services SOAP, o WSDL especifica as mensagens que devem ser enviadas para invocar cada um dos métodos do servidor.

F ___ Nos web services SOAP, ao passar um objeto como parâmetro dum método, este é passado por referência.

V ___ O mecanismo de codificação de dados ProtoBuf pode ser usado em qualquer linguagem.

F ___ O UDDI é necessário para o funcionamento dum sistema de web services SOAP.

F ___ Para implementar uma semântica "exatamente uma vez", apenas o servidor precisa de manter informação em memória estável.

F ___ Seria possível implementar o modelo REST sobre conexões UDP. O datagram packet é demasiado pequeno para maior parte dos dados, entre outras razões

V ___ Uma das vantagens da utilização de uma pool de *threads* para processamento de pedidos num sistema de invocação remota é permitir dimensionar o número de pedidos que podem correr simultaneamente consoante a capacidade da máquina em que o servidor está a executar.

5) Considere que pretende implementar uma aplicação/rede social concorrente do popular Musical.ly. Nesta sua aplicação, o karaoke.ly, os utilizadores poderiam gravar as suas interpretações *karaoke* de vídeos musicais e partilhá-las com os seus amigos. A aplicação dum utilizador reproduz uma interpretação musical a partir de dois ficheiros: o vídeo original e o som gravado por um utilizador. Assim, a aplicação teria de disponibilizar operações para: (1) obter o conteúdo de um vídeo; (2) gravar o som de uma interpretação relativa a um vídeo; (3) obter o som de uma interpretação; (4) gravar informação de uma interpretação de um utilizador; (5) listar as interpretações dum utilizador.
   Para o lançamento do sistema, a sua aplicação permitirá a um utilizador partilhar as suas interpretações de forma privada com um conjunto de outros utilizadores ou de forma pública.

   a) Considere que pretende implementar o seu sistema na Amazon AWS, tendo ao seu dispor um conjunto de três centros de dados – Europa, América do Norte e Ásia. Explique como usaria estes centros de dados para armazenar os diferentes tipos de recurso (vídeos, interpretações, informação sobre interpretações) que a aplicação guarda.

Os vídeos seriam geo-replicados por todos os 3 centros de dados, para permitir a utilização de qualquer vídeo em qualquer região da forma mais rápida possível. As interpretações e a informação sobre elas, não estando limitadas à região onde foram criadas, teriam também que ser geo-replicadas. Desta forma, era assegurada uma latência baixa para todos os utilizadores e, secundariamente, a segurança dos dados.

http:/kareok.ly

b) Considere que pretende disponibilizar uma API REST para as operações indicadas. Apresente o URL e operação que usaria para cada operação.

(1) /video/{id}
GET

(2) /rendition/video/{id}
POST

(3) /rendition/{id}
GET

(4) /rendition/{id}
POST

(5) /userList/{id}
GET

c) Suponha que funcionalidade para permitiria a um utilizador que estivesse a usar a aplicação ser informado sempre que um seu amigo produzisse uma nova interpretação. Que mecanismo de comunicação usaria para suportar esta funcionalidade. Justifique.

6) Numa arquitetura *three-tier*, é comum os servidores aplicacionais não manterem estado.
   a) Discuta as vantagens e desvantagens desta aproximação (servidores aplicacionais *stateless*) para a criação de aplicações web cuja popularidade pode variar muito ao longo do tempo, i.e., em que podem haver períodos com um baixo número de utilizadores e períodos com um muito elevado número de utilizadores.

Em termos de vantagens:
- um servidor stateless pode ser replicado sem complexidade pois não existe estado para manter coerente em todas as instâncias
- a lógica está separada dos dados o que resulta numa implementação mais organizada, onde as falhas de dados são tratadas na base de dados e a falhas de lógica no servidor
- existe uma melhor tolerância a falhas, precisamente por ser facilmente replicado (se um servidor falhar, o pedido pode ser reencaminhado para outro que estja disponível) e por ter os dados numa base de dados (acessos concorrentes e outros tipos de falhas podem ser abordadas pela BD)

Desvantagens: - pode aumentar a latência em situações, uma vez que para além de o cliente comunicar com o servidor, o servidor tem que comunicar com a BD
- é mais complexo de implementar e de manter

b) Neste tipo de arquitetura é comum recorrer-se a um serviço de cache distribuído (e.g. *memcached*) para armazenar o resultado de uma *query* ao serviço de armazenamento ou do resultado dum processamento complexo envolvendo múltiplas *queries*. Indique que arquitetura usaria para implementar o serviço de cache distribuído. Justifique.

Anotações JAVA REST Jersey:

@Path( )
@GET
@POST
@PUT
@DELETE
@Consumes( )
@Produces( )
@PathParam( )

Listagem de Media Types a considerar:

MediaType.APPLICATION_OCTET_STREAM
MediaType.APPLICATION_JSON

Anexo A

```
  @Path("messageboard/messages")

public class MessageBoardRessources {

   public SortedMap<String,Message> messages;

   public MessageBoardRessources() {
      super();
      this.messages = new TreeMap<String, Message>();
   }



   @GET@Path("/")
   @Produces(APPLICATION_JSON)

   public Message[] getMessages()  {
      return messages.values().toArray(new Message[messages.size()]);
   }



   @POST@Path("/")
   @Consumes(APPLICATION_JSON)
   @Produces(APPLICATION_JSON)

   public String postMessage(                    Message msg) {
      this.messages.put(msg.getID(), msg);
      return msg.getID();
   }



   @PUT@Path("/{id}")
   @Consumes(APPLICATION_JSON)

   public void updateMessage     @PathParam    String id,                    Message msg) {
      if(!this.messages.containsKey(id))
         throw new WebApplicationException( NOT_FOUND );
      else
         this.messages.get(id).update(msg);
   }



   @DELETE@
   Path("/{id}")
   public void deleteMessage     @PathParam    String id) {
      if(!this.messages.containsKey(id))
         throw new WebApplicationException(           );
      else
         this.messages.remove(id);
   }
}
```

Anexo B

```
@WebService(...)
public class MessageBoardServiceImpl {

   public SortedMap<String,Message> messages;


   public MessageBoardServiceImpl() {
      super();
      this.messages = new TreeMap<String, Message>();
   }


   @WebMethod
   public Message[] getMessages() {
      return messages.values().toArray(new Message[messages.size()]);
   }


   @WebMethod
   public int postMessage(                        Message msg) {
      this.messages.put(msg.getID(), msg);
      return (this.messages.size() - 1);
   }


   @WebMethod
   public void updateMessage(                     String id,                      Message msg)
         throws NoSuchMessageException {
      if(!this.messages.containsKey(id))
         throw new NoSuchMessageException("Message with id " + id + " not found.");
      else
         this.messages.get(id).update(msg);
   }



   @WebMethod
   public void deleteMessage(                     String id)
         throws NoSuchMessageException {
      if(!this.messages.containsKey(id))
         throw new NoSuchMessageException("Message with id " + id + " not found.");
      else
         this.messages.remove(id);
   }



   @WebFault
   public class NoSuchMessageException extends Exception {

      private static final long serialVersionUID = -2618304854213373109L;

      public NoSuchMessageException(String msg) {
         super(msg);
      }

   }
}
```

Anexo C

```
//Código do processo servidor.

public class MulticastServer {

   public static void main(String[] args) throws Exception {
      InetAddress address = InetAddress.getByName("227.1.1.1");
      MulticastSocket socket = new MulticastSocket(6666);
      socket.joinGroup( address );
      String myURL = "http://" + InetAddress.getLocalHost().getHostAddress() + ":8080/";
      while( true ) {
         byte[] buffer = new byte[65536];
         DatagramPacket request = new DatagramPacket(buffer, buffer.length);
         socket.receive( request );
         DatagramPacket reply = new DatagramPacket(myURL.getBytes(), myURL.length());
         reply.setSocketAddress( new InetSocketAddress(request.getAddress(),
request.getPort()) );
         socket.send(reply);
      }
   }

}




//Código do processo cliente.

public class MulticastClient {

   public static void main(String[] args) throws Exception {

      MulticastSocket socket = new MulticastSocket();

      //Prepare Datagram to send.
      String message = "RandezVouServer";
      DatagramPacket request = new DatagramPacket(message.getBytes(), message.length());
      request.setAddress(InetAddress.getByName("227.1.1.1"));
      request.setPort(6666);

      DatagramPacket reply = new DatagramPacket(new byte[65536], 65536);
      String randezVousServerAddress = null;
      while(randezVousServerAddress == null) {
         socket.send(request);
         socket.receive(reply);
         randezVousServerAddress = new String(reply.getData(), 0, reply.getLength());
      }

      socket.close();
      System.out.println(randezVousServerAddress);
   }

}
```

Anexo D

```
//Código do servidor (classe que representa o recurso REST exposto pelo servidor).

@Path("/rest")
public class SimpleTimeResources {

   @GET
   @Path("/time")
   @Produces(MediaType.APPLICATION_JSON)
   public String getMessages() {
      DateTimeFormatter dtf = DateTimeFormatter.ofPattern("yyyy/MM/dd HH:mm:ss");
      LocalDateTime now = null;
      return dtf.format(now);
   }
}



//Código do cliente

public class TimeClient {

   public static void main(String[] args) throws IOException {

      ClientConfig config = new ClientConfig();
      Client client = ClientBuilder.newClient(config);

      URI baseURI = UriBuilder.fromUri("http://localhost:8080/").build();

      WebTarget target = client.target(baseURI);


      String time = target.path("/rest/time").request().accept(MediaType.APPLICATION_JSON)
            .get(String.class);

      System.out.println("Time at server: " + time);
   }
}
```