

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 ...

- 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.

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

- Um *container* permite executar aplicações de forma mais leve que uma máquina virtual (do tipo sistema).
- Os vários *containers docker* a executar numa máquina têm IPs diferentes.
- Uma mensagem corrompida é um exemplo de falha arbitrária.
- Num sistema cliente/servidor particionado cada servidor recebe em média um maior número de pedidos que num sistema cliente/servidor replicado.
- 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)
- Um *edge server* é tipicamente usado para armazenar todos os conteúdos de um site web.
- Um ISP não tem vantagem nenhuma em ter um *edge server* a funcionar nos seus centros de dados.
- As invocações HTTP assíncronas são usadas principalmente para permitir que uma aplicação web faça dois pedidos concorrentemente a dois servidores.

No contexto da invocação remota de métodos/procedimentos:

- 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.
- o RMI registry é usado para manter os nomes e atributos de servidores implementados usando o sistema Java RMI.
- Nos web services SOAP, o WSDL especifica as mensagens que devem ser enviadas para invocar cada um dos métodos do servidor.
- Nos web services SOAP, ao passar um objeto como parâmetro dum método, este é passado por referência.
- O mecanismo de codificação de dados ProtoBuf pode ser usado em qualquer linguagem.
- O UDDI é necessário para o funcionamento dum sistema de web services SOAP.
- Para implementar uma semântica "exatamente uma vez", apenas o servidor precisa de manter informação em memória estável.
- Seria possível implementar o modelo REST sobre conexões UDP.
- 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.

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

(1)

(2)

(3)

(4)

(5)

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 aplicativos não manterem estado.

a) Discuta as vantagens e desvantagens desta aproximação (servidores aplicativos *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.

- 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

```
public class MessageBoardRessources {

    public SortedMap<String,Message> messages;

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

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

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

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

    public void deleteMessage                String id) {
        if(!this.messages.containsKey(id))
            throw new WebApplicationException(                );
        else
            this.messages.remove(id);
    }
}
```

Anexo B

```
public class MessageBoardServiceImpl {

    public SortedMap<String,Message> messages;

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

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

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

    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);
    }

    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);
    }

    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 = "RendezVousServer";
        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 rendezVousServerAddress = null;
        while(rendezVousServerAddress == null) {
            socket.send(request);
            socket.receive(reply);
            rendezVousServerAddress = new String(reply.getData(), 0, reply.getLength());
        }

        socket.close();
        System.out.println(rendezVousServerAddress);
    }
}
```

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);
    }
}
```