PROGRAMAÇÃO DISTRIBUÍDA

Exame Teórico

23 de Janeiro de 2013

Teste sem consulta / Duração: 60 minutos / Todas as perguntas possuem a mesma cotação.

---

1. Apresente, em termos genéricos, a sequência de operações que é automaticamente desencadeada quando se invoca o método *double getResult(double x, double y)* num objecto remoto Java RMI.

   R: Em termos genéricos, a sequência de operações que é automaticamente desencadeada quando se invoca o método getResult num objeto remoto Java RMI será algo por entre as linhas de (assumindo que o registry, servidor e cliente já estão ligados e que o cliente já tem uma referência para o serviço – obtido através do nome no registry – servidor de diretório):

   a getResult é invocado no stub do Cliente (internamente envia pedido por servidor e espera por resposta nos sockets TCP designados)
   b pedido é recebido no skeleton do Servidor
   c getResult é invocado a partir do skeleton no serviço localmente no servidor
   d resultado é enviado como resposta para o Cliente
   e resultado é recebido no Cliente

2. Explique, em termos de funcionalidades/aplicação, quais são as principais diferenças existentes entre objectos do tipo *Socket* e *ServerSocket* em Java.

   R: Objetos do tipo Socket servem para permitir a comunicação bidirecional orientada a ligação (com recurso ao protocolo TCP/IP) entre dois programas a correr em máquinas virtuais Java diferentes e a sua comunicação é feita através de input streams e output streams (contrariamente a DatagramSockets – protocolo UDP – que comunicam através de pacotes). Objetos do tipo ServerSocket servem para abrir múltiplos canais de comunicação através de Socket's em sequência (habitualmente só são utilizados em servidores enquanto Sockets são obrigatoriamente utilizados em servidores e clientes). Para exemplificar, uma aplicação na qual cada cliente pretende comunicar com o servidor de forma independente, é necessário ter (no servidor) uma thread que aceita pedidos de ligação e inicia uma thread (com um socket novo – através do public Socket ServerSocket.accept()) para cada vez que um pedido de comunicação é feito, na

qual, consequentemente toda a comunicação entre o servidor e o cliente (que se acabou de conectar) será feito.

3. Para que haja comunicação via UDP, a linguagem de programação Java requer o recurso conjunto a objectos do tipo *DatagramSocket* e *DatagramPacket*. Indique os objectivos concretos de cada tipo de objecto e descreva os seus atributos principais (por exemplo, será que existe um atributo relativo ao porto UDP de destino e outro ao porto UDP de origem, ou existe apenas um único atributo relativo ao porto UDP que, na origem, indica… e, no destino, … ?).

   R: Para que haja comunicação via UDP, a linguagem de programação Java requer o recurso conjunto a objectos do tipo *DatagramSocket* e *DatagramPacket*. DatagramSocket é um tipo de socket (que adere ao protocolo da camada de transporte de rede UDP) que comunica através de pacotes (DatagramPacket) e só tem porto e IP local (o IP é o do network interface em utilização na máquina local e o porto pode ser automaticamente ou manualmente escolhido), os pacotes a enviar é que determinam o destino (IP e porto do socket que pretendemos que receba o pacote) dos dados, o socket não sabe nada sobre o destino dos pacotes (daí se dizer que TCP é um protocolo que é orientado a ligação e UDP não).

4. Explique, justificando, se existe algum tipo de relação entre as classes *MulticastSocket* e *DatagramSocket*. Com base na sua discussão, conclua, igualmente, se é possível enviar dados para endereços IP (versão 4) do tipo multicast (classe D) recorrendo a instâncias da classe *DatagramSocket*.

   R: MulticastSocket é um subtipo (relação de herança) de DatagramSocket que permite a receção de pacotes de endereços IP do tipo multicast (através da aderência a estes mesmos endereços/grupos – MulticastSocket.joinGroup(InetAddress addr)). Apesar de não ser possível receber dados de endereços do tipo multicast com DatagramSocket's regulares (por não terem nenhuma forma de aderirem a grupos/endereços de classe D), é possível enviar dados para estes mesmos endereços porque virtualmente a nível de envio, funcionam de forma idêntica.

5. Que diferença existe entre configurar uma *thread* em modo utilizador ou em modo *daemon*? Para cada uma das opções, apresente uma situação concreta em que se justifique a sua escolha.

   R: Uma aplicação em Java só termina quando todas as suas threads em modo utilizador morrerem. O mesmo não se aplica às threads em modo daemon (se as últimas threads vivas forem daemon em algum momento, a aplicação termina). Apesar de ser possível utilizar threads daemon para realizar tarefas de fundo que não tenham importância suficiente para obrigar a aplicação a manter-se viva enquanto não se finalizarem, é consenso da comunidade de programadores Java que são muito raras as situações em que se justifique utilizar threads em modo daemon porque terminá-las sem correr o Thread.join() (método que obriga a thread

que a invoque a esperar até a thread alvo sair do método Thread.run() – existem situações em que se possa passar um intervalo de tempo máximo que a thread invocadora espere) o mesmo que terminá-las abruptamente e como tal é frequente haverem recursos por fechar, para além de que, caso se use algum tipo de I/O na thread Daemon, terminar a metade de uma operação deste tipo causar mesmo corrupção de dados. Threads em modo utilizador são as que mais se usam e um exemplo da sua utilização é abrir um novo canal de comunicação com um cliente que se acabou de conectar à ServerSocket hospedada na aplicação (servidor).

6. Diga, justificando, em que medida pode considerar-se o CORBA como sendo mais flexível do que outras soluções de *middleware* para sistemas distribuídos, tais como o Java RMI e o .*Net Remoting*.

R: O CORBA (Common Object Request Broker Architecture – implementação state-of-the-art do RPC) é considerado mais flexível do que outras soluções de middleware para sistemas dístribuidos como o Java RMI e o .Net Remoting porque é um Middleware que suporta várias linguagens de programação, como C, C++, Java, entre outros e não só as suporta mas suporta várias simultaneamente. Já o Java RMI é específico para Java e o .Net Remoting é específico para o .Net. A forma como suporta estas linguagens é através de uma linguagem de esquema chamada de IDL (Interface Definition Language) que compila para as outras. A forma como as suporta simultaneamente é através da criação de um ORB (Object Request Broker) para cada serviço. O que isto significa é que é possível haver uma aplicação desenvolvida em C++ e outra em Java (ambos na rede – internet) e terem contacto com os objetos remotos (invocarem métodos) de um e do outro. Isto é conseguido através de protocolos como o GIOP (Generic Inter-ORB Protocol, habitualmente recorre-se à especialização deste protocolo para TCP/IP: IIOP (Internet Inter-ORB Protocol)).