Sistemas Distribuídos
Aula 10

Aula de hoje

- Modelo computação distribuída
- RPC
- Marshalling e stubs
- Semântica operacional
- RMI
- *Serverless Computing*

Computação Distribuída

- Como dividir uma computação para torná-la distribuída?
  - arquitetura divide componentes de um sistema
- Como fazer isto? Precisamos de uma abstração de programação
- **Ideia 0**: usar modelo cliente/servidor; troca de mensagens via sockets; troca de dados; seguindo um protocolo
  - ex. HTTP (web)

Exemplo Web

- Cliente estabelece conexão (via sockets)
- Servidor aceita conexão (via sockets), dedica thread para atender cliente
- Cliente envia pedido usando HTTP (objeto)
- Servidor envia objeto requisitado, usando HTTP
- Cliente pode solicitar outros objetos

- Desvantagens deste modelo: baixo nível
  - não abstrai detalhes de sockets e protocolo

Outra Abstração

- Como dividimos a computação em um sistema não distribuído?

Funções (procedimentos)

- **Ideia 1:** usar a mesma abstração para sistemas distribuídos
- chamada de função como forma de distribuir a computação
  - função está em outra máquina

Remote Procedure Call (RPC)

- **RPC**: distribuição da computação através de chamadas de função
- Nível de abstração mais elevado
- Esconde a complexidade
  - nada de sockets ou protocolos para o programador
- Mais transparente
  - independente da máquina, SO, etc
- Modelo compreendido por programadores
  - função recebe parâmetros e retorna um valor

Exemplo em Java

- Lado do cliente

```java
import java.util.*;
import org.apache.xmlrpc.*;

public class JavaClient {
   public static void main (String [] args) {

      try {
         XmlRpcClient server = new XmlRpcClient("http://localhost/RPC2");
         Vector params = new Vector();

         params.addElement(new Integer(17));
         params.addElement(new Integer(13));

         Object result = server.execute("sample.sum", params);

         int sum = ((Integer) result).intValue();
         System.out.println("The sum is: "+ sum);

      } catch (Exception exception) {
         System.err.println("JavaClient: " + exception);
      }
   }
}
```

conecta ao servidor

chama função *sample.sum* no *server* e retorna um objeto

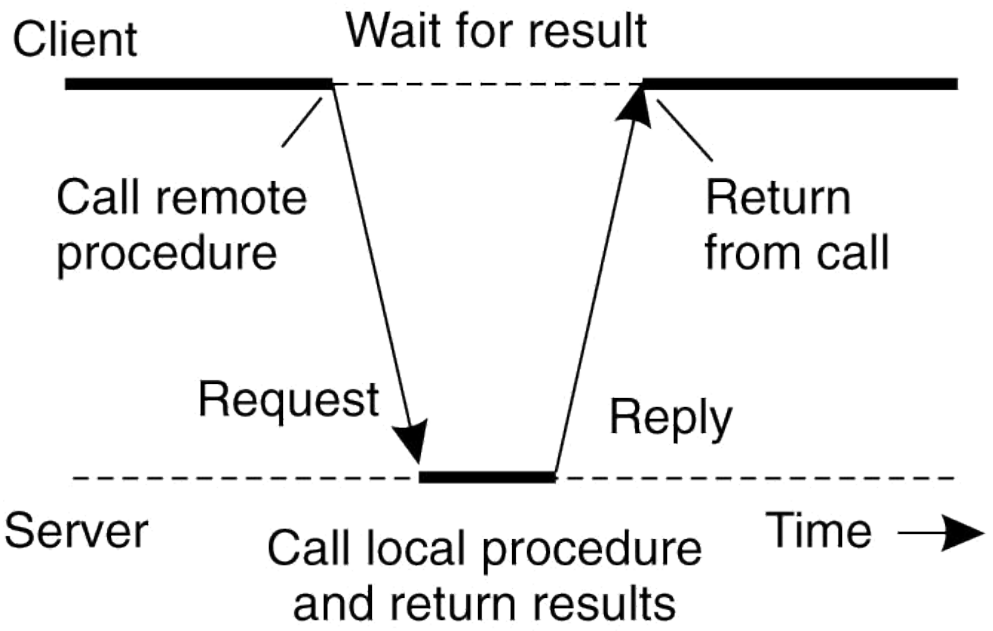
objeto da class Integer

Desafios do RPC

- Processo que chama função e que executa função estão em máquinas diferentes
  - espaço de endereçamento diferentes
  - SO diferente, HW diferente
- Representação dos dados possivelmente diferentes
  - ex. LP (tipos) diferentes, HW diferentes
- Redes e máquinas podem falhar durante a chamada de função

Modelo Síncrono para RPC

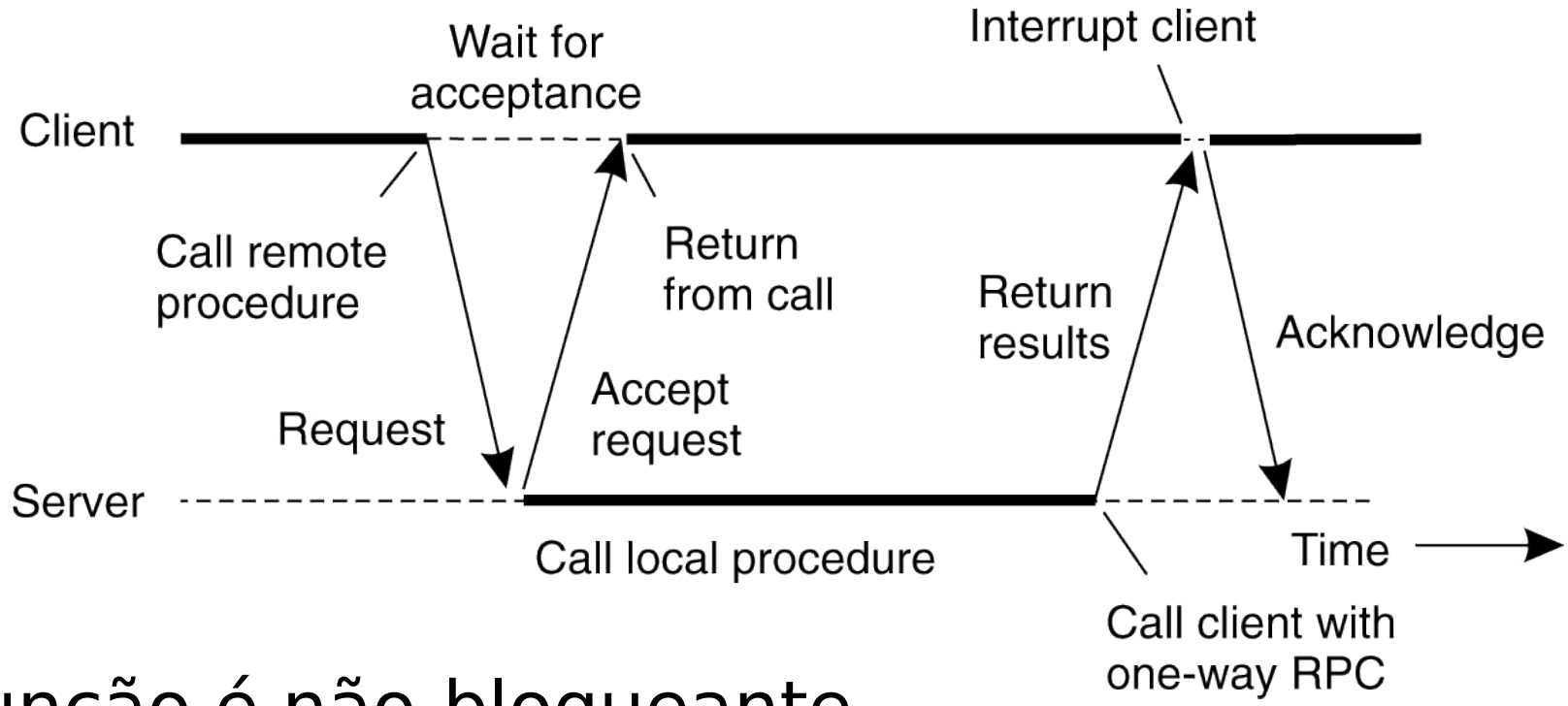
- Síncrono: função retorna quando retorna!



- Função é bloqueante, mais natural para programador
  - retorna quando resultado está pronto

Modelo Assíncrono para RPC

- Assíncrono: função “retorna” duas vezes (fim de chamada, resultado pronto)

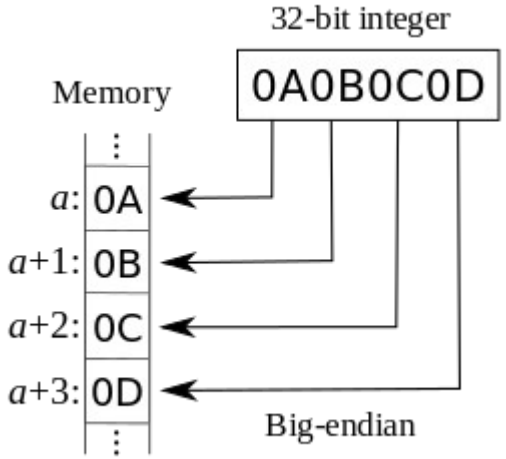


- Função é não-bloqueante
  - permite cliente continuar execução
- Segundo retorno via interrupção (*a la* signal handlers)

Representando Dados

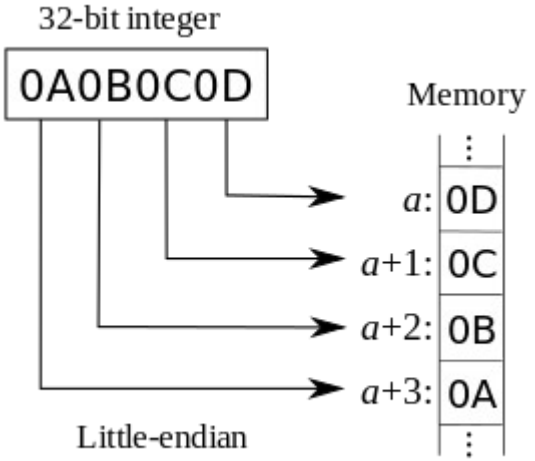
- Como lidar com representação de dados diferentes?
  - ex. *little endian* x *big endian*

Como organizar os bytes de uma palavra (32 bits) na memória do computador?



Byte mais significativo primeiro

ex. ARM, PowerPC



Byte menos significativo primeiro

ex. Intel

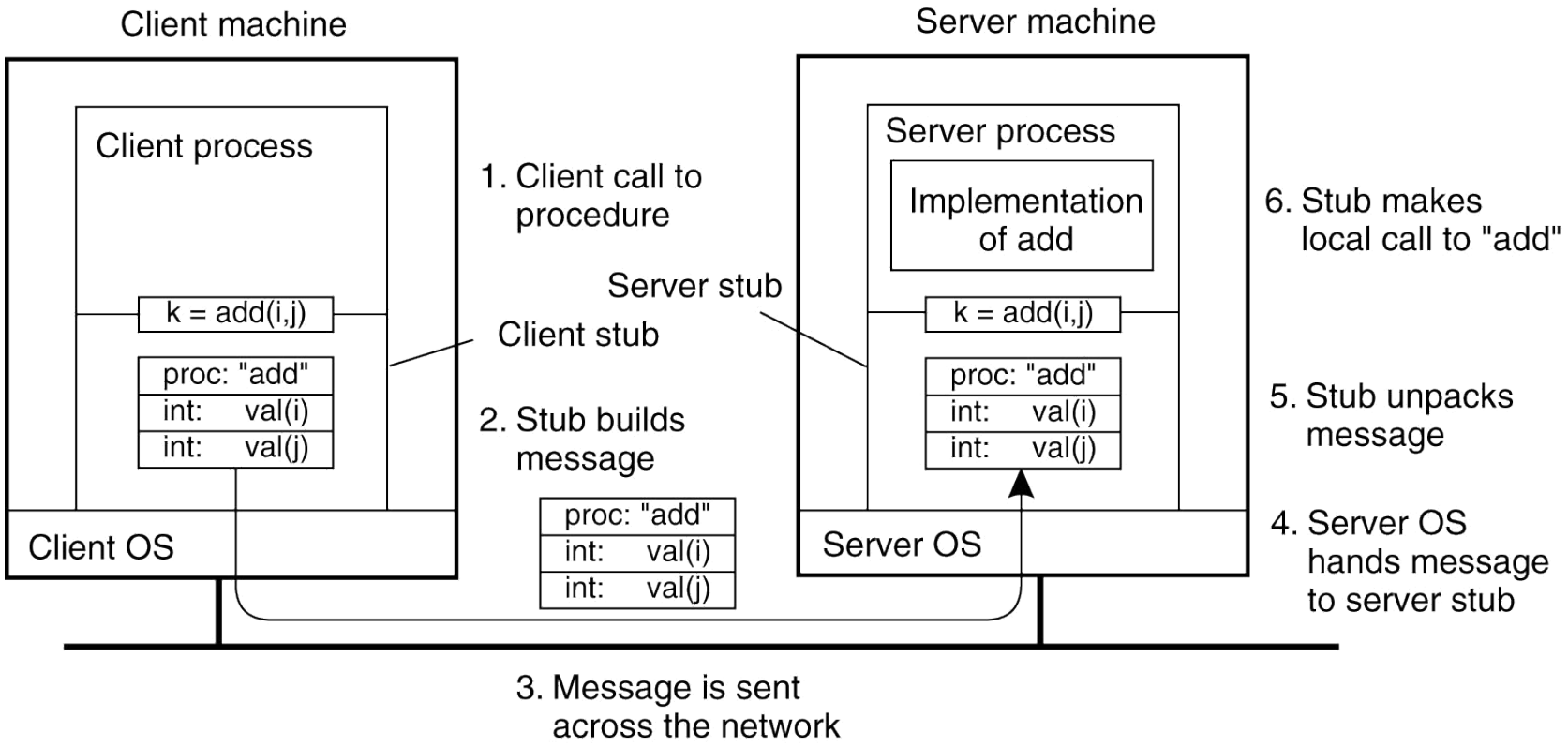
- Chegaram 32 bits da rede, como organizar na memória?
  - little endian ou big endian?
  - depende de quem mandou!

Marshalling e Stubs

- **Ideia:** Converter dados para representação que não depende da máquina
  - de comum acordo entre os dois lados; cada lado faz sua parte
  - encapsulado fora do programa (biblioteca de RPC)
- Usado também para strings, floats, tipos mais complicados (*structs*), etc
- **Stubs:** código responsável por *marshalling* e *unmarshalling* dos parâmetros
  - oferecido pela biblioteca, de uso obrigatório, mas transparente (programador não sabe)
  - descrevem os parâmetros da função através de algum padrão, como XML

Marshalling e Stubs

- Passos em detalhe, retorno é simétrico



- Muitos passos e muitas cópias dos dados
  - fonte de ineficiências

Parâmetros por Referência

- Como passar parâmetros por referência (ponteiros)?
  - espaço de endereçamento é outro

- **Ideia:** *Pass by copy/restore*
  - stub cria estrutura recebida como parâmetro
  - copia dados para a estrutura criada
  - passa dados para servidor
  - recebe dados modificados pelo servidor
  - copia resultado para estrutura original
  - destroi estrutura local (ao stub)

Pass by Copy/Restore

- Transparente para programador
  - mas ineficiente (ainda mais cópias), estrutura pode ser muito grande (tempo de transmissão)
- Como saber tamanho da estrutura de dados?
  - nem sempre explícito na linguagem
- Estrutura de dados que tem ponteiros?
  - criar essas outras estruturas?

Solução parcial

- Depende da biblioteca de RPC

Falhas em RPC

- Como lidar com falhas?
  - ex. servidor falha no meio da chamada

- Chamada local de função (sistema centralizado)
  - se chamada falhar, sistema inteiro falha
- Chamada RPC, o que fazer?
  - **Ideia 0**: imitar chamada local de função
  - Problema: vai gerar muitas falhas, pois falhas parciais fazem parte de sistemas distribuídos!

Como lidar com falhas?

Semântica em RPC

- Falhas parciais são frequentes em SD
  - ex. rede fora do ar
- Como tornar falhas parciais transparente ao programador?

- Introduzir semântica de operação de RPC
  - número de execuções da chamada de função no servidor (programador chama uma vez)
  - 1) exatamente uma vez
  - 2) ao menos uma vez
  - 3) no máximo uma vez
  - 4) zero ou uma vez (com conhecimento)
- Software (stubs) garantem a semântica

Garantindo Semânticas

- Ao menos uma vez: stub faz a chamada repetidas vezes até receber resposta
  - servidor processa, responde, sem guardar estado
  - útil apenas para operações idempotentes, pois servidor pode executar várias vezes
- No máximo uma vez: stub faz a chamada repetidas vezes até receber resposta
  - servidor mantém registro das chamadas, responde a chamadas repetidas sem processar novamente (guarda resposta)
- Exatamente uma vez: stub precisa ter certeza de que a chamada foi executada
  - não pode desisitir até receber resposta (difícil garantir)

Implementando Semânticas

- Semântica definida pela biblioteca de RPC
- Implementanda pela biblioteca
  - assim como os stubs para marshalling
  - nada fácil de garantir semântica
- Semântica permite sistema distribuído operar com falhas parciais
  - fundamental para construção de SD
  - transparente para programador

Remote Method Invocation

- RMI: RPC aplicado a orientação a objetos
  - sai função entra método
- Objetos instanciados do lado do servidor (remote) e do lado do cliente (local)
  - objetos podem persistir no servidor, mantém seu estado
- Facilita a distribuição dos dados no sistema
  - programação ainda mais transparente
- Precisa resolver mesmos problemas que RPC, e alguns outros
  - como persistência dos objetos e sincronização das chamadas

RPC na Nuvem

- *Serverless Computing*: executar uma função (sua) na nuvem
  - não precisa alugar uma máquina virtual
  - basta “instalar” a função na nuvem
- Cobrança é feita por chamada à função
  - *Function as a Service* (FaaS)
  - preço depende do tempo de execução e da memória usada pela função
- Disponível em todas as nuvens: Amazon AWS, Google Cloud, Microsoft Azure
  - nuvem dimensiona recursos dinamicamente
- Revolucionando computação distribuída
  - RPC em grande estilo!