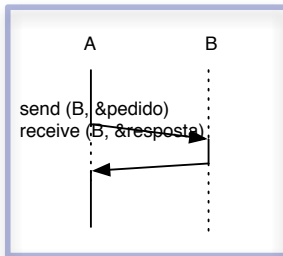


Sistemas Distribuídos

Chamada Remota de Procedimento

março de 2021

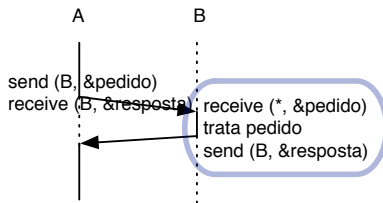
RPC: motivação



como facilitar esse padrão tão comum?

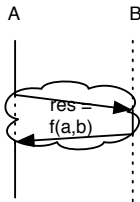
- encapsulamento de detalhes de comunicação
 - criação, envio e recebimento de mensagens
 - empacotamento de argumentos
 - tratamento de reenvio, etc

RPC: motivação



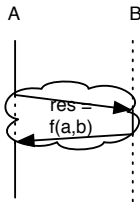
- lado do servidor também pode se beneficiar de abstração
- uma mesma função pode tratar pedido local ou remoto

RPC: abstração



- originalmente: ênfase em transparência
- programa distribuído com mesma organização que programa local
- tratamento automático de empacotamento e desempacotamento

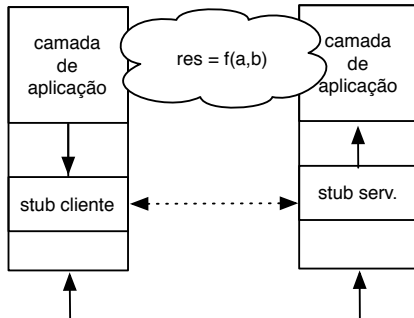
RPC: modelo de execução tradicional



- chamador permanece bloqueado até chegada de resposta
 - analogia direta com caso local
- modelo utilizado largamente em redes locais
 - servidores de arquivos
 - servidores de impressão
 - ...

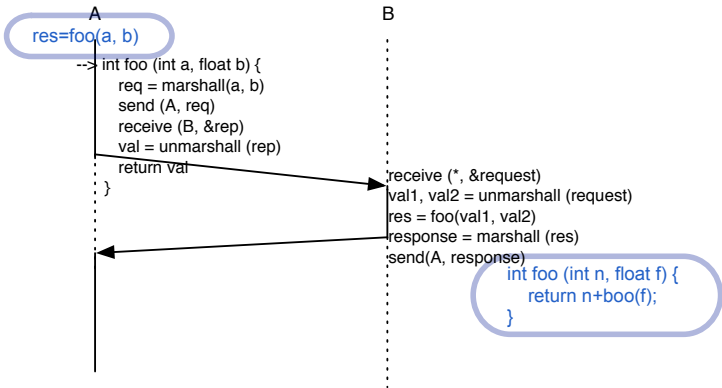
- A. Birrell and B. Nelson. 1984. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1), 39-59.

RPC: implementação

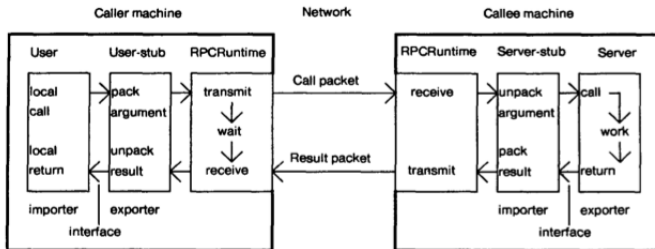


- *stubs* intermediam comunicação

RPC: implementação



RPC: implementação tradicional



- geração automática de stubs cliente e servidor a partir de *interface*
 - introdução de *IDLs*

RPC: especificação de interfaces

exemplo Sun RPC:

```
struct intpair {
    int a;
    int b;
};
program ADD_PROG {
    version ADD_VERS {
        int ADD(intpair) = 1;
    } = 1;
} = 0x23451111;
```

pré-compilador (no caso rpcgen)

- geração de stub cliente e stub servidor, com chamadas à biblioteca RPC
 - *marshalling*
 - *unmarshalling*
 - comunicação na rede
 - arquivo de interface para cliente

RPC: empacotamento de dados (1)

- problemas com representações diferentes e alinhamento de dados
- surgimento de protocolos e formatos padronizados
 - biblioteca XDR, formato ASN.1 (ISO), ...
 - codificações com tipo explícito X implícito

RPC: empacotamento de dados (2)

- referências de memória não fazem sentido
- empacotamento de estruturas complexas?
 - possibilidade do programador definir empacotamentos
- referências voltam a fazer sentido no contexto de objetos distribuídos!

binding

- problema semelhante ao de localização de destinatário de mensagem, mas agora com abstração de mais alto nível
 - uso de bases de dados centralizadas
 - uso de bases de dados por máquina

- utilização de características da linguagem
- interfaces x classes
- interface `Remote` define propriedades comuns a todos os objetos remotos usada nas declarações do cliente
- exceção `RemoteException`
- classe `UnicastRemoteObject` implementa funcionalidade básica de objeto remoto estendida pela implementação do objeto servidor
- carga dinâmica (download) de stubs e de implementações de argumentos

Java RMI – Cliente

```
package example.hello;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class Client {
    private Client() {}
    public static void main(String[] args) {
        String host = (args.length < 1) ? null : args[0];
        try {
            Registry registry = LocateRegistry.getRegistry(host);
            Hello stub = (Hello) registry.lookup("Hello");
            String response = stub.sayHello();
            System.out.println("response: " + response);
        } catch (Exception e) {
            System.err.println("Client exception: " + e.toString());
            e.printStackTrace();
        }
    }
}
```

Java RMI – Servidor

```
/* SampleServer.java */
package example.hello;
import java.rmi.*;
public class Server implements Hello {
    public Server() {}
    public String sayHello() {
        return "Hello, world!";
    }
    public static void main(String args[]) {
        try {
            Server obj = new Server();
            Hello stub = (Hello) UnicastRemoteObject.exportObject(obj, 0);
            // Bind the remote object's stub in the registry
            Registry registry = LocateRegistry.getRegistry();
            registry.bind("Hello", stub);

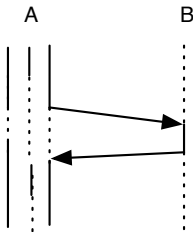
            System.err.println("Server ready");
        } catch (Exception e) {
            System.err.println("Server exception: " + e.toString());
        }
    }
}
```


RPC: falhas e semântica

- transparência preconizada inicialmente “quebra” diante da possibilidade de falhas
 - diferença para chamadas locais
- classificação em diferentes modelos: exatamente uma vez, no máximo uma vez, no mínimo uma vez
 - importância de chamadas **idempotentes**
- tratamento de exceções

RPC: concorrência

- como sobrepor tempo de chamada com processamento?
 - solução clássica: uso de threads



RPC: cenários pouco acoplados

problemas com sincronização causada pelo bloqueio

- escalabilidade de threads de OS
- coleta de lixo: threads a espera de servidores que falham
- sobrecarga de preempção

- oneway
- futuros
- chamadas assíncronas