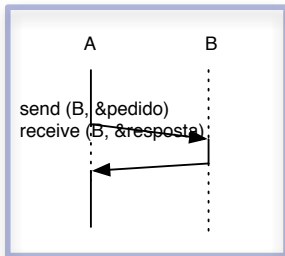


Sistemas Distribuídos

Chamada Remota de Procedimento

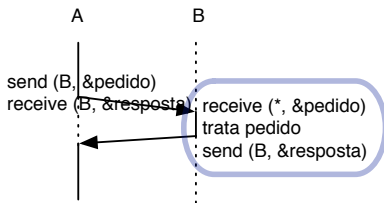
abril de 2017



como facilitar esse padrão tão comum?

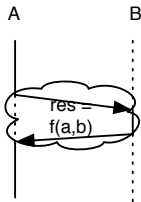
- encapsulamento de detalhes de comunicação
 - criação, envio e recebimento de mensagens
 - empacotamento de argumentos
 - tratamento de reenvio, etc

RPC: motivação



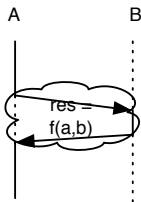
- lado do servidor também pode se beneficiar de abstração
- “serviço” escrito como uma função qualquer

RPC: abstração



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

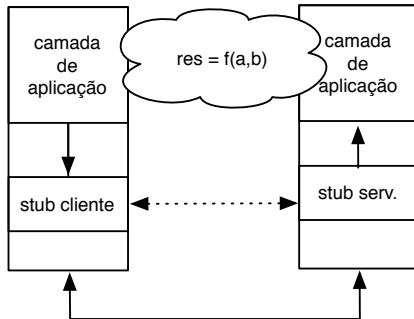
RPC: modelo de execução tradicional



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

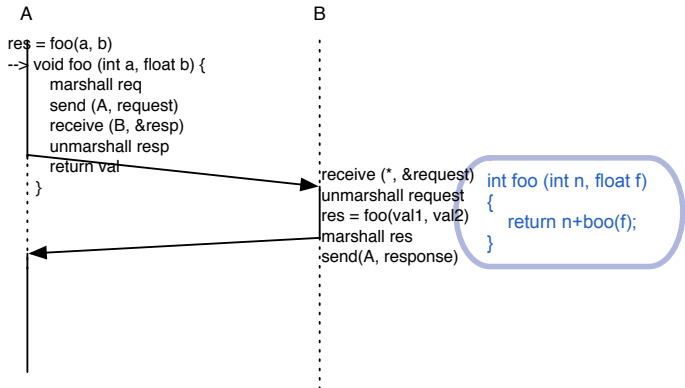
- A. Birrell and B. Nelson. 1984. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1), 39-59.

RPC: implementação

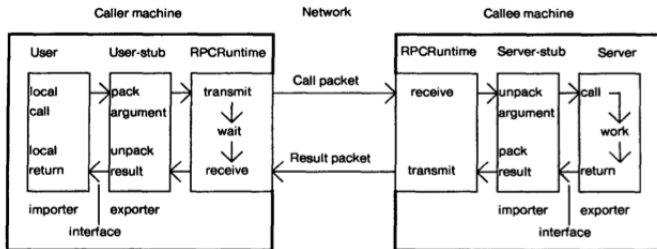


- *stubs* intermediam comunicação

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!
 - *callbacks*

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 client;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
...
public class ComputePi {
    public static void main(String args[]) {
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new SecurityManager());
        }
        try {
            String name = "Compute";
            Registry registry = LocateRegistry.getRegistry(args[0]);
            Compute comp = (Compute) registry.lookup(name);
            Pi task = new Pi(Integer.parseInt(args[1]));
            BigDecimal pi = comp.executeTask(task);
            System.out.println(pi);
        } catch (Exception e) {
            System.err.println("ComputePi exception:"); e.printStackTrace();
        }
    }
}
```


Java RMI – Servidor

```
/* SampleServer.java */  
import java.rmi.*;  
public interface SampleServer extends Remote  
{  
    public int sum(int a,int b) throws RemoteException;  
}
```

```
/* SampleServerImpl.java */  
...  
public class SampleServerImpl extends UnicastRemoteObject implements SampleServer  
{  
    SampleServerImpl() throws RemoteException  
    {  
        super();  
    }  
}
```

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

luarpc — construção dinâmica de stubs

- registerServant (idl, servantobject)
- waitIncomingRequests ()
- createProxy (idl, ip, port)

Trabalho: RPC com Lua

```
o1 = { foo = function(a, b)
    return a+b, "alo alo"
end,
    boo = function (self, z)
    self.bar, self.foo = self.foo, self.bar
end,
    bar = function(a, b)
    return a-b, "tchau tchau"
end,
}
o2 = { foo = function(m, n) ...
}
ip, p = registerServant (idl, o1)
print ("sou 1, estou esperando reqs na
porta " .. p)
ip, p = registerServant (idl, o2)
print ("sou 2, estou esperando reqs na
porta " .. p)
waitIncoming()
```

```
rep1 = createProxy (idlserv, ip, porta)
rep2 = createProxy(idlserv, ip, outraporta)
...
print (rep1:foo(4,5))
rep1:boo()
print (rep2:foo(x,y))
```

- tanto cliente como servidor são single-threaded
- servidor deve poder receber pedidos para qualquer servente

Trabalho: RPC com Lua

```
o1 = { foo = function(a, b)
    return a+b, "alo alo"
end,
    boo = function (self, z)
    self.bar, self. foo = self.foo, self.bar
end,
    bar = function(a, b)
    return a-b, "tchau tchau"
end,
}
o2 = {f = ...}
ip, p = registerServant (idl, o1)
print ("estou esperando reqs para xxx na porta " .. p)
ip, p = registerServant (outraidl, o2)
print ("estou esperando reqs para yyy na porta " .. p)
waitIncoming()
```

```
rep = createProxy (idl1, ip, porta)
...
print (rep:foo(4,5))
rep:boo()
```

```
p = createProxy (idl2, ip, porta)
...
p:f()
...
```

```
interface { name = minhaInt,  
            methods = {  
                foo = {  
                    resulttype = "double",  
                    args = {{direction = "in",  
                             type = "double"},  
                           {direction = "in",  
                             type = "double"},  
                           {direction = "out",  
                             type = "string"},  
                          }  
                },  
                boo = {  
                    resulttype = "void",  
                    args = {{ direction = "inout",  
                             type = "double"},  
                          }  
                }  
            }  
}
```

trecho extraído de trabalho anterior

```
function creatercproxy(hostname, port, interface)
local functions = {}
local prototypes = parser(interface)
for name,sig in pairs(prototypes) do
  functions[name] = function(...) -- !!!
    -- validating params
    local params = {...}
    local values = {name}
    local types = sig.input
    for i=1,#types do
      if (#params >= i) then
        values[#values+1] = params[i]
        if (type(params[i])~="number") then
          values[#values] = "\"" .. values[#values] .. "\""
        end
        ...
      end
    end
    -- creating request
    local request = pack(values)
    -- creating socket
    local client = socket.tcp()
    ...
    local conn = client:connect(hostname, port)
    ...
    local result = client:send(request .. '\n')
    ...
  end
end
return functions;
end
```

- proposta original simula chamada local: bloqueio da linha executora