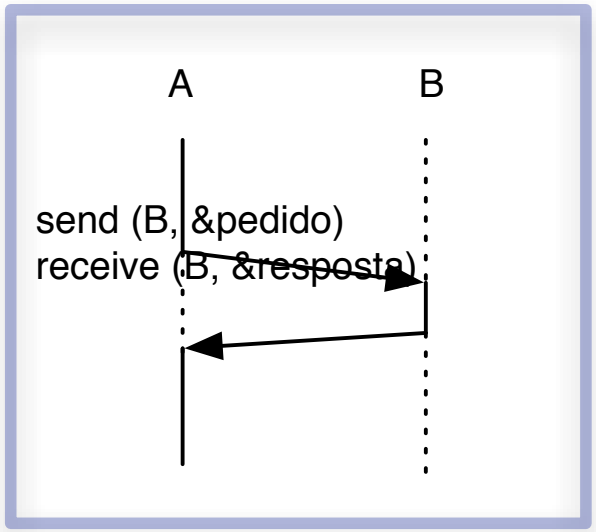
Sistemas Distribuídos

Chamada Remota de Procedimento

março de 2015

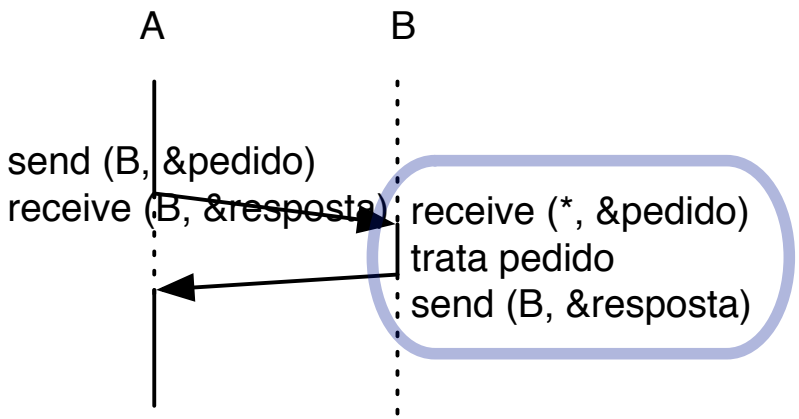
RPC: motivação



como facilitar esse padrão tão comum?

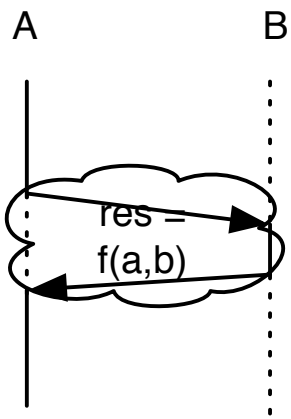
- encapsulamento de detalhes de comunicação
  - criação, envio e recebimento de mensagens
  - empacotamento de argumentos
  - tratamento de reenvio, etc

RPC: motivação



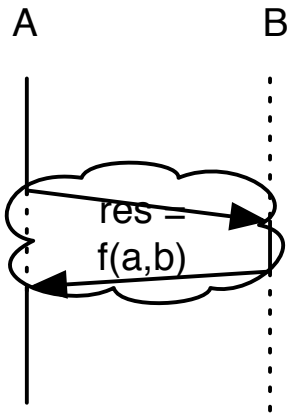
- lado do servidor também pode se beneficiar de abstração
- uma mesma função pode tratar pedido local ou remoto

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

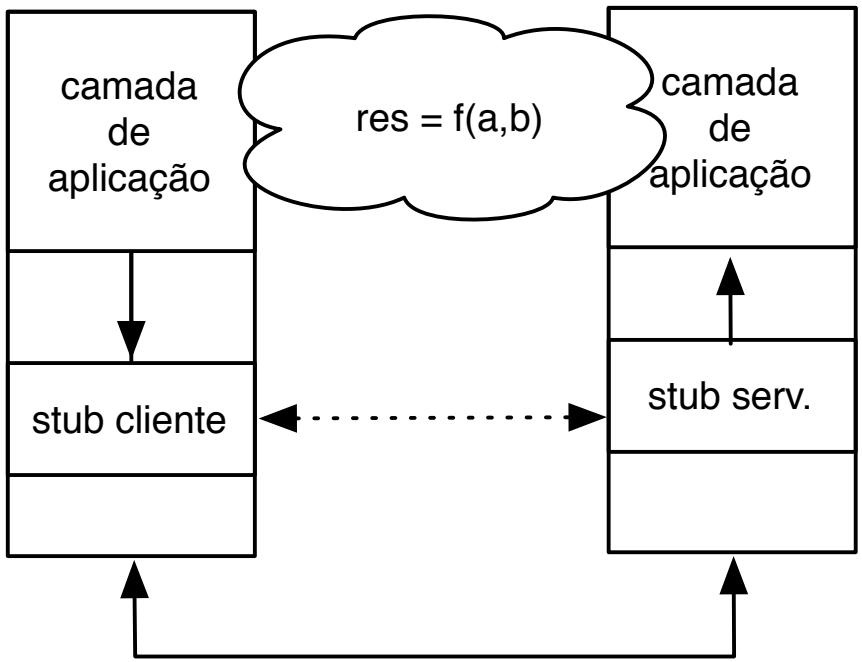


- chamador permanece bloqueado até chegada de resposta
  - analogia direta com caso local
- modelo utilizado largamente em redes locais
  - servidores de arquivos
  - servidores de impressão
  - ...

RPC – referência clássica

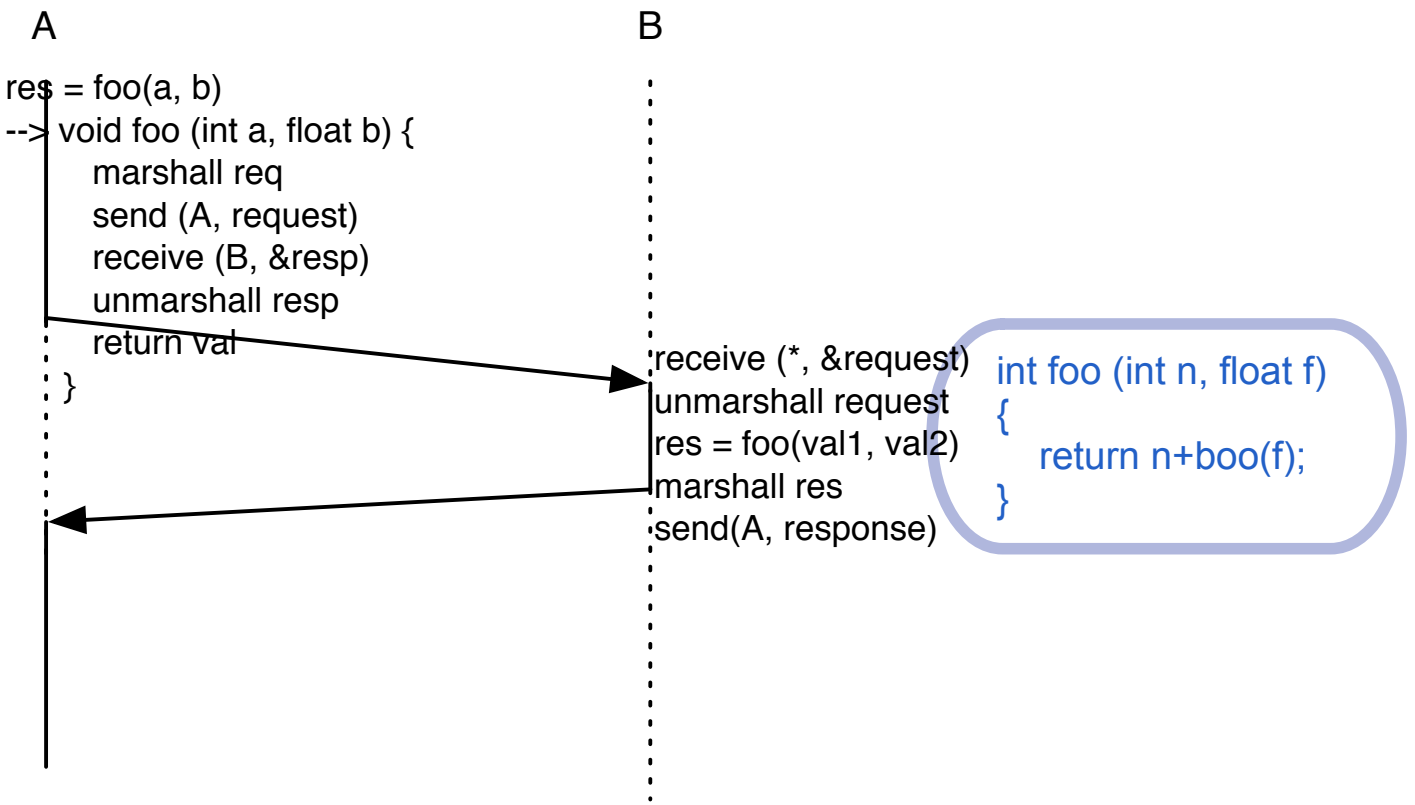
- A. Birrell and B. Nelson. 1984. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1), 39-59.

RPC: implementação

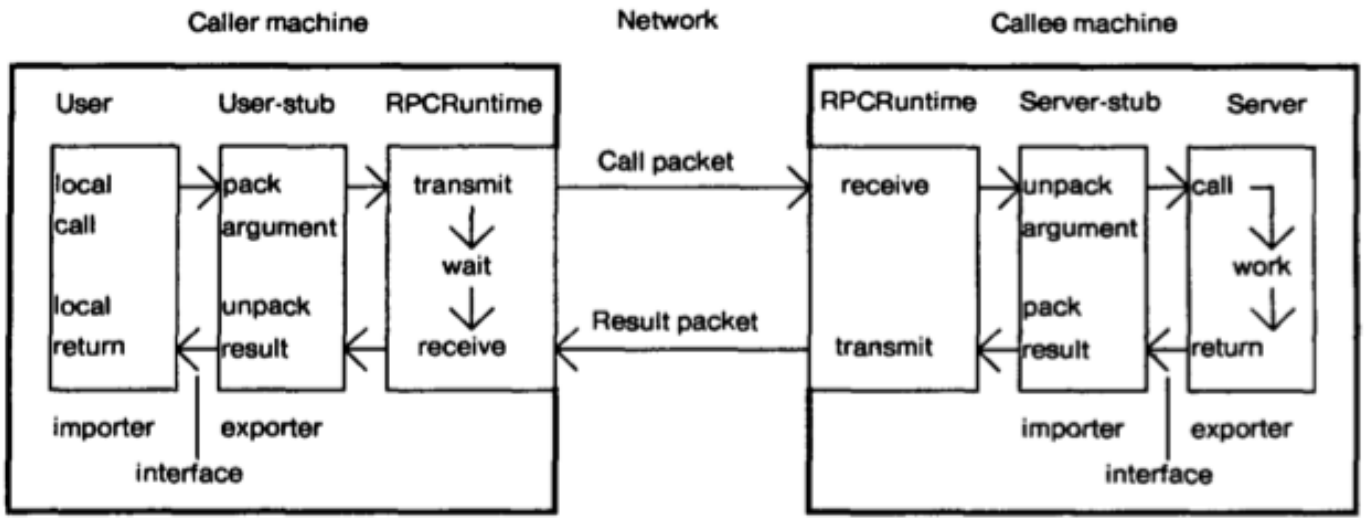


- *stubs* intermediam comunicação

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!

Localização de Servidores

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

Java RMI

- 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

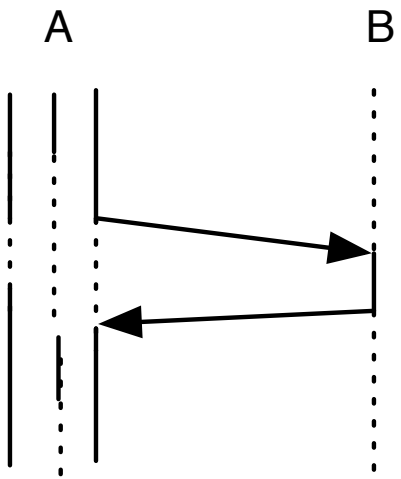
```
/* SampleServer.java */
import java.rmi.*;
public interface SampleServer extends Remote
{
  public int sum(int a,int b) throws RemoteException;
}

/* SampleServerImpl.java */
...
public class SampleServerImpl extends UnicastRemoteObject implements SampleServ
{
  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

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

RPC e assincronismo

- oneway
- futuros
- chamadas assíncronas

Futuros

- chamada retorna imediatamente devolvendo um descritor
- descritor usado posteriormente para sincronização
- muito popular atualmente!

Futuros em C++11

```
#include <future>
#include <iostream>
#include <vector>
int twice(int m) {
  return 2 * m;
}
int main() {
  std::vector<std::future<int>> futures;
  for(int i = 0; i < 10; ++i) {
    futures.push_back (std::async(twice, i));
  }
  //retrive and print the value stored in the future
  for(auto &e : futures) {
    std::cout << e.get() << std::endl;
  }
  return 0;
}
```

Futuros em ProActive

```
m1 = m0.getBlock (0, 0, m, n-1);
m2 = m0.getBlock (m+1, 0, n-1, n-1);

m1=(Matrix) Javall.turnActive(m1, remoteNode);
m2=(Matrix) Javall.turnActive(m2, localNode);

// Computes both right products
v1 = m1.rightProduct (v0);
v2 = m2.rightProduct (v0);

// Creates result vector
v3 = v1.concat (v2);
```

Futuros e avaliação postergada

- objetos retornados por operações assíncronas podem ser passados como argumentos em novas operações
- otimização da transferência de dados

```
v1 = a.foo (...); // chamada assíncrona
v2 = a.bar(...); // chamada assíncrona
...
v1.f(v2)
```


RPC assíncrona

- chamada retorna imediatamente
- retorno dispara execução de *callback*
  - em alguns casos, *callback* especificada na chamada
- casamento com modelo de execução em uso

Chamada assíncrona "em Lua"

```
function collect(val)
  acc = acc + val
  repl = repl + 1
  if (repl==expected) then print ("Current Value: ",
                                  acc/repl)
  end
end
function askvals (peers)
  repl = 0; expected = 0; acc = 0
  for p in pairs (peers) do
    expected = expected + 1
    p:currValue{callback=collect}
  end
end
```

- estado registrado em globais
- e se novo pedido for realizado antes do primeiro estar completo?

Chamada assíncrona "em Lua" com closures

```
function request(peers)
  local acc, repl = 0, 0
  local expected = table.getn(peers)
------------
  function avrg (val)
    repl = repl+1
    acc = acc + val
    if (repl==expected) then print ("Current Value: ",
                                    acc/repl)
    end
  end
-------------
  for _,p in ipairs (peers) do
    rpc.async(p, "currValue", avrg)()
  end
end
```

RPC: outras críticas

acoplamento cliente-servidor também espacial

- identificação de servidor que deve tratar a requisição
- endereço bem conhecido funciona bem em ambientes controlados

Trabalho: RPC com Lua

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()
...
```

RPC com Lua — IDL

```
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 createrpcproxy(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
      -- 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
```