

Como Implementar o Middleware?

Parte II

Nelson Rosa – nsr@cin.ufpe.br



UNIVERSIDADE
FEDERAL
DE PERNAMBUCO

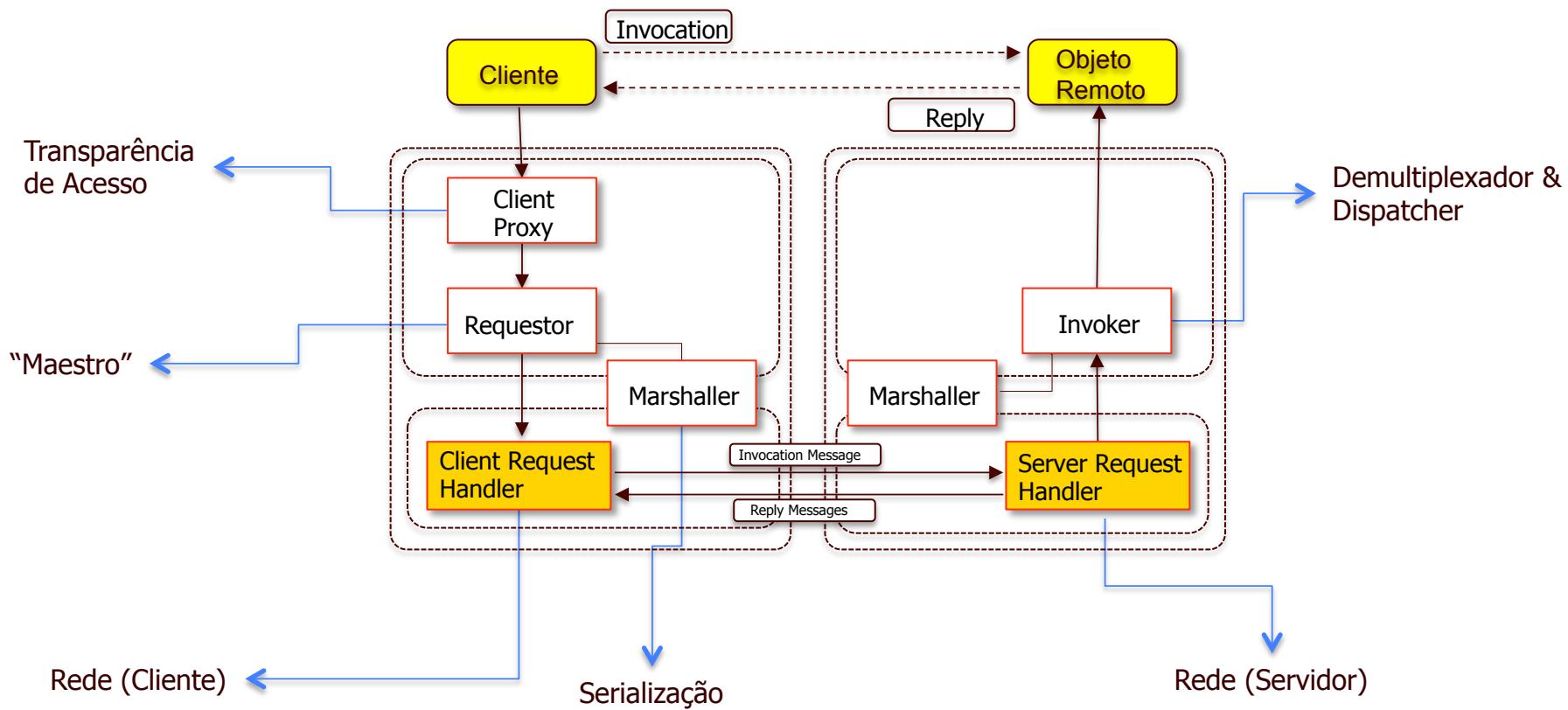
CIn.ufpe.br

Objetivos

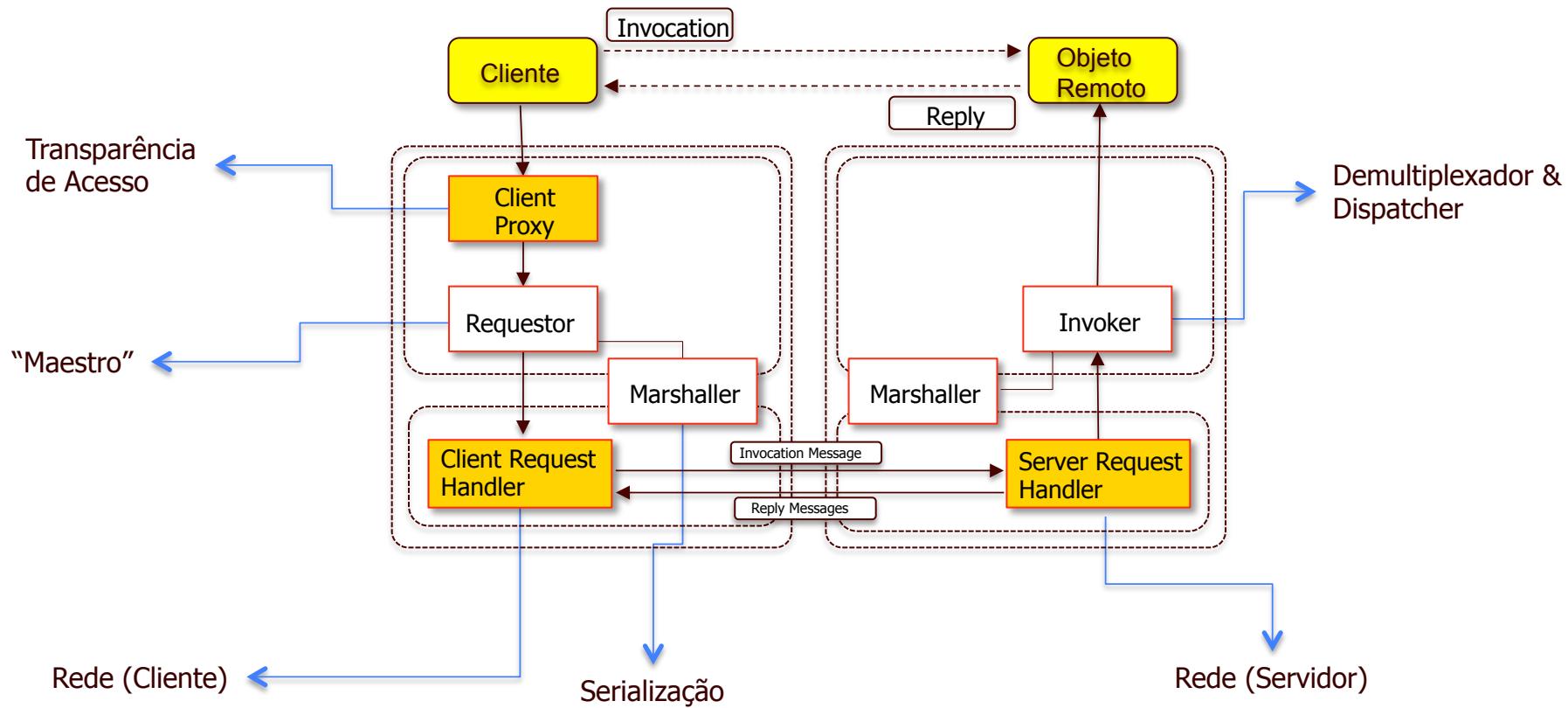
- Apresentar Remoting Patterns da Camada de Distribuição
- Discutir estratégias de implementação destes padrões.

Client Proxy

Remoting Patterns



Remoting Patterns



Client Proxy

■ Contexto

- O desenvolvedor da aplicação distribuída já possui um *Requestor* para acessar os objetos remotos

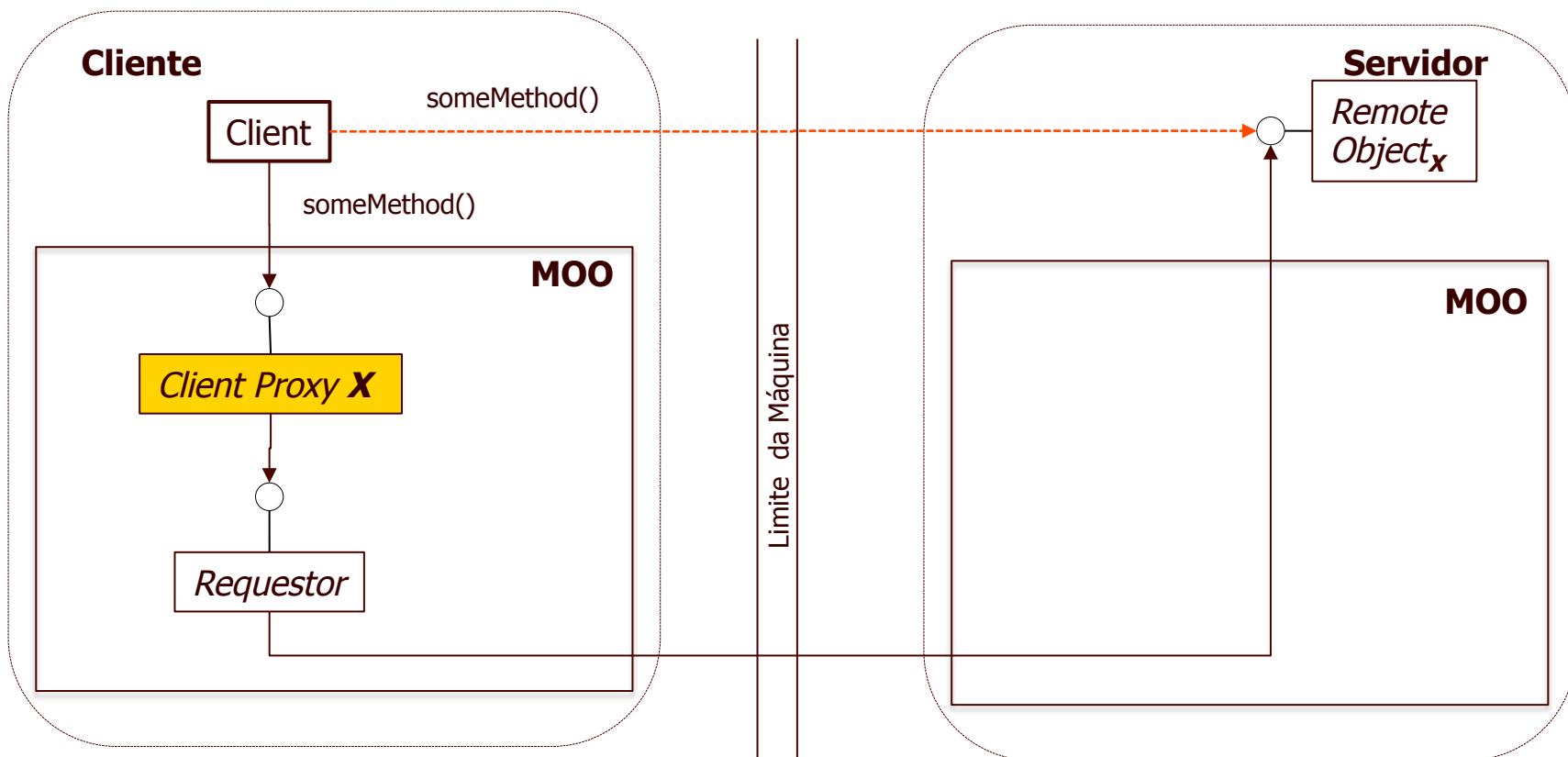
■ Problema

- Como permitir que objetos remotos sejam acessados da mesma forma que objetos locais (**transparência de acesso**)?
- Considerando que...
 - ... o *Requestor* esconde detalhes da comunicação, mas não fornece checagem de tipos estática (o *Requestor* é “type-independent”).

■ Solução

- Usar um *Client Proxy* que possui a mesma interface do objeto remoto.

Client Proxy



► *Client Proxy* é **específico** para cada tipo de objeto remoto.

► *Client Proxy* é **gerado** [manualmente/automaticamente] a partir da interface do objeto.

Client Proxy:: Calculadora

Cliente

Client

add(1,3)

MOO

CalculatorProxy

Invoke(...)

Requestor

marshaller.go x client.go x

```
1 package main
2
3 import ...
7
8 func main() {...}
```

```
1 package clientproxy
2
3 import ...
4
5 type CalculatorProxy struct {...}
6
7 func NewCalculatorProxy() CalculatorProxy {...}
8
9 func (proxy CalculatorProxy) Add(p1 int, p2 int) int {...}
10
11 func (proxy CalculatorProxy) Sub(p1 int, p2 int) int {...}
12
13 func (proxy CalculatorProxy) Mul(p1 int, p2 int) int {...}
14
15 func (proxy CalculatorProxy) Div(p1 int, p2 int) int {...}
```

```
1 package requestor
2
3 import ...
4
5 type Requestor struct{}
6
7 func (Requestor) Invoke(inv shared.Invocation) interface{} {...}
```

Client Proxy:: Calculadora

```
1 package clientproxy  
2  
3 type ClientProxy struct {  
4     Host      string  
5     Port      int  
6     Id       int  
7     TypeName string  
8 }
```

- ✓ Informações registradas no **Serviço de Nomes** pelo **Servidor** da Calculadora.
- ✓ Informações obtidas pelo **Cliente** ao acessar o **Serviço de Nomes**.
- ✓ Informações usadas a cada invocação da **Calculadora**.

Client Proxy:: Calculadora

```
1 package proxies
2
3 +import ...
10
11 +type CalculatorProxy struct {...}
14
15 +func NewCalculatorProxy() CalculatorProxy {...}
24
25 +func (proxy CalculatorProxy) Add(p1 int, p2 int) int {...}
40
41 +func (proxy CalculatorProxy) Sub(p1 int, p2 int) int {...}
56
57 +func (proxy CalculatorProxy) Mul(p1 int, p2 int) int {...}
72
73 +func (proxy CalculatorProxy) Div(p1 int, p2 int) int {...}
88
```

Client Proxy:: Calculadora

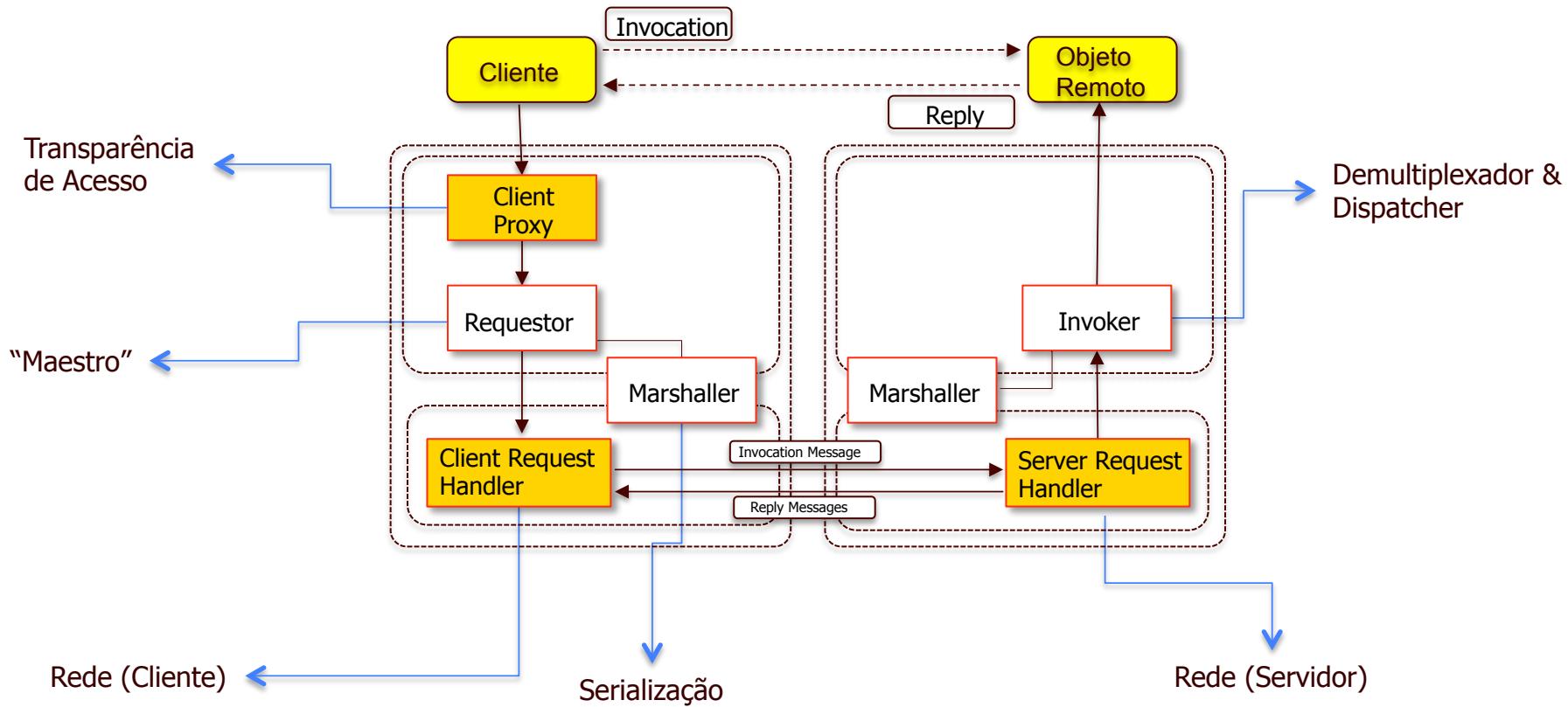
```
1 package proxies
2
3 import ...
4
5 type CalculatorProxy struct {...}
6
7 func NewCalculatorProxy() CalculatorProxy {...}
8
9 func (proxy CalculatorProxy) Add(p1 int, p2 int) int {
10
11     // prepare invocation
12     params := make([]interface{}, 2)
13     params[0] = p1
14     params[1] = p2
15     request := aux.Request{Op:"Add", Params:params}
16     inv := aux.Invocation{Host:proxy.Proxy.Host, Port:proxy.Proxy.Port, Request:request}
17
18     // invoke requestor
19     req := requestor.Requestor{}
20     ter := req.Invoke(inv).([]interface{})
21
22     return int(ter[0].(float64))
23 }
```

- Estes passos se repetem para cada método do Client Proxy.
- Parâmetros das operações podem mudar de método para método.

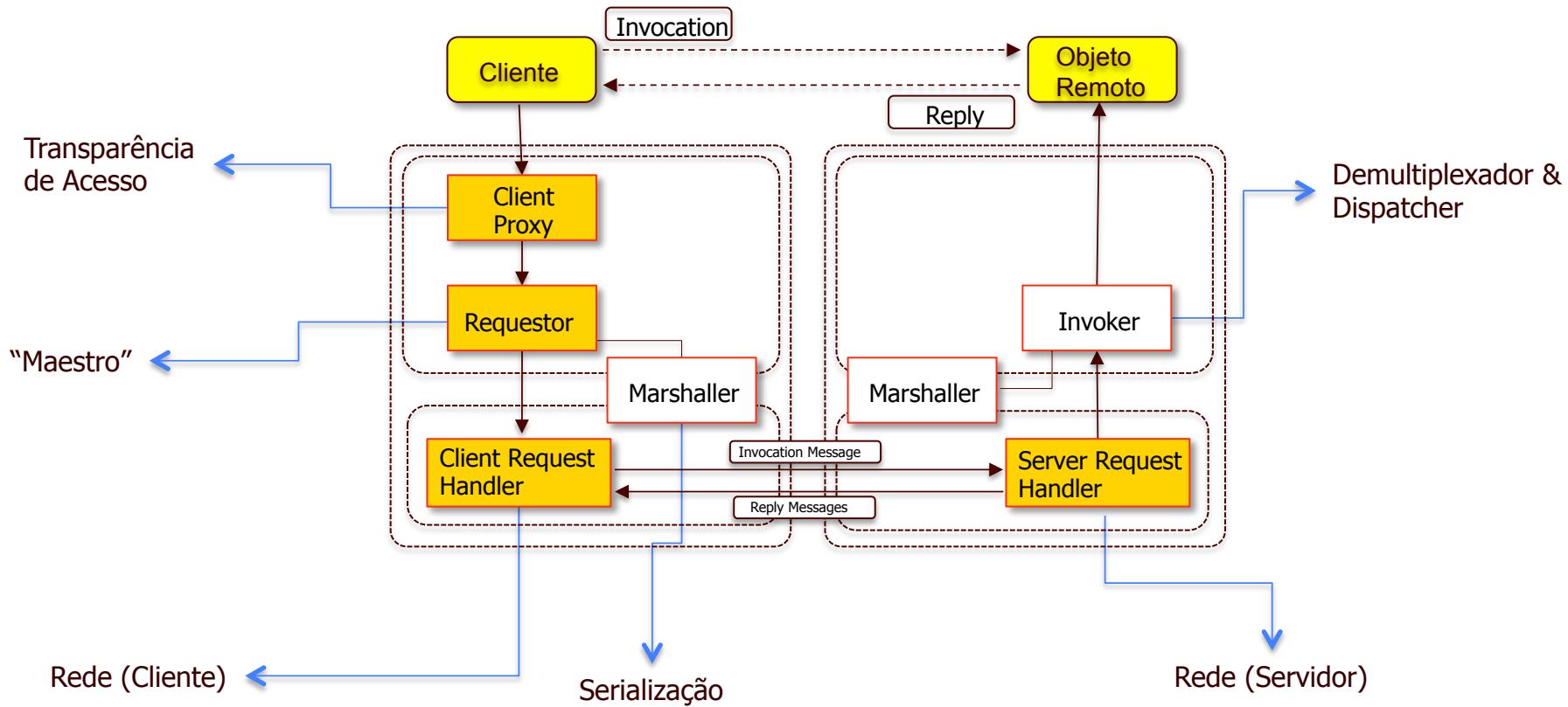
1. Recebe informações do **Cliente**
2. Prepara a invocação ao **Requestor**
3. Invoca o **Requestor** (síncrono)
4. Envia resposta ao **Cliente**

Requestor

Remoting Patterns



Remoting Patterns



Requestor

■ Contexto

- O cliente precisa acessar um ou mais objetos remotos no servidor

■ Problema

- **Como o cliente pode invocar uma operação de um objeto remoto?**
- Considerando que: na rede só passa **sequência de bytes**, é preciso estabelecer uma **conexão**, a invocação precisa ser enviada ao objeto **remoto**, precisa receber o **resultado** da invocação, precisa tratar **erros**

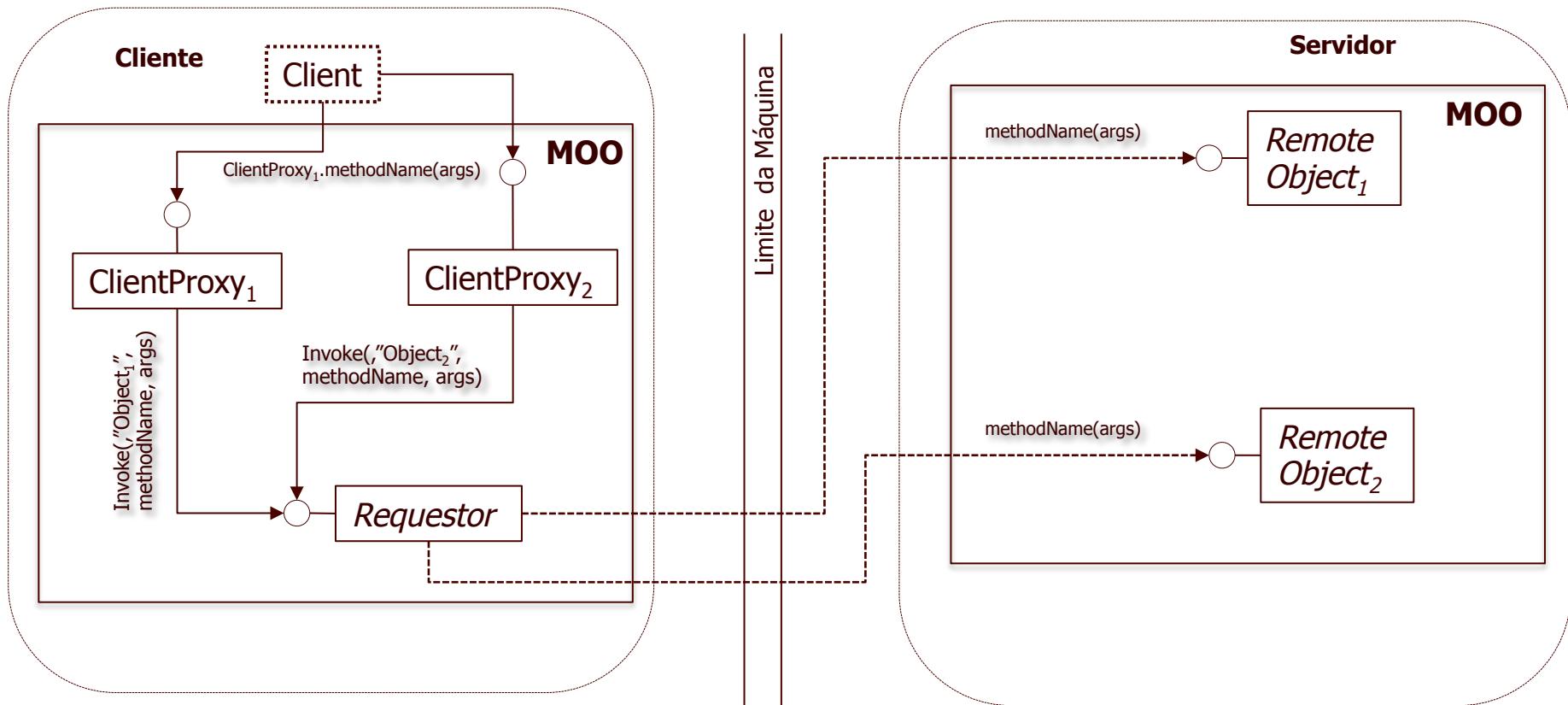
■ Solução 1

- O desenvolvedor pode incorporar a solução destes problemas ao próprio código da aplicação distribuída.

■ Solução 2

- Usar um *Requestor* para acessar o objeto remoto.

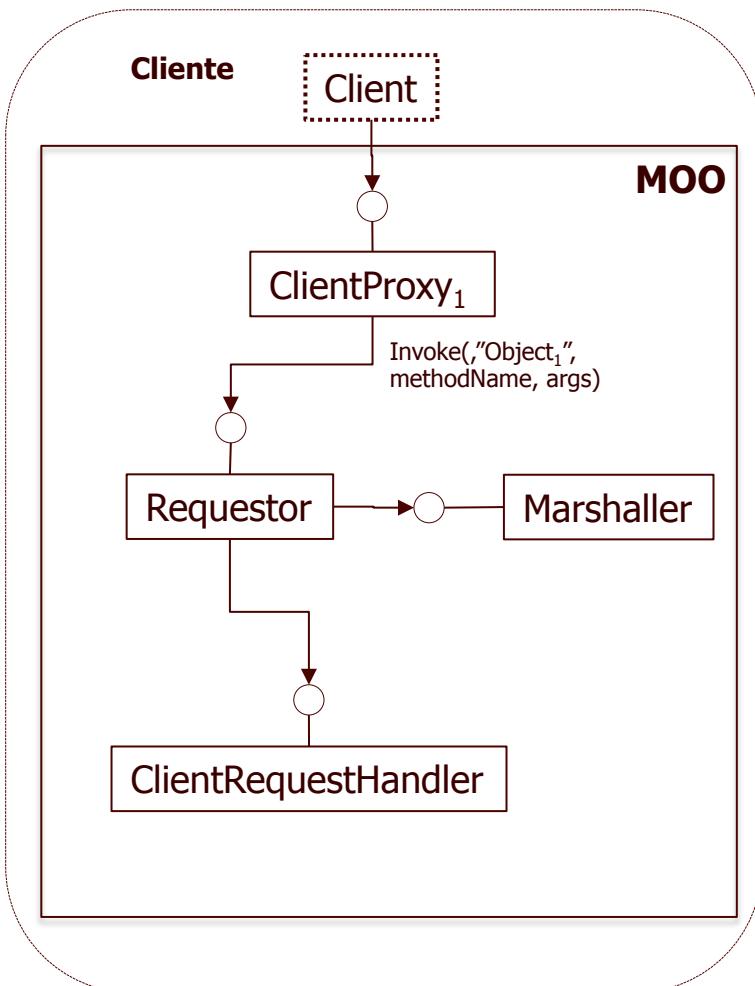
Requestor



► **Requestor** é independente de detalhes do objeto remoto, e.g., tipo, operações.

► Um **Requestor** por Cliente que trata todas as invocações remotas.

Requestor



```
1 package requestor
2
3 import ...
4
5 type Requestor struct{}
6
7 func (Requestor) Invoke(inv aux.Invocation) interface{} {...}
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
```

↑

```
1 package marshaller
2
3 import ...
4
5 type Marshaller struct{}
6
7 func (Marshaller) Marshall(msg miop.Packet) []byte {...}
8
9
10
11
12
13
14
15
16
17
18
19
20
21
```

↓

```
1 package crh
2
3 import ...
4
5
6 type CRH struct {
7     ServerHost string
8     ServerPort int
9 }
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
```

↓

```
1 package requestor
2
3 import ...
4
5 type Requestor struct{}
6
7 func (Requestor) Invoke(inv aux.Invocation) interface{} {...}
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
```

Requestor

```
package requestor

import ...

type Requestor struct{}

func (Requestor) Invoke(inv aux.Invocation) interface{} {
    marshallerInst := marshaller.Marshaller{}
    crhInst := crh.CRH{ServerHost:inv.Host, ServerPort:inv.Port}

    // create request packet
    reqHeader := miop.RequestHeader{Context:"Context",RequestId:1000,ResponseExpected:true,ObjectKey:2000,Operation:inv.Request.Op}
    reqBody := miop.RequestBody{Body:inv.Request.Params}
    header := miop.Header{Magic:"MIOP",Version:"1.0",ByteOrder:true,MessageType:shared.MIOP_REQUEST}
    body := miop.Body{ReqHeader:reqHeader,ReqBody:reqBody}
    miopPacketRequest := miop.Packet{Hdr:header,Bd:body}

    // serialise request packet
    msgToClientBytes := marshallerInst.Marshall(miopPacketRequest)

    // send request packet and receive reply packet
    msgFromServerBytes := crhInst.SendReceive(msgToClientBytes)
    miopPacketReply := marshallerInst.Unmarshall(msgFromServerBytes)

    // extract result from reply packet
    r := miopPacketReply.Bd.RepBody.OperationResult

    return r
}
```

- Estes passos são independentes da aplicação.
- Passos 3 e 4 implementam um protocolo Request/Reply.

1. Cria a **mensagem** a ser transmitida

2. **Serializa** a mensagem de request

3. **Envia** a mensagem

4. **Recebe** a mensagem de resposta

5. **Deserializa** a mensagem de resposta

6. Envia resposta ao **Client Proxy**

Requestor

```
package requestor

import ...

type Requestor struct{}

func (Requestor) Invoke(inv aux.Invocation) interface{} {
    marshallerInst := marshaller.Marshaller{}
    crhInst := crh.CRH{ServerHost:inv.Host, ServerPort:inv.Port}

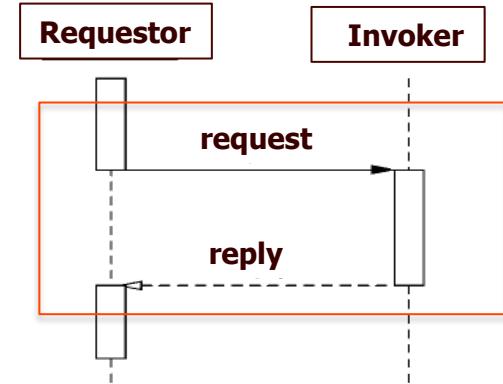
    // create request packet
    reqHeader := miop.RequestHeader{Context:"Context",RequestId:1000,ResponseExpected:true,ObjectKey:2000,Operation:inv.Request.Op}
    reqBody := miop.RequestBody{Body:inv.Request.Params}
    header := miop.Header{Magic:"MIOP",Version:"1.0",ByteOrder:true,MessageType:shared.MIOP_REQUEST}
    body := miop.Body{ReqHeader:reqHeader,ReqBody:reqBody}
    miopPacketRequest := miop.Packet{Hdr:header,Bd:body}

    // serialise request packet
    msgToClientBytes := marshallerInst.Marshall(miopPacketRequest)

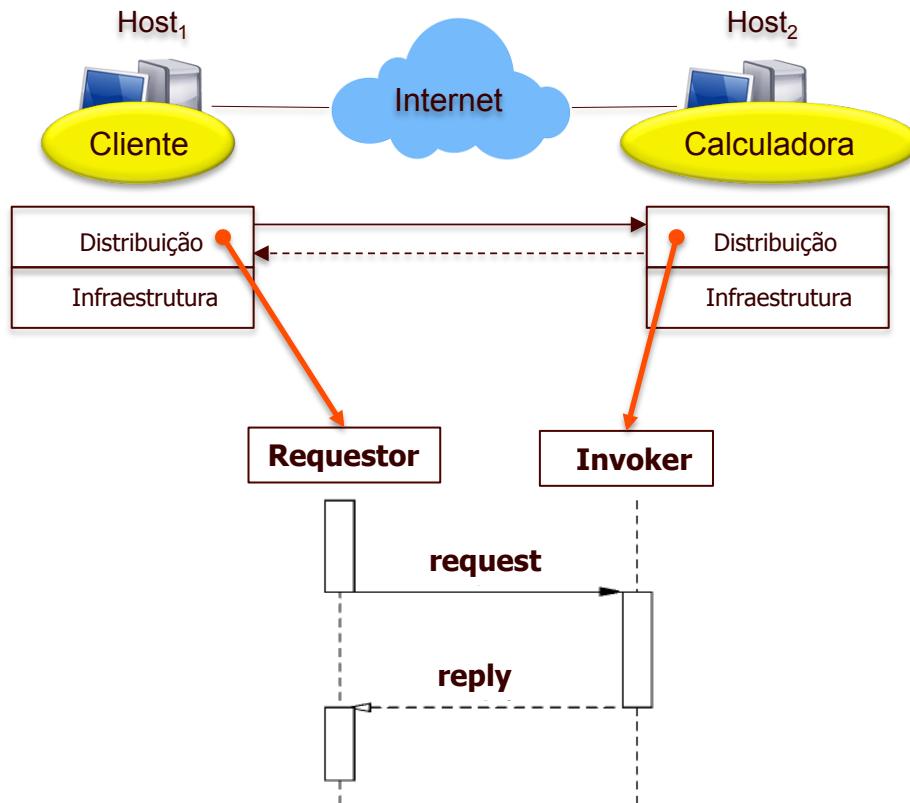
    // send request packet and receive reply packet
    msgFromServerBytes := crhInst.SendReceive(msgToClientBytes)
    miopPacketReply := marshallerInst.Unmarshall(msgFromServerBytes)

    // extract result from reply packet
    r := miopPacketReply.Bd.RepBody.OperationResult

    return r
}
```



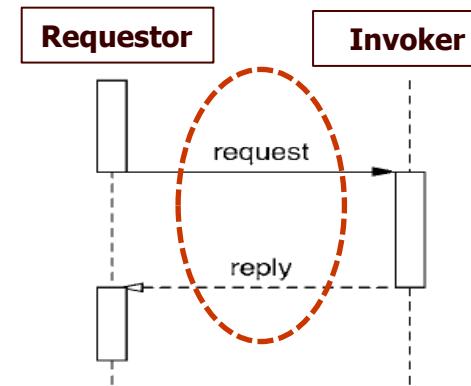
Requestor



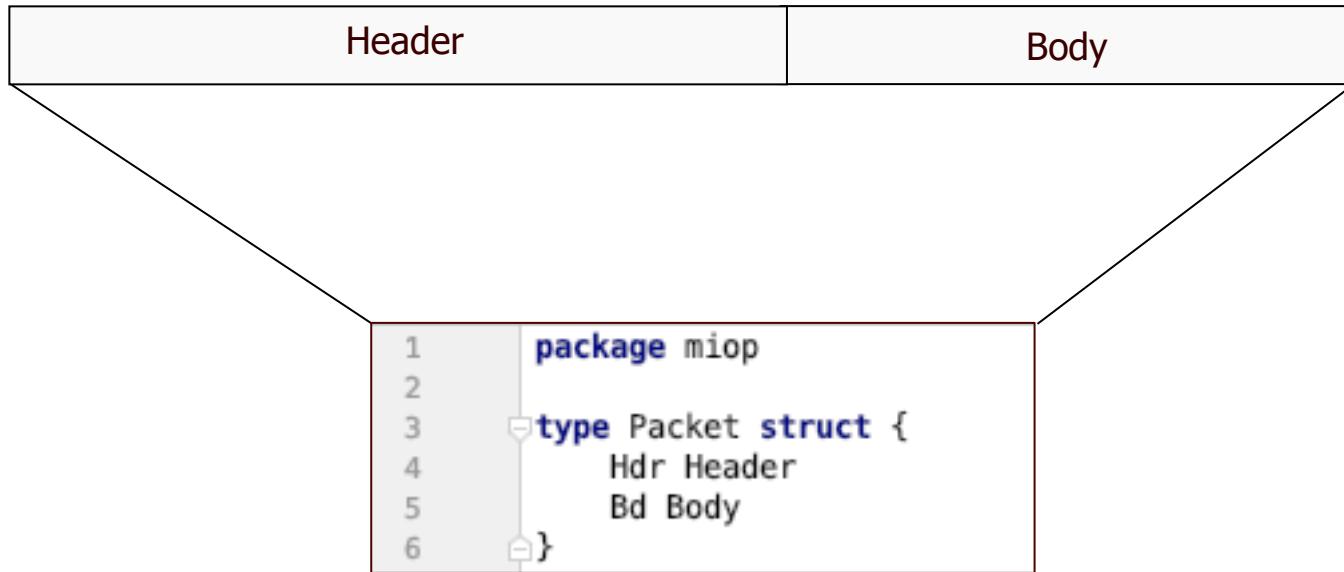
Requestor:: Protocolo

■ Características do protocolo

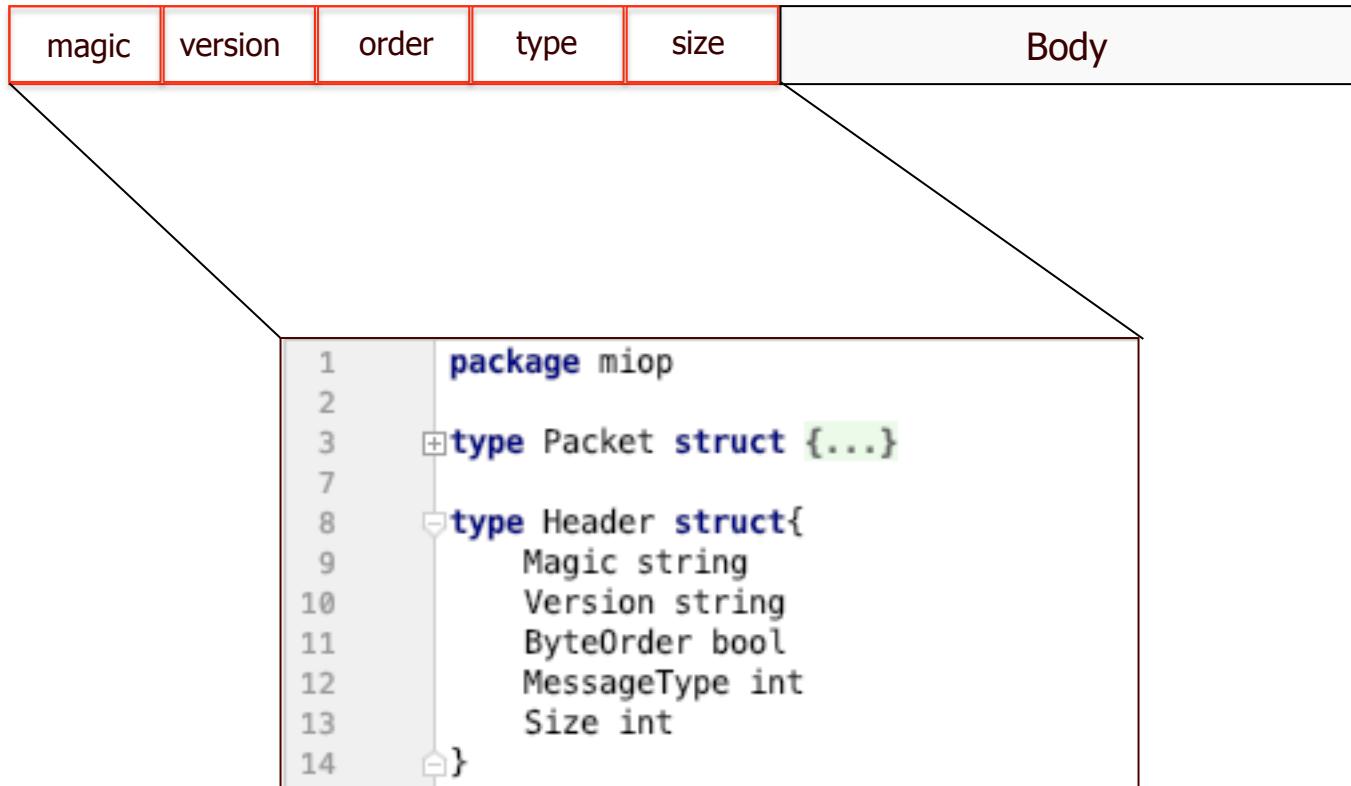
- baseado em uma **camada de transporte confiável**
- permite a **interação entre objetos**
- deve ser **independente** da tecnologia de rede do transmissor/receptor
- o transmissor determina a regra de codificação (e.g., big endian, little endian)



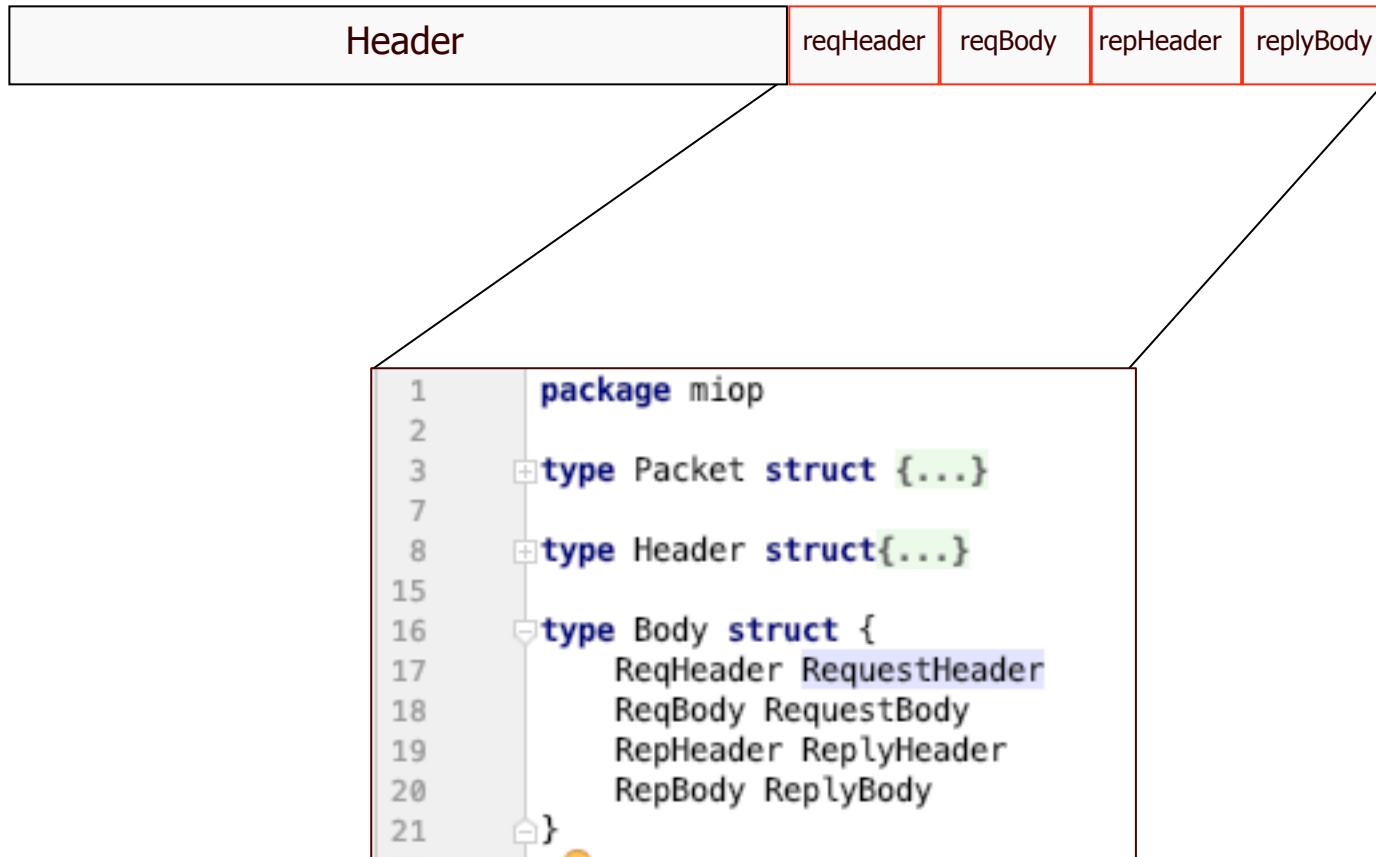
Requestor:: Protocol:: Formato



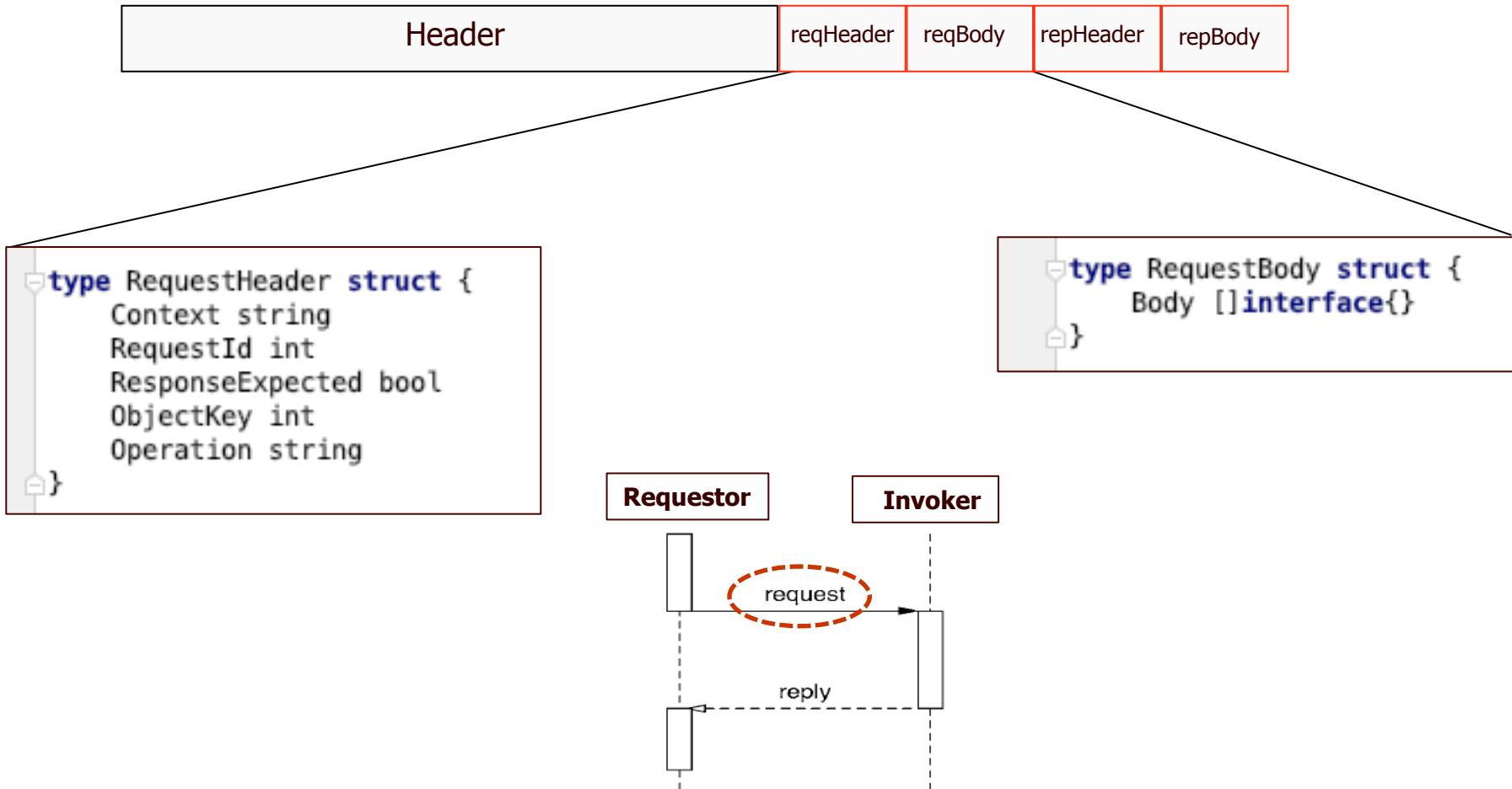
Requestor:: Protocolo:: Formato



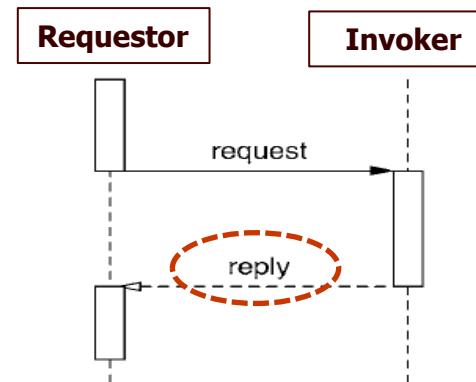
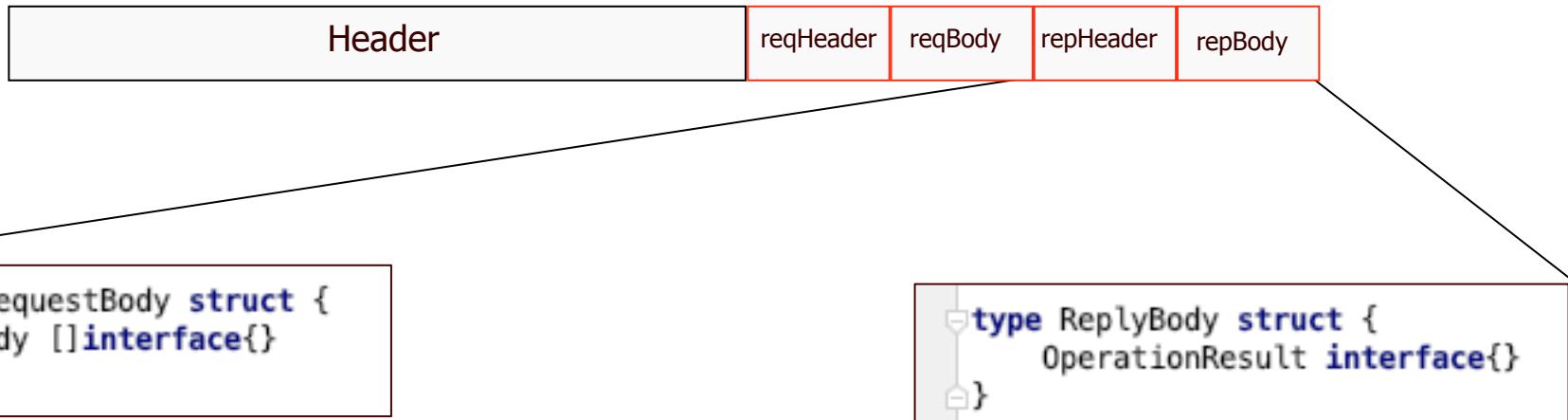
Requestor:: Protocolo:: Formato



Requestor:: Protocolo:: Formato

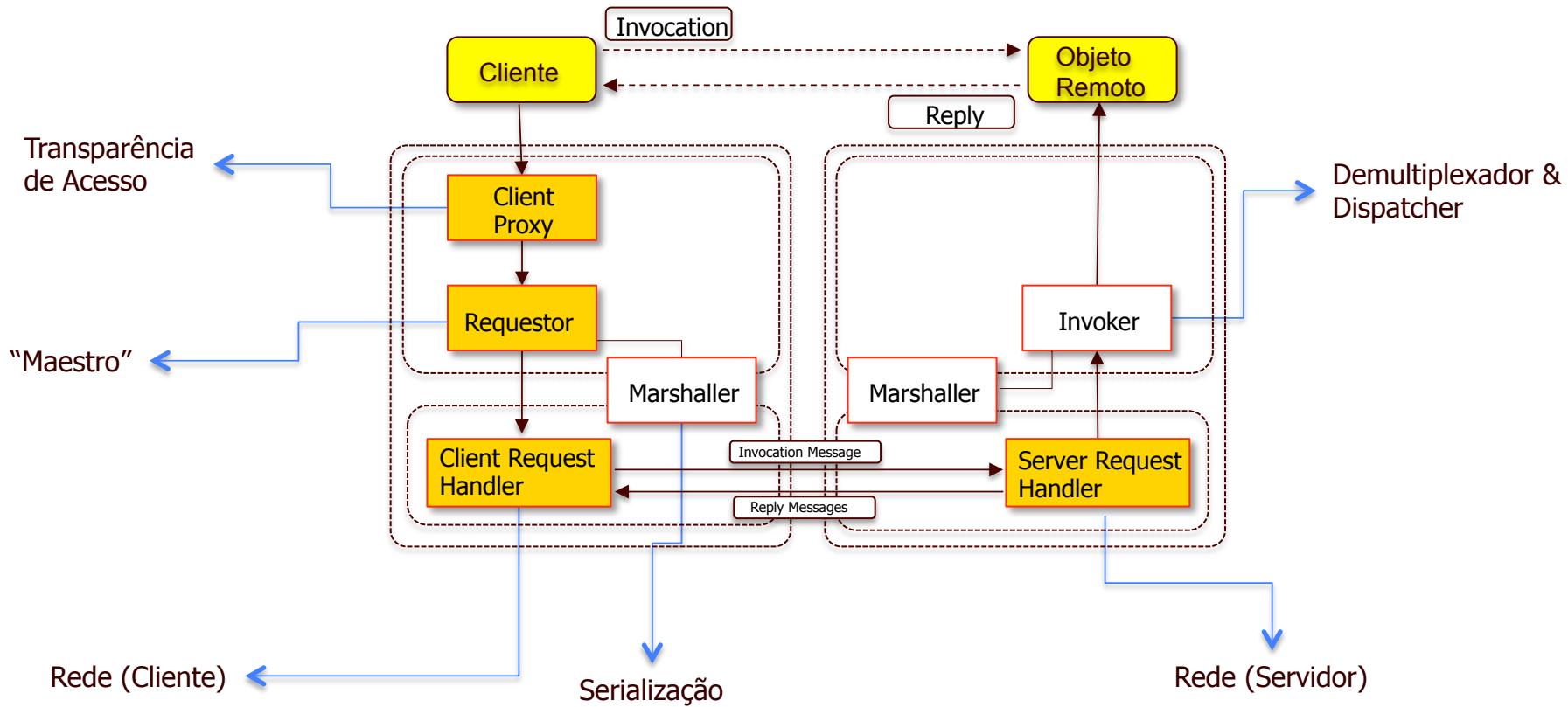


Requestor:: Protocolo:: Formato

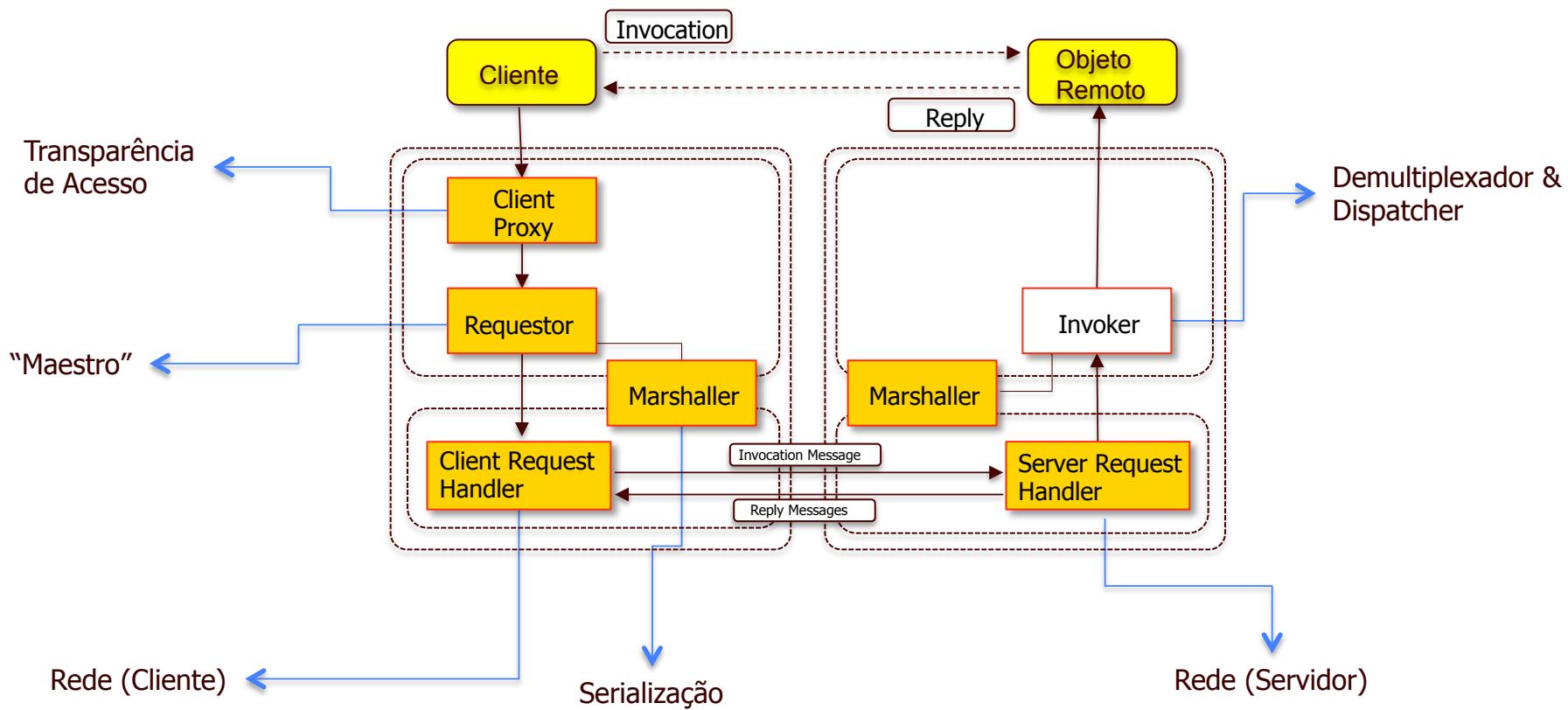


Marshaller

Remoting Patterns



Remoting Patterns



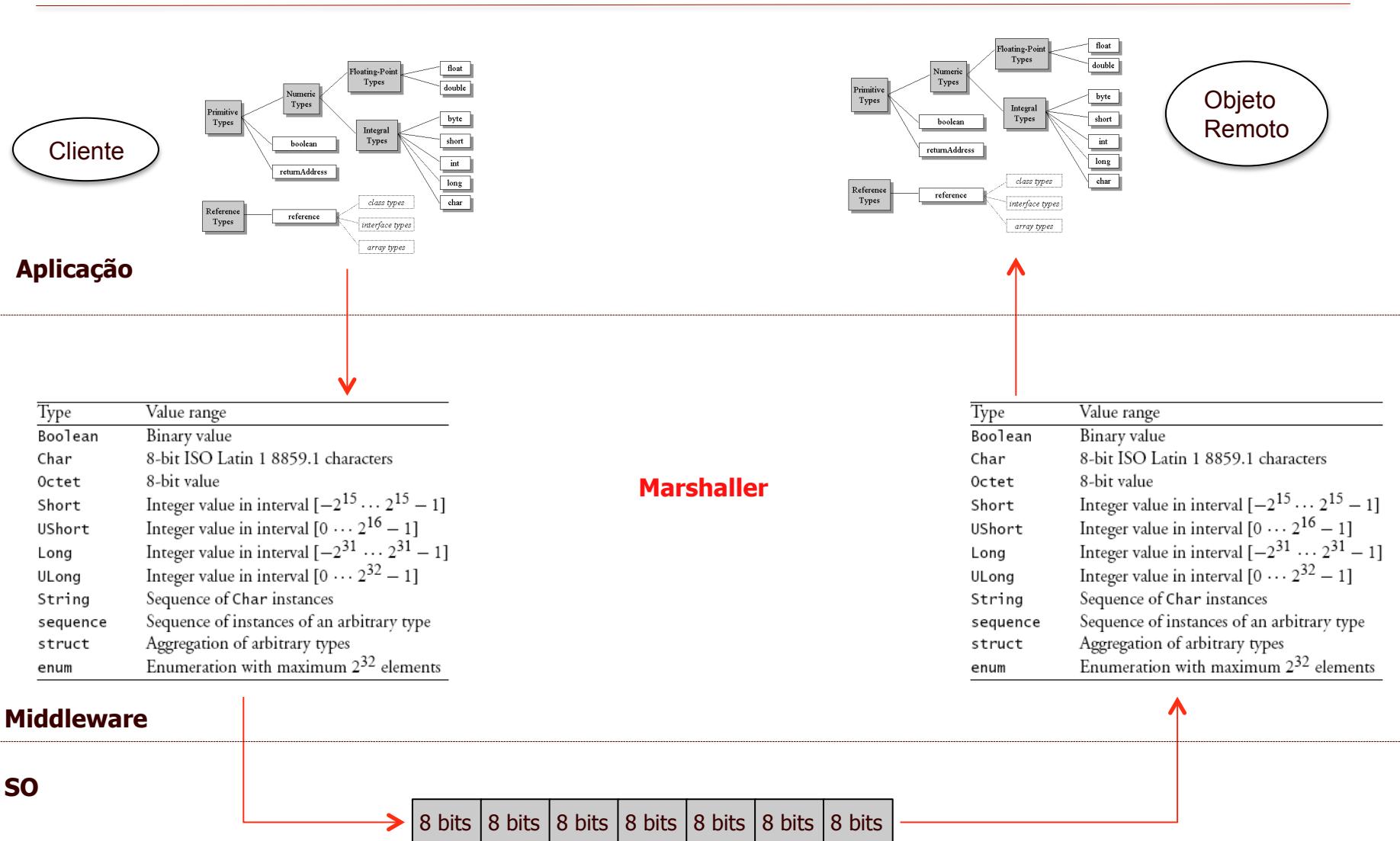
Marshaller

- **Duas funções básicas**
 1. **Mapear** tipos das linguagens em tipos usados no middleware, e.g., regras de mapeamento contidas no CORBA CDR (*Common Data Representation*)
 2. **Serializar** dados , i.e., transformar os dados em uma sequência de bytes que pode ser transmitida pela rede
- **Os tipos usados no middleware são aqueles definidos na IDL (Interface Definition Language)**
 - Se os tipos da IDL forem os mesmos das linguagens de implementação do Cliente/Servidor, **o Passo 1 não é necessário.**
- **Deve ser capaz de serializar diferentes tipos de dados**

Marshaller:: Exemplos de Tipos da IDL

Type	Value range
Boolean	Binary value
Char	8-bit ISO Latin 1 8859.1 characters
Octet	8-bit value
Short	Integer value in interval $[-2^{15} \dots 2^{15} - 1]$
UShort	Integer value in interval $[0 \dots 2^{16} - 1]$
Long	Integer value in interval $[-2^{31} \dots 2^{31} - 1]$
ULong	Integer value in interval $[0 \dots 2^{32} - 1]$
String	Sequence of Char instances
sequence	Sequence of instances of an arbitrary type
struct	Aggregation of arbitrary types
enum	Enumeration with maximum 2^{32} elements

Marshaller:: Exemplos de Tipos da IDL



Marshaller

■ Contexto

- *Requests* e respostas precisam ser transportados através da rede

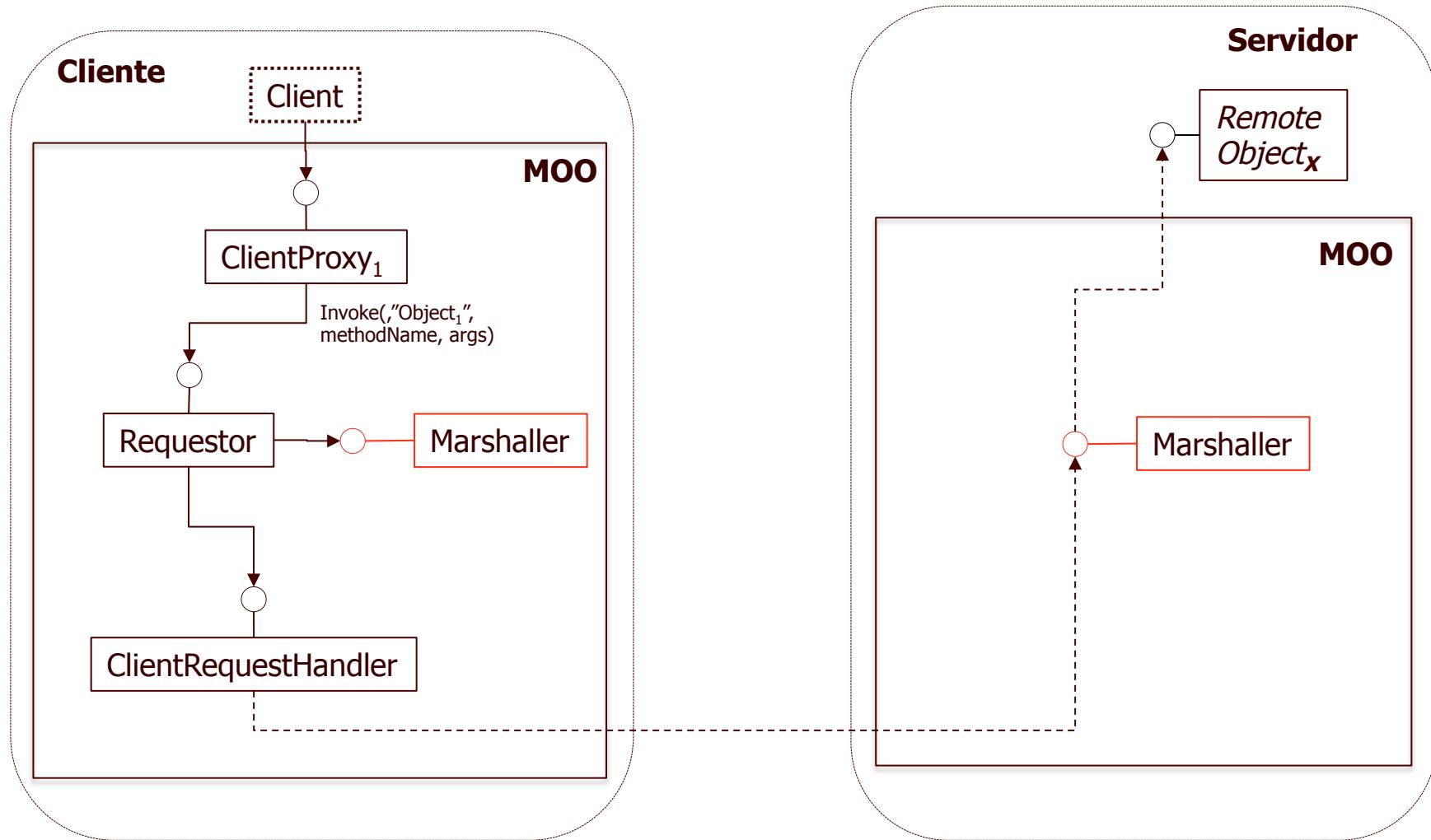
■ Problema

- Dados dos *requests*
 - Object ID, nome da operação, parâmetros e valores de retorno
 - [informações de contexto]
- Apenas *streams* de bytes são transportados pela rede

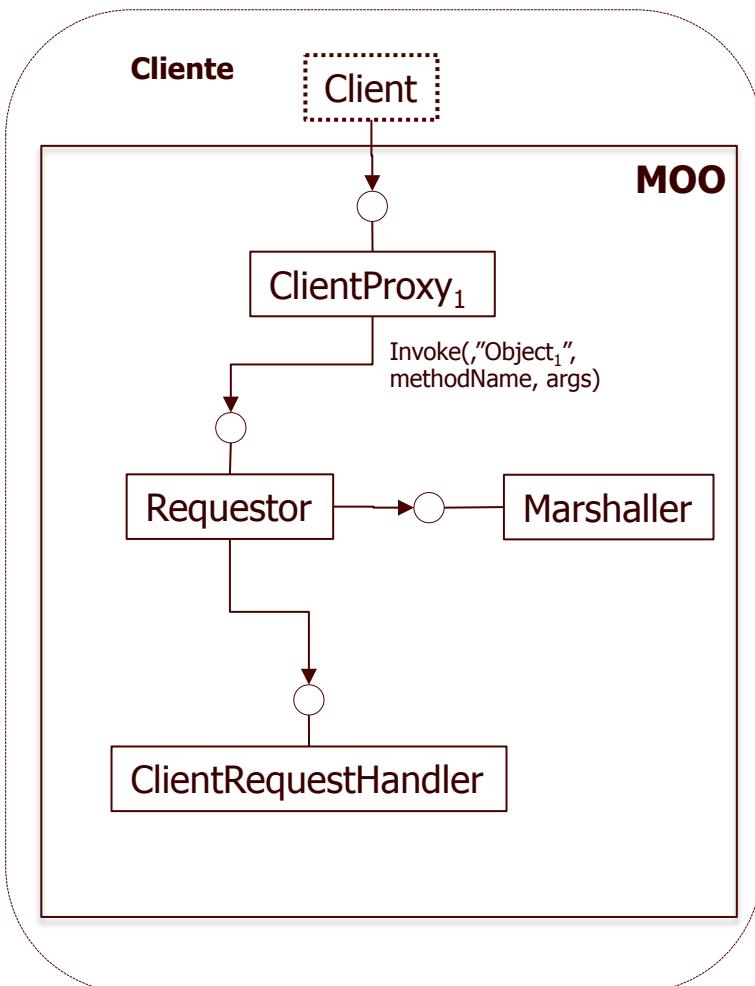
■ Solução

- Uso de **Marshallers** no cliente e servidor que serializem os dados do request/reply
- Tipos primitivos e não primitivos são serializados

Marshaller



Requestor



```
1 package requestor
2
3 import ...
4
5 type Requestor struct{}
6
7 func (Requestor) Invoke(inv aux.Invocation) interface{} {...}
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
```

Upward double-headed arrow between Requestor and Marshaller code blocks.

```
1 package marshaller
2
3 import ...
4
5 type Marshaller struct{}
6
7 func (Marshaller) Marshall(msg miop.Packet) []byte {...}
8
9
10
11
12
13
14
15
16
17
18
19
20
21
```

```
1 package crh
2
3 import ...
4
5
6 type CRH struct {
7     ServerHost string
8     ServerPort int
9 }
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
```

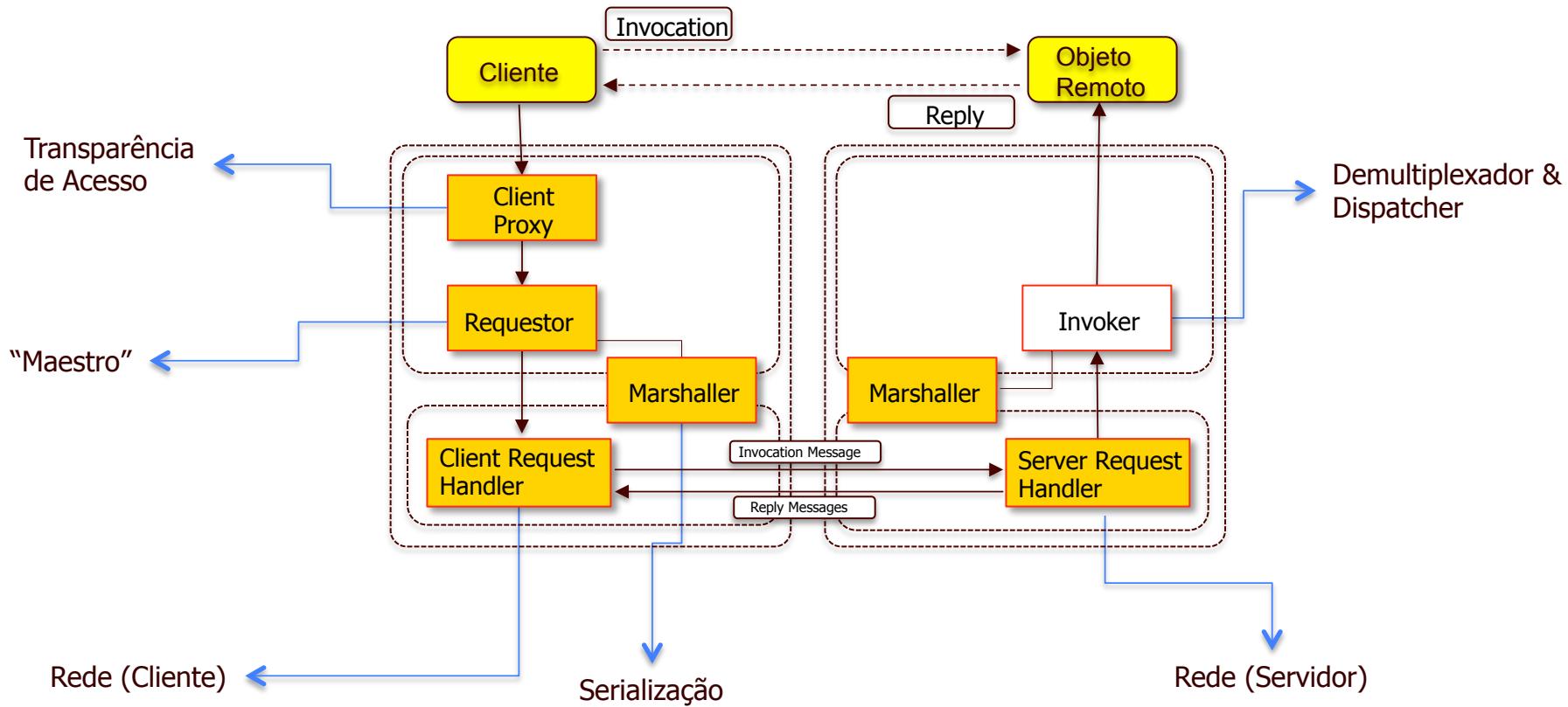
Downward double-headed arrow between Marshaller and CRH code blocks.

Marshaller

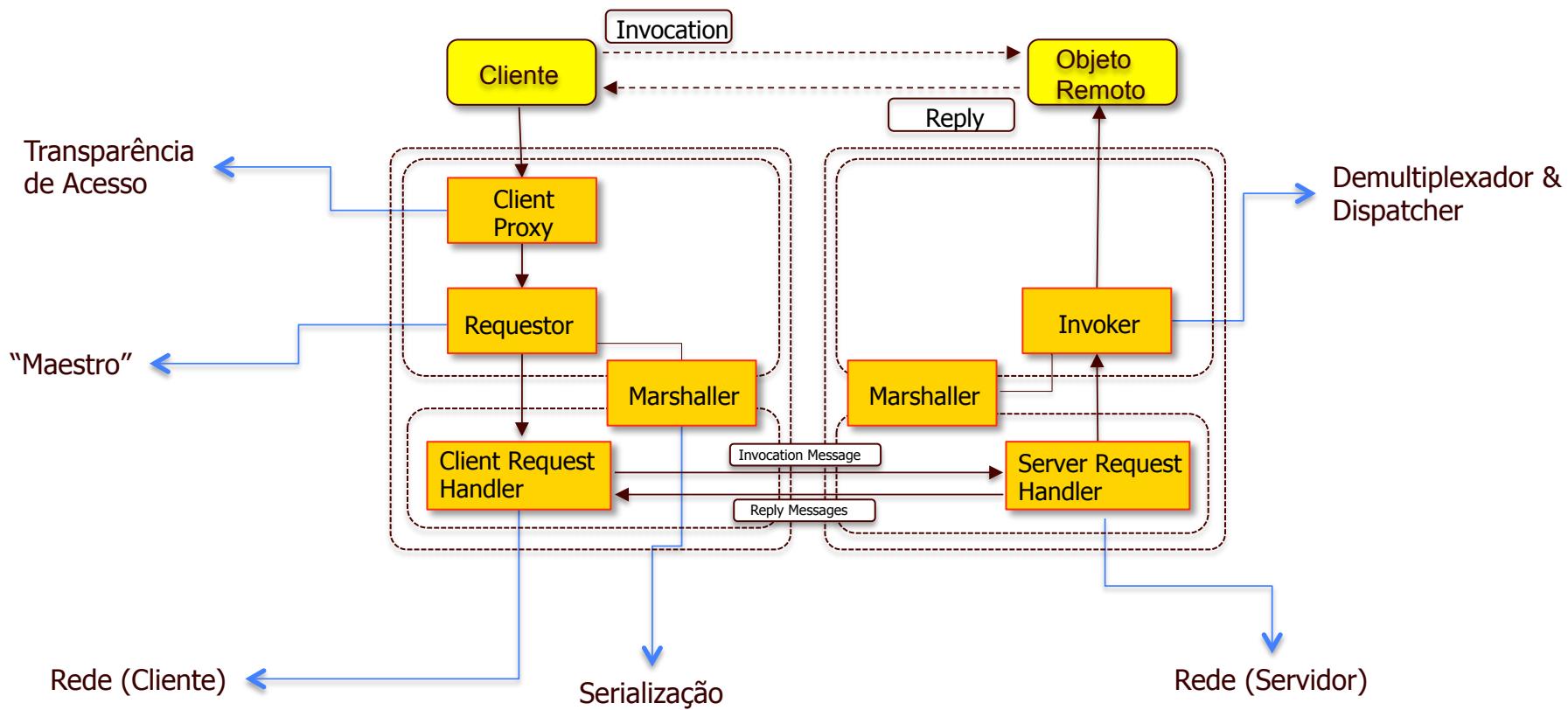
```
1 package marshaller
2
3 import ...
4
5
6 type Marshaller struct{}
7
8
9 func (Marshaller) Marshall(msg miop.Packet) []byte {
10
11     r, err := json.Marshal(msg)
12
13     if err != nil {...}
14
15     return r
16
17 }
18
19
20 func (Marshaller) Unmarshall(msg []byte) miop.Packet {
21
22     r := miop.Packet{}
23
24     err := json.Unmarshal(msg, &r)
25
26     if err != nil {...}
27
28     return r
29
30 }
```

Invoker

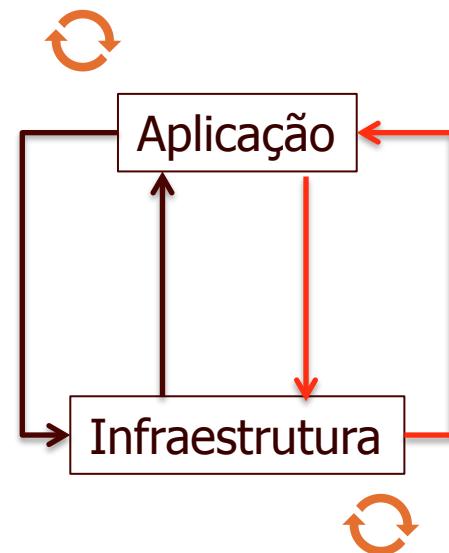
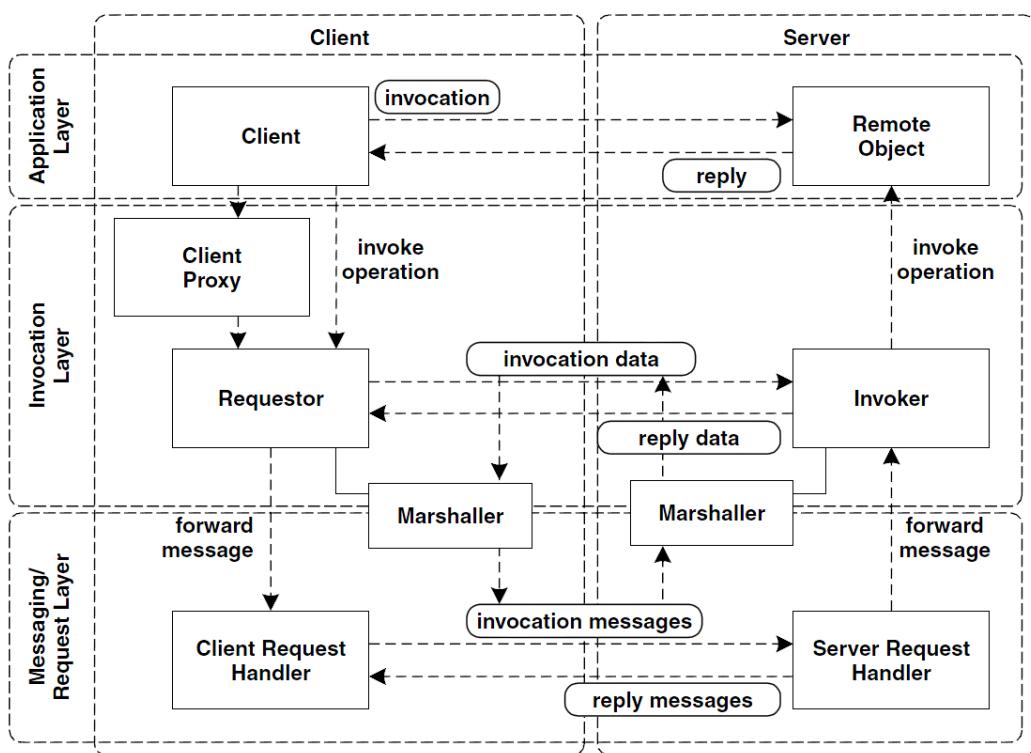
Remoting Patterns



Remoting Patterns



Padrões de Projeto:: Remoting Patterns



- ✓ **Loop de Inversão de Controle:** loop que controla as interações dos componentes no lado do servidor.
- ✓ **Onde colocar o loop de Inversão de Controle?**
 - ✓ **Opção 1:** Na aplicação servidora (sem uso de middleware).
 - ✓ **Opção 2: No Invoker.**
 - ✓ **Opção 3:** No Server Request Handler.

Padrões de Projeto:: Remoting Patterns

Opção 1 (Aplicação -> Infraestrutura)

Servidor (Aplicação)

While (true)

- ```
{
 1. Receber mensagem do cliente
 2. Executar operação
 3. Enviar resposta ao Cliente
}
```

Sistema Operacional

- ```
=>  
1. Receber Mensagem do Cliente  
2. Enviar resposta ao Cliente
```

Objeto Remoto (Aplicação)

- ```
→ 1. Executar a operação
```

## Opção 2 (Infraestrutura->Aplicação)

Invoker

While (true)

- ```
{  
    → 1. Receber mensagem do cliente  
    → 2. Fazer Unmarshall da mensagem  
    → 3. Decidir qual Objeto invocar (demultiplexação)  
    → 4. Decidir qual método invocar (demultiplexação)  
    → 5. Invocar o Objeto remoto  
    → 6. Receber o resultado da invocação  
    → 7. Fazer Marhsall da mensagem  
    → 8. Enviar a resposta ao Cliente  
}
```

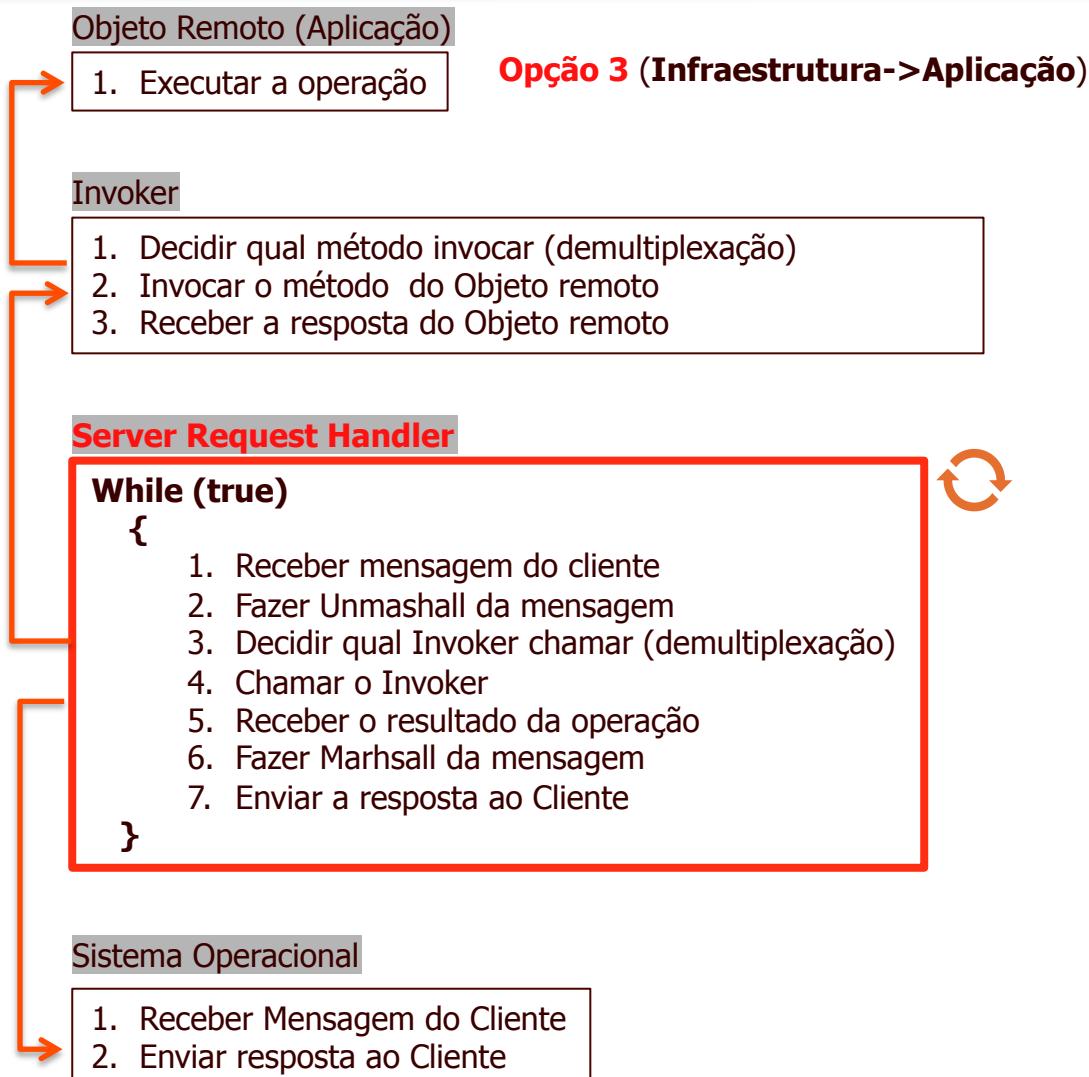
Server Request Handler

- ```
→ 1. Receber Mensagem do Cliente
2. Enviar resposta ao Cliente
```

Sistema Operacional

- ```
=>  
1. Receber Mensagem do Cliente  
2. Enviar resposta ao Cliente
```

Padrões de Projeto:: Remoting Patterns



Padrões de Projeto:: Remoting Patterns

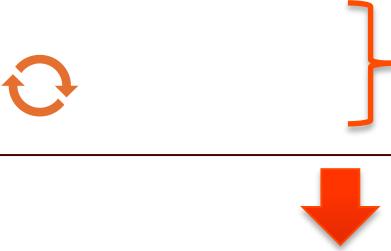
```
1 package main
2
3 import ...
9
10 func main() {
11
12     // create a built-in proxy of naming service
13     namingProxy := proxy.NamingProxy{}
14
15     // create a proxy of calculator service
16     calculator := proxies.NewCalculatorProxy()
17
18     // register service in the naming service
19     namingProxy.Register( p1: "Calculator", calculator)
20
21     // control loop passed to middleware
22     fmt.Println( a: "Calculator Server running!!")
23     calculatorInvoker := invoker.NewCalculatorInvoker()
24     calculatorInvoker.Invoke()
25
26     fmt.Scanln()
27 }
```



Servidor sem loop!!

Padrões de Projeto:: Remoting Patterns

```
1 package invoker
2
3 import ...
10
11 type CalculatorInvoker struct {}
12
13 func NewCalculatorInvoker() CalculatorInvoker {...}
18
19 func (CalculatorInvoker) Invoke(){
20     srhImpl := srh.SRH{ServerHost:"localhost", ServerPort:shared.CALCULATOR_PORT}
21     marshallerImpl := marshaller.Marshaller{}
22     calculatorImpl := impl.Calculadora{}
23     miopPacketReply := miop.Packet{}
24     replParams := make([]interface{},1)
25
26     for {...}
68
69 }
70 }
```



Loop de "Inversão de Controle"

```
package srh
import ...
type SRH struct {...}

var ln net.Listener
var conn net.Conn
var err error

func (srh SRH) Receive() []byte {...}
func (SRH) Send(msgToClient []byte) {...}
```

Invoker

■ Contexto

- O que acontece no “servidor” quando o *Requestor* envia a invocação para um objeto remoto?

■ Problema

- Como fazer a invocação alcançar o objeto remoto depois que ela chega ao servidor?

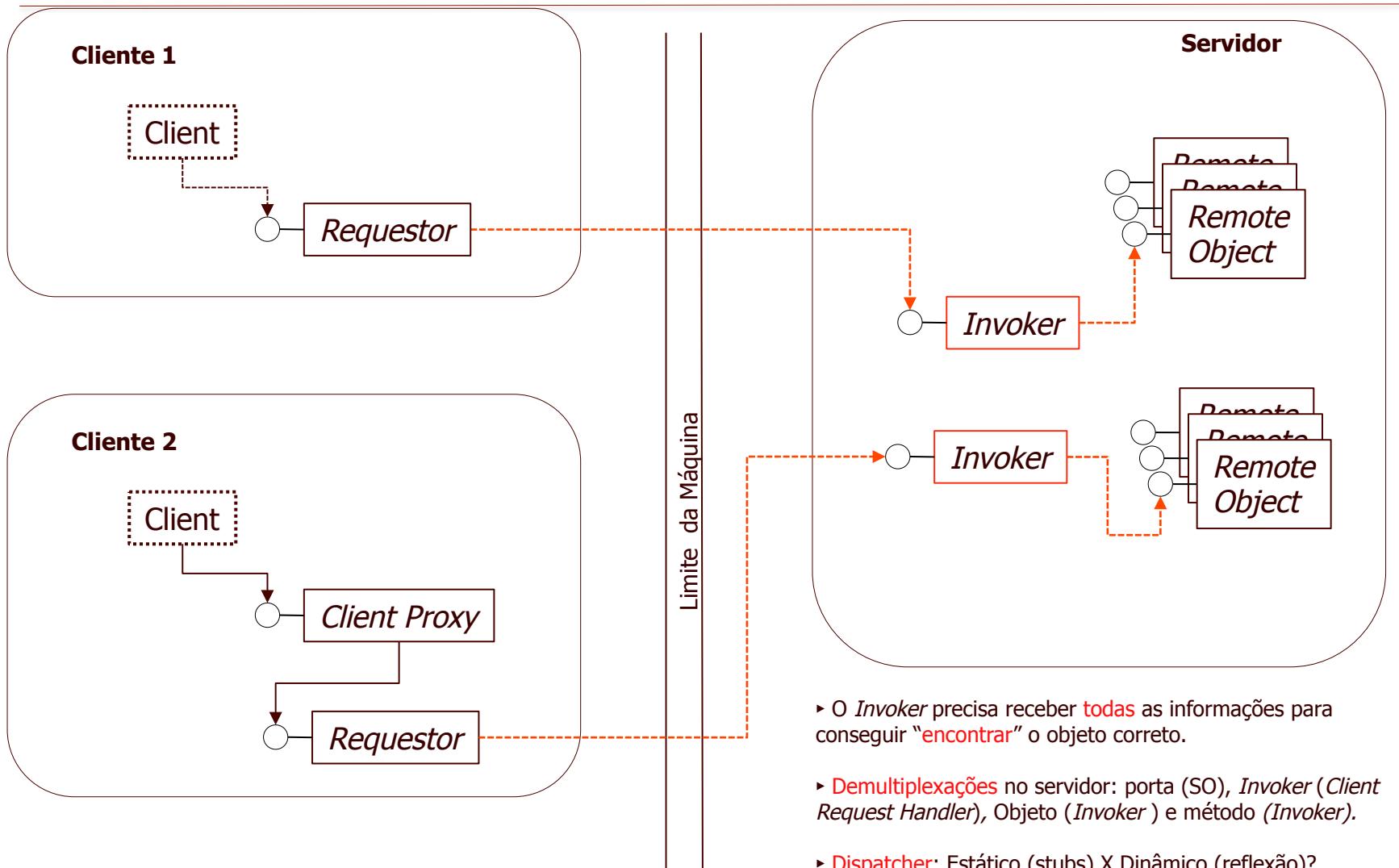
■ Solução 1

- Permitir que todo objeto remoto seja acessível diretamente através de IP+Porta individuais.
 - Neste caso o objeto remoto precisaria ainda tratar as conexões e a serialização

■ Solução 2

- Usar um *Invoker* fornecendo as “informações” necessárias para que ele selecione o objeto remoto apropriado.
- **Importante:** O objeto remoto não deve “se preocupar” com detalhes da comunicação, a serialização ou ativação.

Invoker



Invoker

```
1 package invoker
2
3 import ...
4
5 type CalculatorInvoker struct {}
6
7 func NewCalculatorInvoker() CalculatorInvoker {...}
8
9 func (CalculatorInvoker) Invoke(){
10    srhImpl := srh.SRH{ServerHost:"localhost", ServerPort:shared.CALCULATOR_PORT}
11    marshallerImpl := marshaller.Marshaller{}
12    miopPacketReply := miop.Packet{}
13    replParams := make([]interface{},1)
14
15    calculatorImpl := impl.Calculadora{}
16
17    for {...}
18
19}
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71 }
```

Passo-a-Passo dentro do Invoker:

1. Criar o objeto remoto
2. Invocar o Server Request Handler para receber uma mensagem com a Invocação do Cliente;
3. Fazer o unmarshalling da Mensagem;
4. Usando a Mensagem, fazer a demultiplexação do método a ser invocado no Objeto Remoto;
5. Disparar a Invocação ao Método (**Síncrono**);
6. Fazer o marshalling da resposta da invocação do Método;
7. Invocar o Server Request Handler para enviar a resposta ao Cliente.

Invoker

```
1 package invoker
2
3 import ...
4
5 type CalculatorInvoker struct {}
6
7 func NewCalculatorInvoker() CalculatorInvoker {...}
8
9 func (CalculatorInvoker) Invoke(){
10    srhImpl := srh.SRH{ServerHost:"localhost", ServerPort:shared.CALCULATOR_PORT}
11    marshallerImpl := marshaller.Marshaller{}
12    miopPacketReply := miop.Packet{}
13    replyParams := make([]interface{},1)
14
15    calculatorImpl := impl.Calculadora{}
16
17    for {...}
18
19 }
```

Passo-a-Passo dentro do Invoker:

1. Criar o objeto remoto
2. Invocar o Server Request Handler para receber uma mensagem com a Invocação do Cliente;
3. Fazer o unmarshalling da Mensagem;
4. Usando a Mensagem, fazer a demultiplexação do método a ser invocado no Objeto Remoto;
5. Disparar a Invocação ao Método (**Síncrono**);
6. Fazer o marshalling da resposta da invocação do Método;
7. Invocar o Server Request Handler para enviar a resposta ao Cliente.

Invoker:: Passos 2 & 3

```
for {
    // receive data
    rcvMsgBytes := srhImpl.Receive()

    // unmarshall
    miopPacketRequest := marshallerImpl.Unmarshal(rcvMsgBytes)
    operation := miopPacketRequest.Bd.ReqHeader.Operation

    // demux request
    switch operation {
        case "Add":
            _p1 := int(miopPacketRequest.Bd.ReqBody.Body[0].(float64))
            _p2 := int(miopPacketRequest.Bd.ReqBody.Body[1].(float64))
            replParams[0] = calculatorImpl.Add(_p1,_p2)
        case "Sub" :...
        case "Mul" :...
        case "Div" :...
    }

    // assembly packet
    repHeader := miop.ReplyHeader{Context:"", RequestId: miopPacketRequest.Bd.ReqHeader.RequestId, Status:1}
    repBody := miop.ReplyBody{OperationResult:replParams}
    header := miop.Header{Magic: "MIO", Version: "1.0", ByteOrder:true, MessageType:shared.MIOP_REQUEST}
    body := miop.Body{RepHeader:repHeader, RepBody:repBody}
    miopPacketReply = miop.Packet{Hdr:header, Bd:body}

    // marshall reply
    msgToClientBytes := marshallerImpl.Marshall(miopPacketReply)

    // send reply
    srhImpl.Send(msgToClientBytes)
}
```

Passo-a-Passo dentro do Invoker:

1. Criar o objeto remoto
2. Invocar o Server Request Handler para receber uma mensagem com a Invocação do Cliente;
3. Fazer o unmarshalling da Mensagem;
4. Usando a Mensagem, fazer a demultiplexação do método a ser invocado no Objeto Remoto;
5. Disparar a Invocação ao Método (**Síncrono**);
6. Fazer o marshalling da resposta da invocação do Método;
7. Invocar o Server Request Handler para enviar a resposta ao Cliente.

Invoker:: Passos 2 & 3

```
for {
    // receive data
    rcvMsgBytes := srhImpl.Receive()

    // unmarshall
    miopPacketRequest := marshallerImpl.Unmarshal(rcvMsgBytes)
    operation := miopPacketRequest.Bd.ReqHeader.Operation

    // demux request
    switch operation {
        case "Add":
            _p1 := int(miopPacketRequest.Bd.ReqBody.Body[0].(float64))
            _p2 := int(miopPacketRequest.Bd.ReqBody.Body[1].(float64))
            replParams[0] = calculatorImpl.Add(_p1,_p2)
        case "Sub": ...
        case "Mul": ...
        case "Div": ...
    }

    // assembly packet
    repHeader := miop.ReplyHeader{Context:"", RequestId: miopPacketRequest.Bd.ReqHeader.RequestId, Status:1}
    repBody := miop.ReplyBody{OperationResult:replParams}
    header := miop.Header{Magic: "MIO", Version: "1.0", ByteOrder:true, MessageType:shared.MIOP_REQUEST}
    body := miop.Body{RepHeader:repHeader, RepBody:repBody}
    miopPacketReply = miop.Packet{Hdr:header, Bd:body}

    // marshall reply
    msgToClientBytes := marshallerImpl.Marshall(miopPacketReply)

    // send reply
    srhImpl.Send(msgToClientBytes)
}
```

Passo-a-Passo dentro do Invoker:

1. Criar o objeto remoto
2. Invocar o Server Request Handler para receber uma mensagem com a Invocação do Cliente;
3. Fazer o unmarshalling da Mensagem;
4. Usando a Mensagem, fazer a demultiplexação do método a ser invocado no Objeto Remoto;
5. Disparar a Invocação ao Método (**Síncrono**);
6. Fazer o marshalling da resposta da invocação do Método;
7. Invocar o Server Request Handler para enviar a resposta ao Cliente.

Invoker:: Passos 2 & 3

```
for {  
    // receive data  
    rcvMsgBytes := srhImpl.Receive()  
  
    // unmarshall  
    miopPacketRequest := marshallerImpl.Unmarshal(rcvMsgBytes)  
    operation := miopPacketRequest.Bd.ReqHeader.Operation  
  
    // demux request  
    4 switch operation {  
        case "Add":  
            _p1 := int(miopPacketRequest.Bd.ReqBody.Body[0].(float64))  
            _p2 := int(miopPacketRequest.Bd.ReqBody.Body[1].(float64))  
            replParams[0] = calculatorImpl.Add(_p1,_p2)  
        case "Sub":...  
        case "Mul":...  
        case "Div":...  
    }  
  
    // assembly packet  
    repHeader := miop.ReplyHeader{Context:"", RequestId: miopPacketRequest.Bd.ReqHeader.RequestId, Status:1}  
    repBody := miop.ReplyBody{OperationResult:replParams}  
    header := miop.Header{Magic: "MIO", Version: "1.0", ByteOrder:true, MessageType:shared.MIOP_REQUEST}  
    body := miop.Body{RepHeader:repHeader, RepBody:repBody}  
    miopPacketReply = miop.Packet{Hdr:header, Bd:body}  
  
    // marshall reply  
    msgToClientBytes := marshallerImpl.Marshall(miopPacketReply)  
  
    // send reply  
    srhImpl.Send(msgToClientBytes)  
}
```

Passo-a-Passo dentro do Invoker:

1. Criar o objeto remoto
2. Invocar o Server Request Handler para receber uma mensagem com a Invocação do Cliente;
3. Fazer o unmarshalling da Mensagem;
4. Usando a Mensagem, fazer a demultiplexação do método a ser invocado no Objeto Remoto;
5. Disparar a Invocação ao Método (**Síncrono**);
6. Fazer o marshalling da resposta da invocação do Método;
7. Invocar o Server Request Handler para enviar a resposta ao Cliente.

Invoker:: Passos 2 & 3

```
for {
    // receive data
    rcvMsgBytes := srhImpl.Receive()

    // unmarshall
    miopPacketRequest := marshallerImpl.Unmarshal(rcvMsgBytes)
    operation := miopPacketRequest.Bd.ReqHeader.Operation

    // demux request
    switch operation {
        case "Add":
            _p1 := int(miopPacketRequest.Bd.ReqBody.Body[0].(float64))
            _p2 := int(miopPacketRequest.Bd.ReqBody.Body[1].(float64))
            replParams[0] = calculatorImpl.Add(_p1,_p2)
            5
        case "Sub": ...
        case "Mul": ...
        case "Div": ...
    }

    // assembly packet
    repHeader := miop.ReplyHeader{Context:"", RequestId: miopPacketRequest.Bd.ReqHeader.RequestId, Status:1}
    repBody := miop.ReplyBody{OperationResult:replParams}
    header := miop.Header{Magic: "MIO", Version: "1.0", ByteOrder:true, MessageType:shared.MIOP_REQUEST}
    body := miop.Body{RepHeader:repHeader, RepBody:repBody}
    miopPacketReply = miop.Packet{Hdr:header, Bd:body}

    // marshall reply
    msgToClientBytes := marshallerImpl.Marshall(miopPacketReply)

    // send reply
    srhImpl.Send(msgToClientBytes)
}
```

Passo-a-Passo dentro do Invoker:

1. Criar o objeto remoto
2. Invocar o Server Request Handler para receber uma mensagem com a Invocação do Cliente;
3. Fazer o unmarshalling da Mensagem;
4. Usando a Mensagem, fazer a demultiplexação do método a ser invocado no Objeto Remoto;
5. **Disparar a Invocação ao Método (Síncrono);**
6. Fazer o marshalling da resposta da invocação do Método;
7. Invocar o Server Request Handler para enviar a resposta ao Cliente.

Invoker:: Passos 2 & 3

```
for {
    // receive data
    rcvMsgBytes := srhImpl.Receive()

    // unmarshall
    miopPacketRequest := marshallerImpl.Unmarshal(rcvMsgBytes)
    operation := miopPacketRequest.Bd.ReqHeader.Operation

    // demux request
    switch operation {
        case "Add":
            _p1 := int(miopPacketRequest.Bd.ReqBody.Body[0].(float64))
            _p2 := int(miopPacketRequest.Bd.ReqBody.Body[1].(float64))
            replParams[0] = calculatorImpl.Add(_p1,_p2)
        case "Sub" :...
        case "Mul" :...
        case "Div" :...
    }

    // assembly packet
    repHeader := miop.ReplyHeader{Context:"", RequestId: miopPacketRequest.Bd.ReqHeader.RequestId, Status:1}
    repBody := miop.ReplyBody{OperationResult:replParams}
    header := miop.Header{Magic: "MIO", Version: "1.0", ByteOrder:true, MessageType:shared.MIOP_REQUEST}
    body := miop.Body{RepHeader:repHeader, RepBody:repBody}
    miopPacketReply = miop.Packet{Hdr:header, Bd:body}

    // marshall reply
    msgToClientBytes := marshallerImpl.Marshall(miopPacketReply)

    // send reply
    srhImpl.Send(msgToClientBytes)
}
```

Passo-a-Passo dentro do Invoker:

1. Criar o objeto remoto
2. Invocar o Server Request Handler para receber uma mensagem com a Invocação do Cliente;
3. Fazer o unmarshalling da Mensagem;
4. Usando a Mensagem, fazer a demultiplexação do método a ser invocado no Objeto Remoto;
5. Disparar a Invocação ao Método (**Síncrono**);
6. Fazer o marshalling da resposta da invocação do Método;
7. Invocar o Server Request Handler para enviar a resposta ao Cliente.

Invoker:: Passos 2 & 3

```
for {
    // receive data
    rcvMsgBytes := srhImpl.Receive()

    // unmarshall
    miopPacketRequest := marshallerImpl.Unmarshal(rcvMsgBytes)
    operation := miopPacketRequest.Bd.ReqHeader.Operation

    // demux request
    switch operation {
        case "Add":
            _p1 := int(miopPacketRequest.Bd.ReqBody.Body[0].(float64))
            _p2 := int(miopPacketRequest.Bd.ReqBody.Body[1].(float64))
            replParams[0] = calculatorImpl.Add(_p1,_p2)
        case "Sub" :...
        case "Mul" :...
        case "Div" :...
    }

    // assembly packet
    repHeader := miop.ReplyHeader{Context:"", RequestId: miopPacketRequest.Bd.ReqHeader.RequestId, Status:1}
    repBody := miop.ReplyBody{OperationResult:replParams}
    header := miop.Header{Magic: "MIO", Version: "1.0", ByteOrder:true, MessageType:shared.MIOP_REQUEST}
    body := miop.Body{RepHeader:repHeader, RepBody:repBody}
    miopPacketReply = miop.Packet{Hdr:header, Bd:body}

    // marshall reply
    msgToClientBytes := marshallerImpl.Marshall(miopPacketReply)

    // send reply
    srhImpl.Send(msgToClientBytes)
}
```

Passo-a-Passo dentro do Invoker:

1. Criar o objeto remoto
2. Invocar o Server Request Handler para receber uma mensagem com a Invocação do Cliente;
3. Fazer o unmarshalling da Mensagem;
4. Usando a Mensagem, fazer a demultiplexação do método a ser invocado no Objeto Remoto;
5. Disparar a Invocação ao Método (**Síncrono**);
6. Fazer o marshalling da resposta da invocação do Método;
7. Invocar o Server Request Handler para enviar a resposta ao Cliente.

Aplicação:: Servidor

MyMiddleware

```
1 package main
2
3 import ...
4
5
6 func main() {
7
8     // create a built-in proxy of naming service
9     namingProxy := proxy.NamingProxy{}
10
11    // create a proxy of calculator service
12    calculator := proxies.NewCalculatorProxy()
13
14    // register service in the naming service
15    namingProxy.Register( p1: "Calculator", calculator)
16
17    // control loop passed to middleware
18    fmt.Println( a: "Calculator Server running!!")
19    calculatorInvoker := invoker.NewCalculatorInvoker()
20    calculatorInvoker.Invoke()
21
22    fmt.Scanln()
23
24
25
26
27 }
```

Go RPC

```
package main
import ...
func servidorRPCTCP(){
    // cria instância da calculadora
    calculadora := new(impl.CalculadoraRPC)

    // cria um novo servidor rpc e registra a calculadora
    server := rpc.NewServer()
    server.RegisterName( name: "Calculadora", calculadora)

    // cria um listen rpc-sender
    l, err := net.Listen( network: "tcp", ":"+strconv.Itoa(shared.CALCULATOR_PORT))
    shared.ChecaErro(err, msg: "Servidor não está pronto")

    // aguarda por chamadas
    fmt.Println( a: "Servidor pronto (RPC TCP) ...")
    server.Accept(l)
}

func main() {...}
```

Aplicação:: Cliente

```
▶ func main() {  
    // create a built-in proxy of naming service  
    namingService := proxy.NamingProxy{}  
  
    // look for a service in naming service  
    calculator := namingService.Lookup( p1: "Calculator").(proxies.CalculatorProxy)  
  
    // invoke remote operation  
    fmt.Println(calculator.Add( p1: 1, p2: 2))  
}
```

MyMiddleware

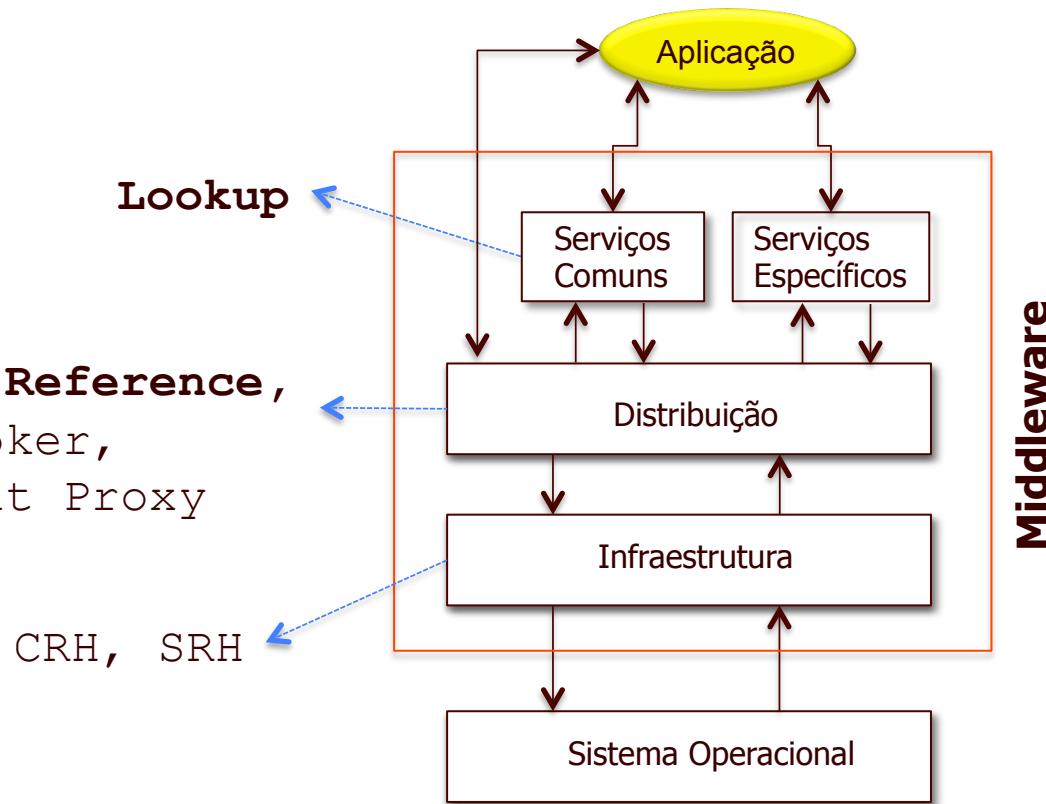
```
▶ func clienteRPCTCP() {  
    var reply int  
  
    // conectar ao servidor  
    client, err := rpc.Dial( network: "tcp", ":"+strconv.Itoa(shared.CALCULATOR_PORT))  
    shared.ChecaErro(err, msg: "Servidor não está pronto")  
  
    defer client.Close()  
  
    // invoca request  
    args := shared.Args{A: 1, B: 2}  
    client.Call( serviceMethod: "Calculadora.Add", args, &reply)  
}
```

Go RPC

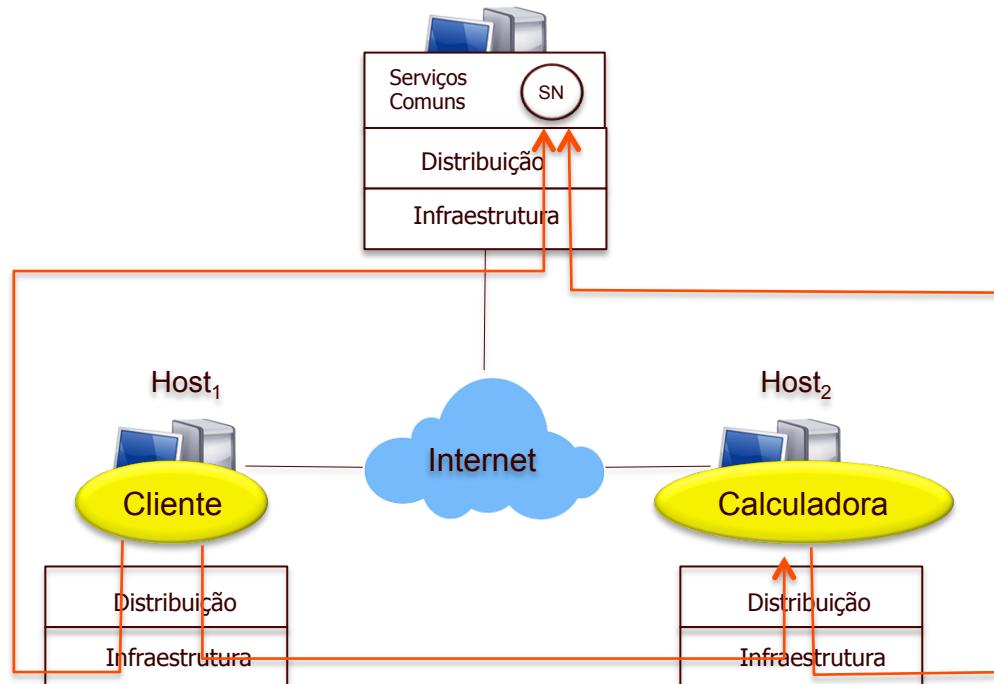
Padrões de Identificação

Basic Remoting Patterns

objectId,
Absolute Object Reference,
Marshaller, Invoker,
Requestor, Client Proxy



Padrões de Identificação:: Contexto



 Serviço de Nomes

Padrões de Identificação:: *Object ID*

■ Contexto

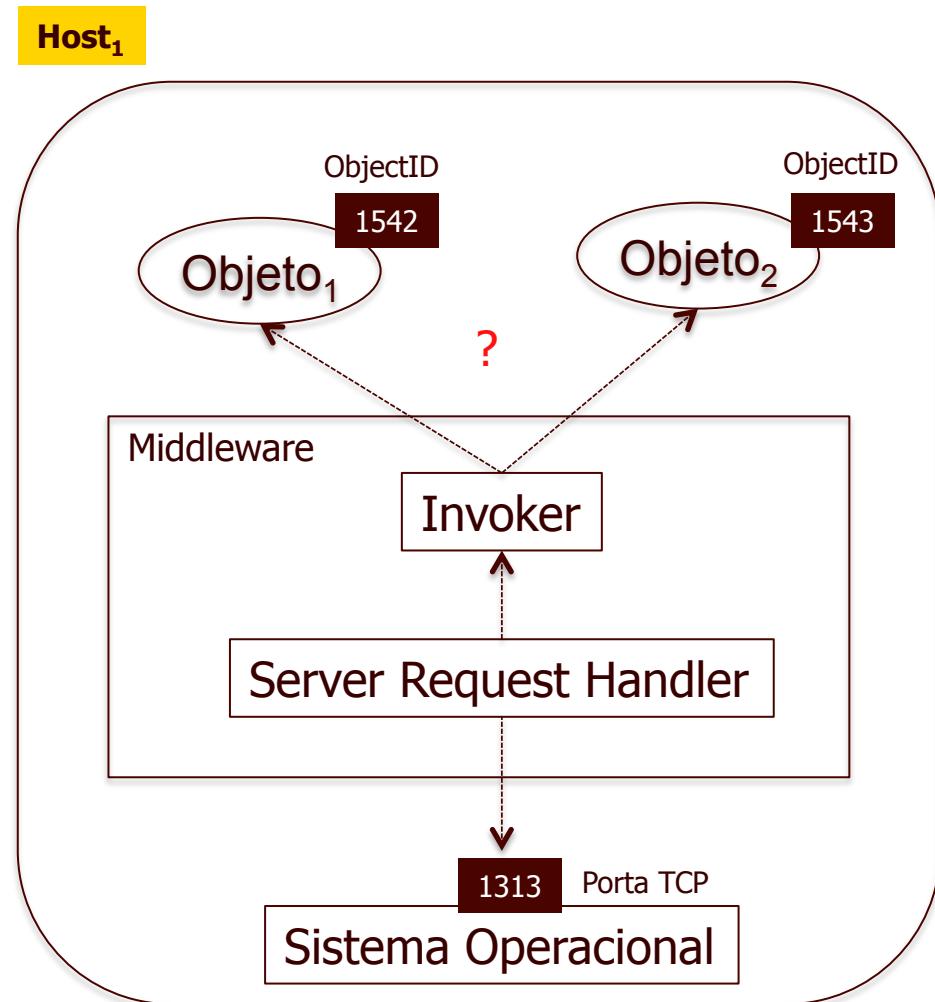
- O *Invoker* tem que selecionar um objeto remoto (já registrado) para enviar a invocação

■ Problema

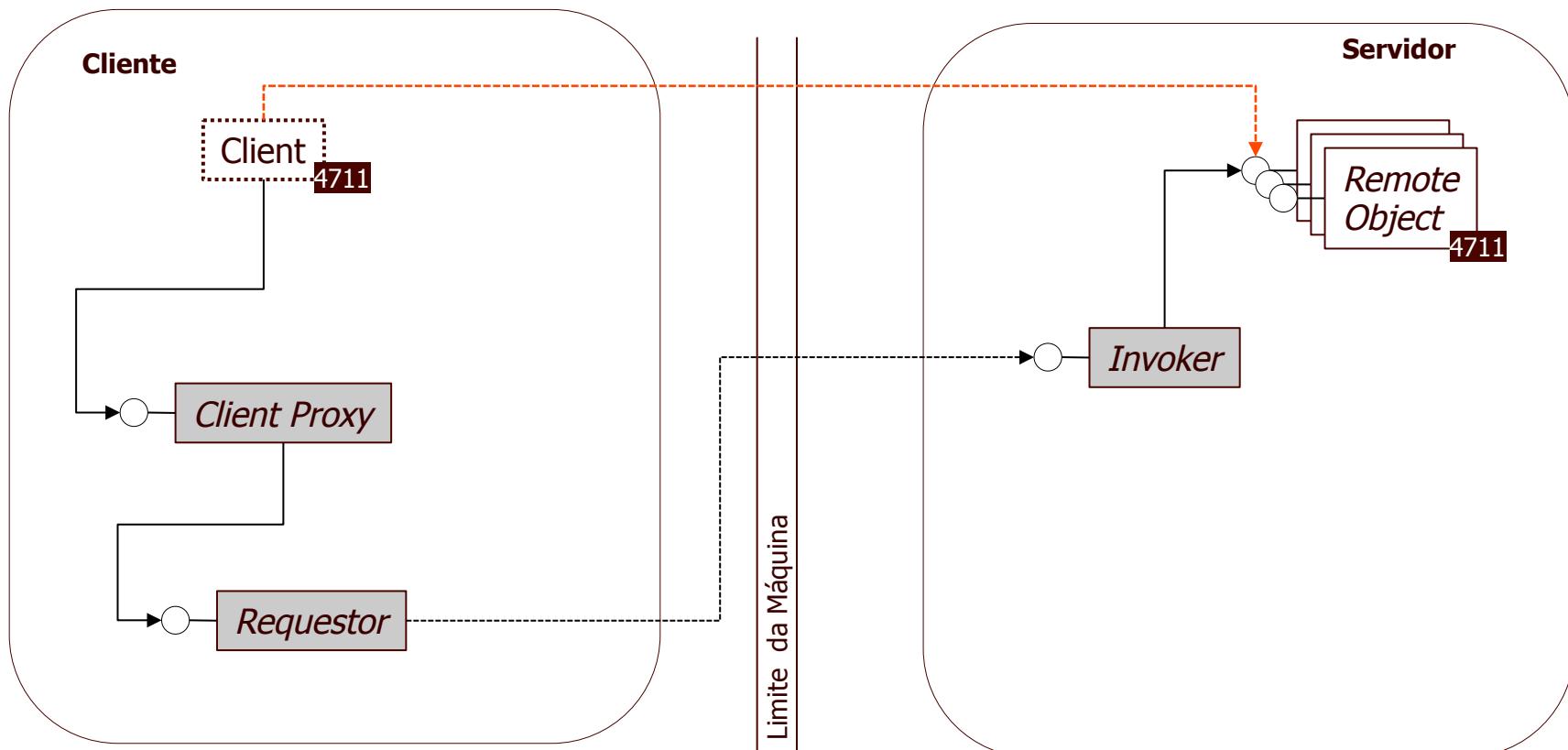
- Como o *Invoker* seleciona o objeto remoto que receberá a invocação?
- Considerando que ...
 - *Invoker* recebe invocações do *Server Request Handler* que precisam ser encaminhadas para vários objetos remotos.

■ Solução

- Associar a cada objeto remoto um *Object Id* único (geralmente um número) no contexto do *Invoker*.



Padrões de Identificação:: *Object ID*



- ▶ O *Object Id* pode ser **global** (único não apenas no servidor).
- ▶ *Object Id* identifica o objeto remoto.
- ▶ **Servant** “realiza” o objeto remoto em tempo de execução (um objeto remoto pode ter vários *servants*)

- ▶ O *Object Id* é **armazenado** no *Client Proxy*.
- ▶ Todas as invocações de um mesmo *Client Proxy* chegam no **mesmo** objeto remoto.

Padrões de Identificação:: *Absolute Object Reference*

■ Contexto

- *Invoker* usa *Object Ids* para enviar a invocação ao objeto remoto correto.

■ Problema

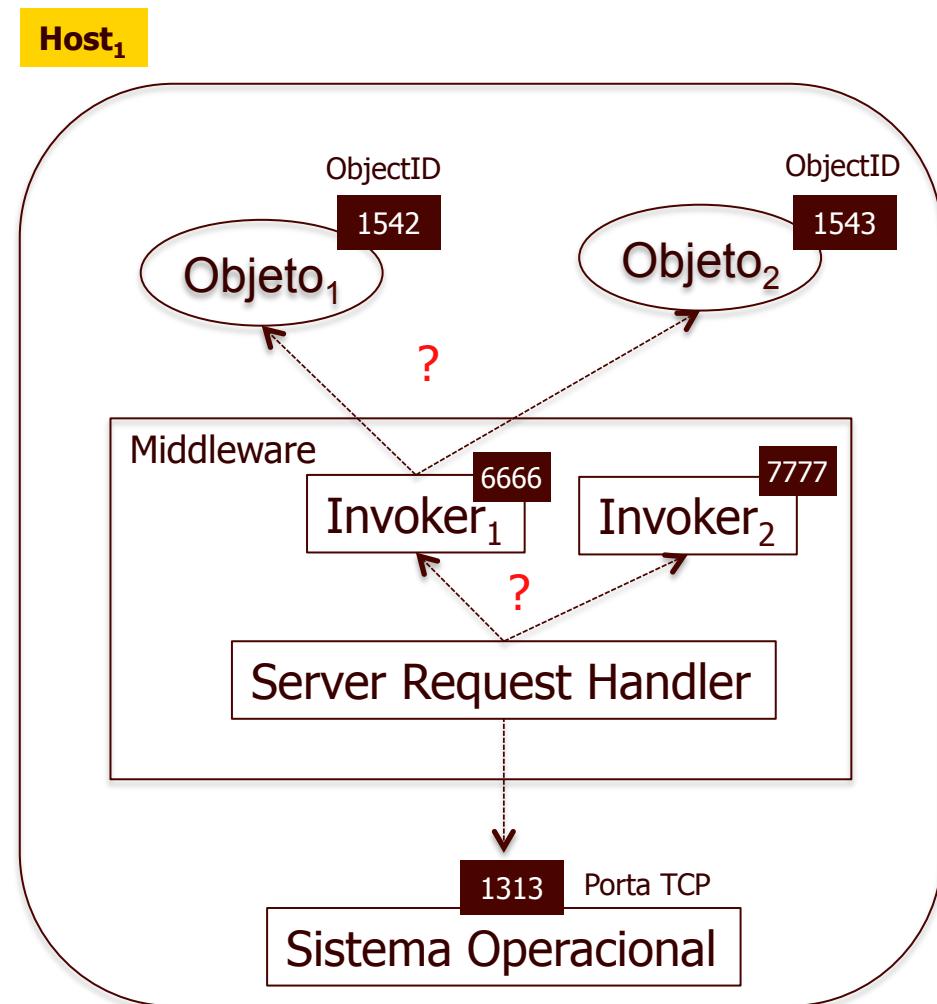
- Como o *Object Id* é enviado ao *Server Request Handler* (e *Invoker*) correto(s)?

■ Solução 1

- O cliente pode manter um mapa que relaciona os *Object Ids* aos *endpoints* de rede

■ Solução 2

- Criar um *Absolute Object Reference* (AOR) que identifica unicamente o *Invoker* e o objeto remoto.
- **Absolute Object Reference** = IP+
PORTA + [id do *Invoker*] + [protocolo de
comunicação] + *Object Id*



Padrões de Identificação:: *Absolute Object Reference*

- AOR deve conter informação suficiente para permitir o cliente contactar o *Invoker* que mantém o registro do objeto remoto
- Quando há múltiplos *Invokers*, AOR deve incluir o Id do *Invoker*
- Quando o *Client Request Handler* usa mais de um protocolo, a AOR também inclui a identificação do protocolo.
- Tipos de AOR
 - **Opacas**: Cliente obtém uma AOR através do serviço de nomes.
 - **Não-opacas**: Cliente constrói a AOR (e.g., acesso inicial a objetos remotos “importantes”), i.e., “**adeus**” transparência de localização
 - **Transiente**: torna-se inválida depois que o objeto remoto é desativado ou a aplicação é reiniciada.
 - **Persistente**: permanece válida depois que a aplicação é reiniciada.

Padrões de Identificação:: *Lookup*

■ Contexto

- Clientes querem usar serviços fornecidos por objetos remotos.

■ Problema

- **Como um cliente pode obter uma AOR (inicial) válida?**

■ Solução 1

- O servidor pode enviar um e-mail com a AOR

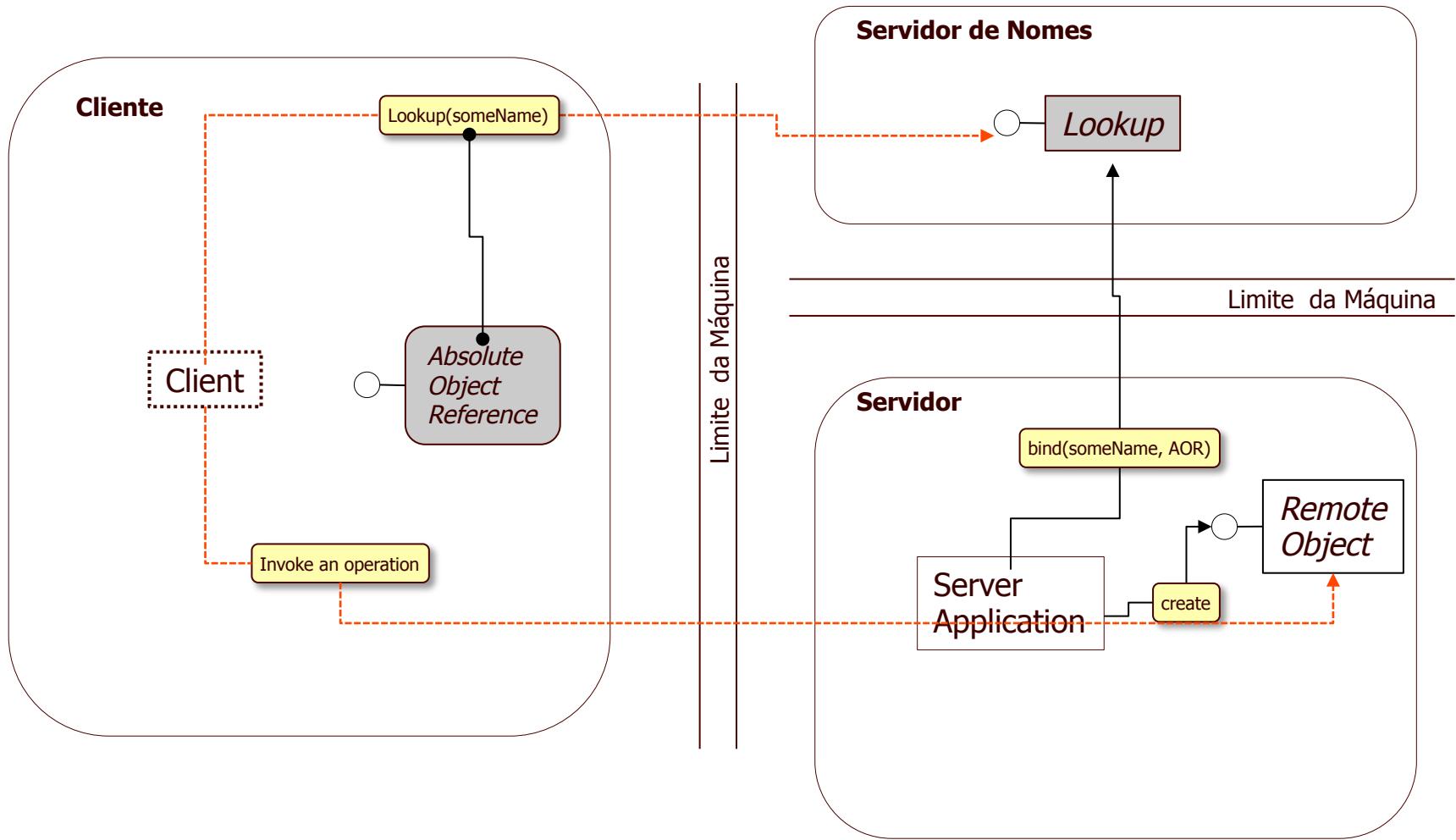
■ Solução 2

- Invocar uma operação a “outro” objeto remoto que retorna a AOR desejada. Como obter a AOR do “outro” objeto remoto?

■ Solução 3

- Implementar um serviço de nomes (*Lookup*) no middleware.
- Servidores registram AORs (por nome) e clientes obtém estas AORs.

Padrões de Identificação:: *Lookup*



Padrões de Identificação:: *Lookup*

■ Interface

- Registro (bind)
- Procura (lookup)

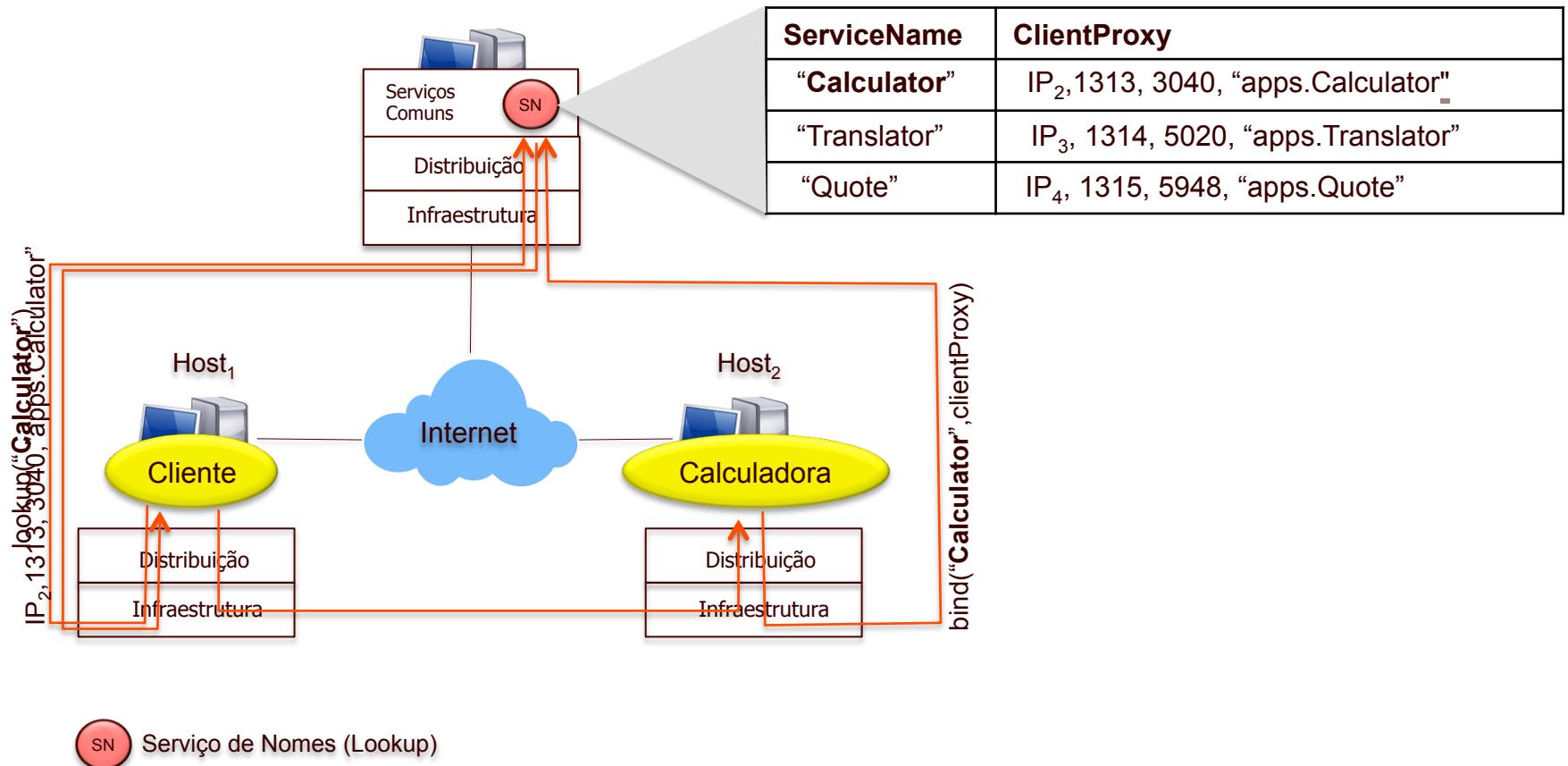
■ AOR do Lookup é persistente

■ Como “achar” um serviço/objeto?

- nome (serviço de nomes)
 - e.g., /impressoras/laser/hp02
- propriedades (trader)
 - e.g., printertype=“laser” and location=“Bloco D”

■ AOR pode ter um leasing (“tempo de vida”)

Padrões de Identificação:: *Lookup*:: Implementação



Padrões de Identificação:: *Lookup*:: Implementação

```
package naming

import ...

type NamingService struct {...}

func (naming *NamingService) Register(name string, proxy clientproxy.ClientProxy) (bool) {...}

func (naming NamingService) Lookup(name string) clientproxy.ClientProxy {...}

func (naming NamingService) List(name string) map[string]clientproxy.ClientProxy {...}
```

- ✓ **Register**: registra um serviço (pelo nome) no serviço de nomes.
- ✓ **Lookup**: procura (pelo nome) por um serviço registrado no serviço de nomes.
- ✓ **List**: lista os serviços registrados no serviço de nomes.
- ✓ **Operações adicionais comuns: unbind** (remove um serviço) e **rebind** (registra um serviço sobrepondo um nome já existente)

Padrões de Identificação:: *Lookup*:: Implementação

```
type NamingService struct {
    Repository map[string]clientproxy.ClientProxy
}
```

ServiceName	ClientProxy
“Calculator”	IP ₂ , 1313, 3040, “apps.Calculator”
“Translator”	IP ₃ , 1314, 5020, “apps.Translator”
“Quote”	Host ₄ , 1315, 5948, “apps.Quote”

```
package clientproxy

type ClientProxy struct {
    Host     string
    Port     int
    Id       int
    TypeName string
}
```

Sobre os Serviços de Nomes (Lookups)

- Há vários serviços de nomes em sistemas distribuídos
 - RM-ODP (ISO/IEC 10746-1)
 - UDDI (OASIS)
 - CORBA (OMG)
 - Registry (RMI)
 - OSGi Framework service registry (OSGi Alliance)
- A estruturação das informações no serviço de nomes independe do padrão de projeto Lookup
- Serviços de nomes “mais complexos” são conhecidos como “traders”
 - “mais complexos” ≈ serviços são procurados por propriedades (e não por nomes)
- Há ainda API, como o JNDI, que é “independente” do serviço de nomes específicos.

Fim dos Slides