

# Como Implementar um Middleware?

## Parte III

Nelson Rosa – nsr@cin.ufpe.br

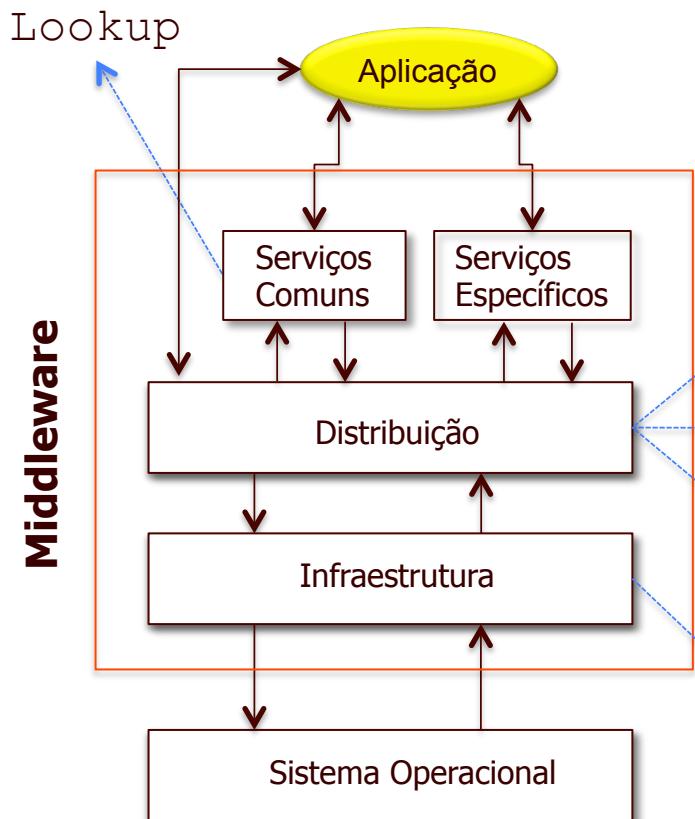
# Objetivos

---

## ■ Apresentar e discutir os Remoting Patterns

- Padrões de Gerenciamento de Ciclo de Vida
- Padrões de Extensão
- Padrões de Infraestrutura Estendida
- Padrões de Invocação Assíncrona

# Remoting Patterns



## Padrões de Gerenciamento de Ciclo de Vida

- Object Id, Absolute Object Reference
- Requestor, Invoker, Marshaller, Client Proxy
- Client Request Handler, Server Request Handler

# Lifecycle Management Patterns

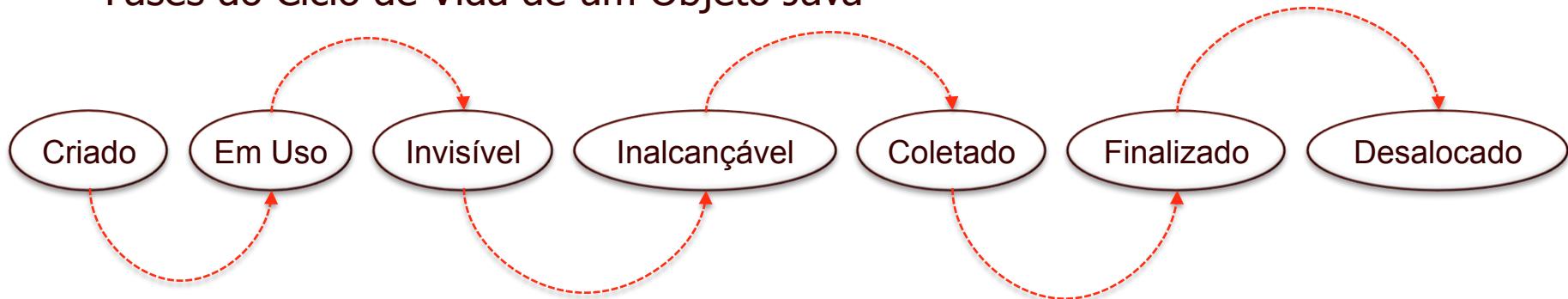
---

- **Fatos Básicos**
  - 07 Padrões **comportamentais** para gerenciamento do ciclo de vida dos objetos remotos
  - **Não há correspondência “um padrão de projeto implementado por uma classe”**
  - Usados no “lado do servidor”
- **Gerenciamento do Ciclo de Vida ≈ Ativação/Desativação do Objeto Remoto**
- **Ativar o Objeto Remoto**
  - Todas as ações necessárias para tornar um objeto remoto “pronto” para receber as invocações
    1. Criar uma instância do objeto remoto (servant)
    2. Inicializar e registrar no Invoker e no Lifecycle Manager
    3. Registrar no serviço de nomes (**Lookup**)
- **Desativar o Objeto Remoto**
  - Reverter a ativação

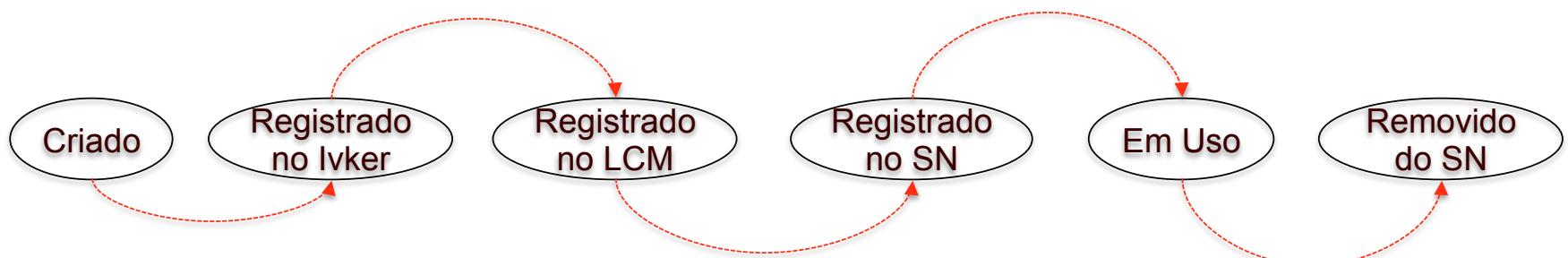
# Lifecycle Management Patterns

---

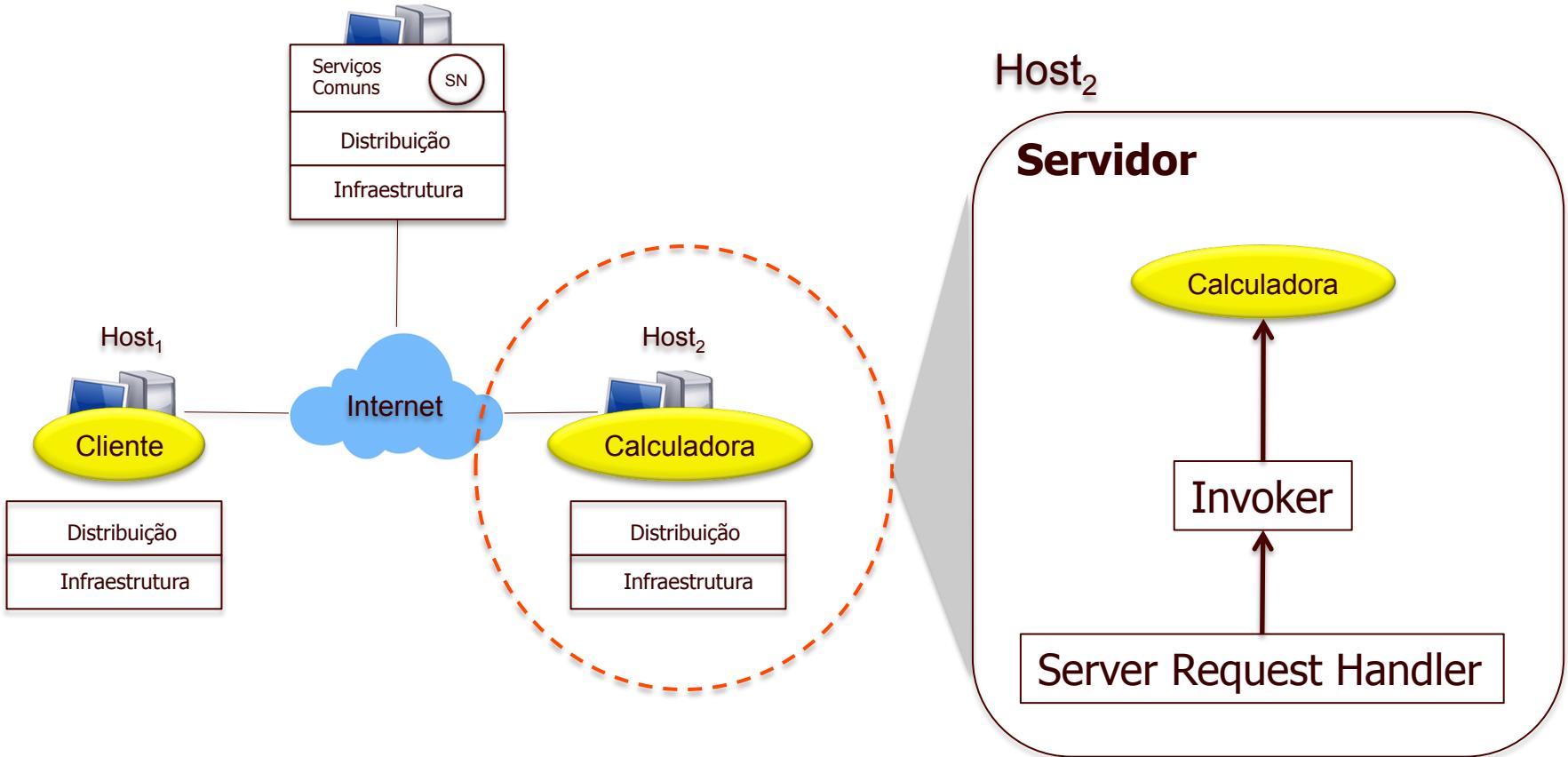
- ✓ Fases do Ciclo de Vida de um Objeto Java



- ✓ [Possível] Ciclo de Vida um Objeto Remoto

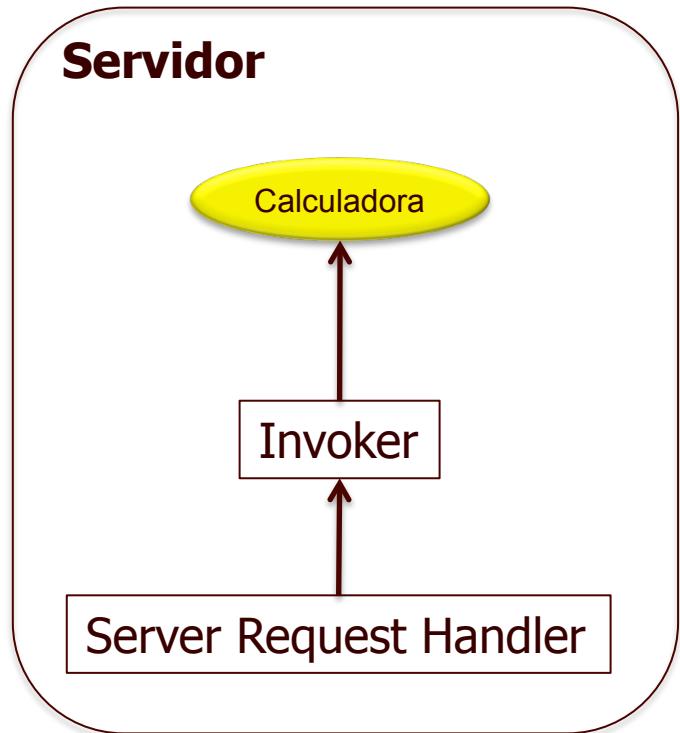


# Lifecycle Management Patterns



# Lifecycle Management Patterns

Host<sub>2</sub>

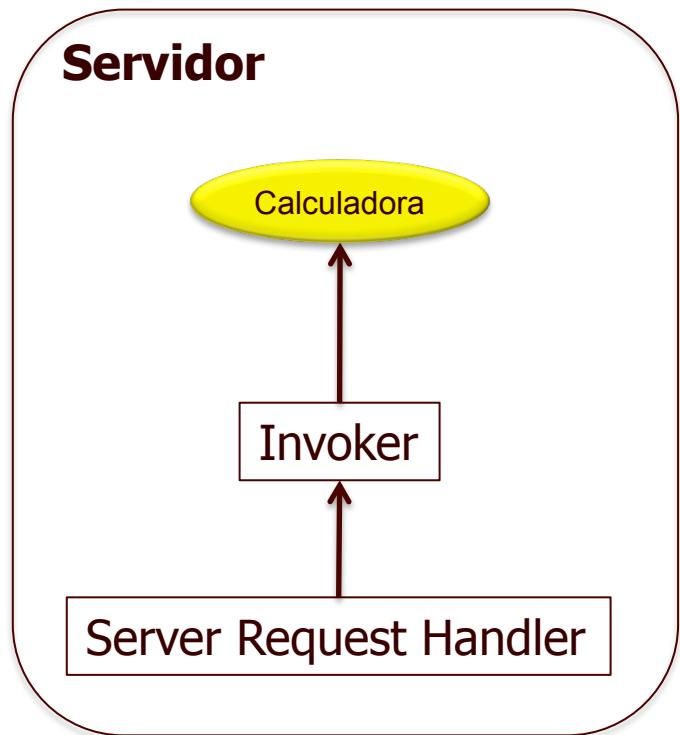


## Sobre a instância:

- Uma instância da Calculadora atende a todos os Clientes (**Static Instance**).
- Uma instância da Calculadora é criada para cada Cliente (**Client-Dependent Instance**).
- Uma instância da Calculadora é criada para cada invocação do Cliente (**Per-Request Instance**)

# Lifecycle Management Patterns

Host<sub>2</sub>



**Quando** a instância é criada?

- No momento em que é invocada (**Lazy Acquisition**).
- Várias instâncias são criadas a priori (**Pooling**)

O que acontece com a instância **após ser criada**?

- É destruída após um tempo pré-estabelecido (**Leasing**)
- É passivada após um tempo pré-estabelecido (**Passivation**)
- **IMPORTANTE:** Estes comportamentos podem ser combinados de diversas formas.

# Lifecycle Management Patterns:: *Static Instance*

---

## ■ Contexto

- Quando um objeto remoto fornece um serviço que é independente de um cliente específico

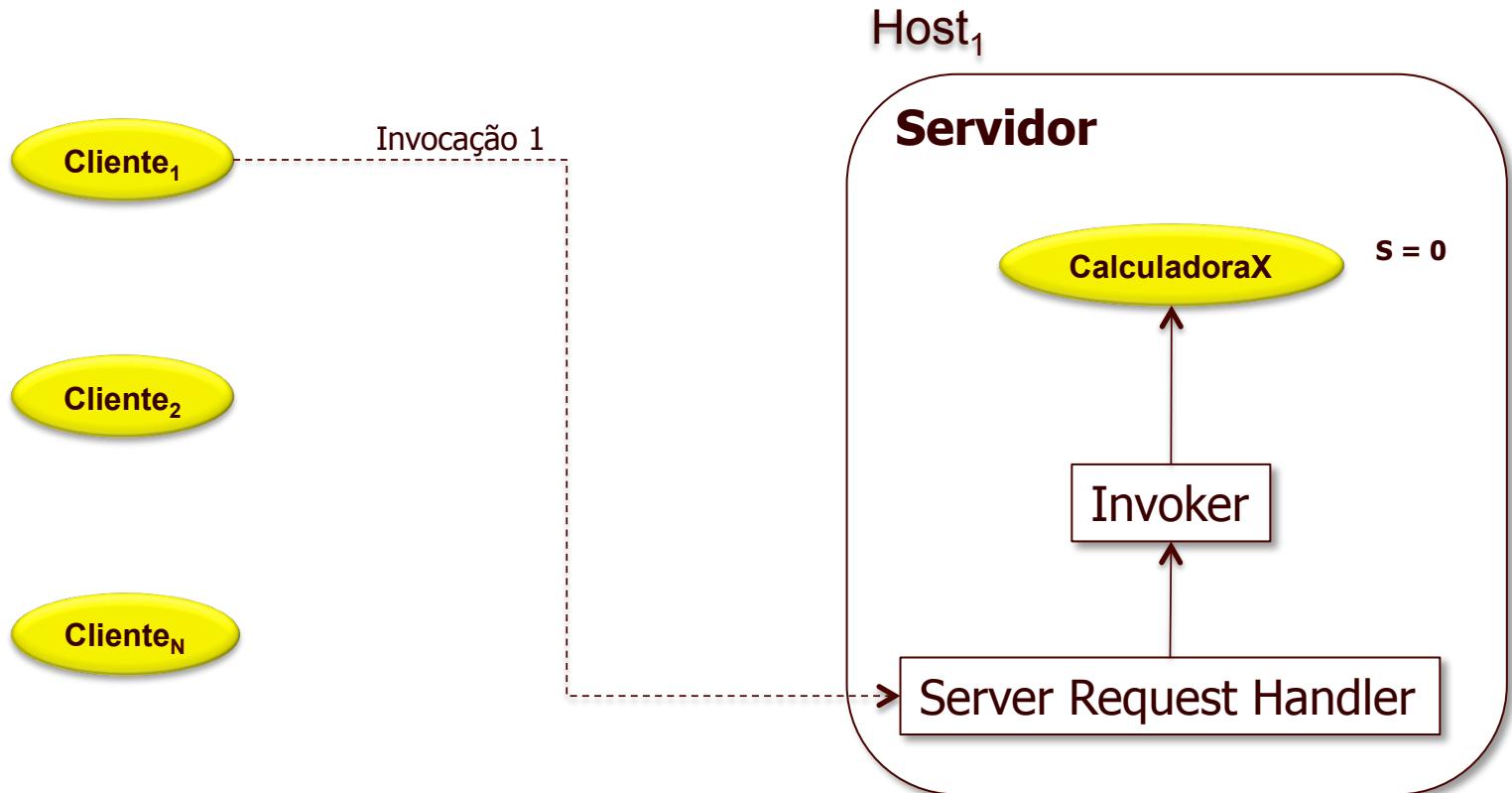
## ■ Problema

- **Como manter o estado do objeto remoto entre invocações sucessivas de um ou mais clientes?**

## ■ Solução

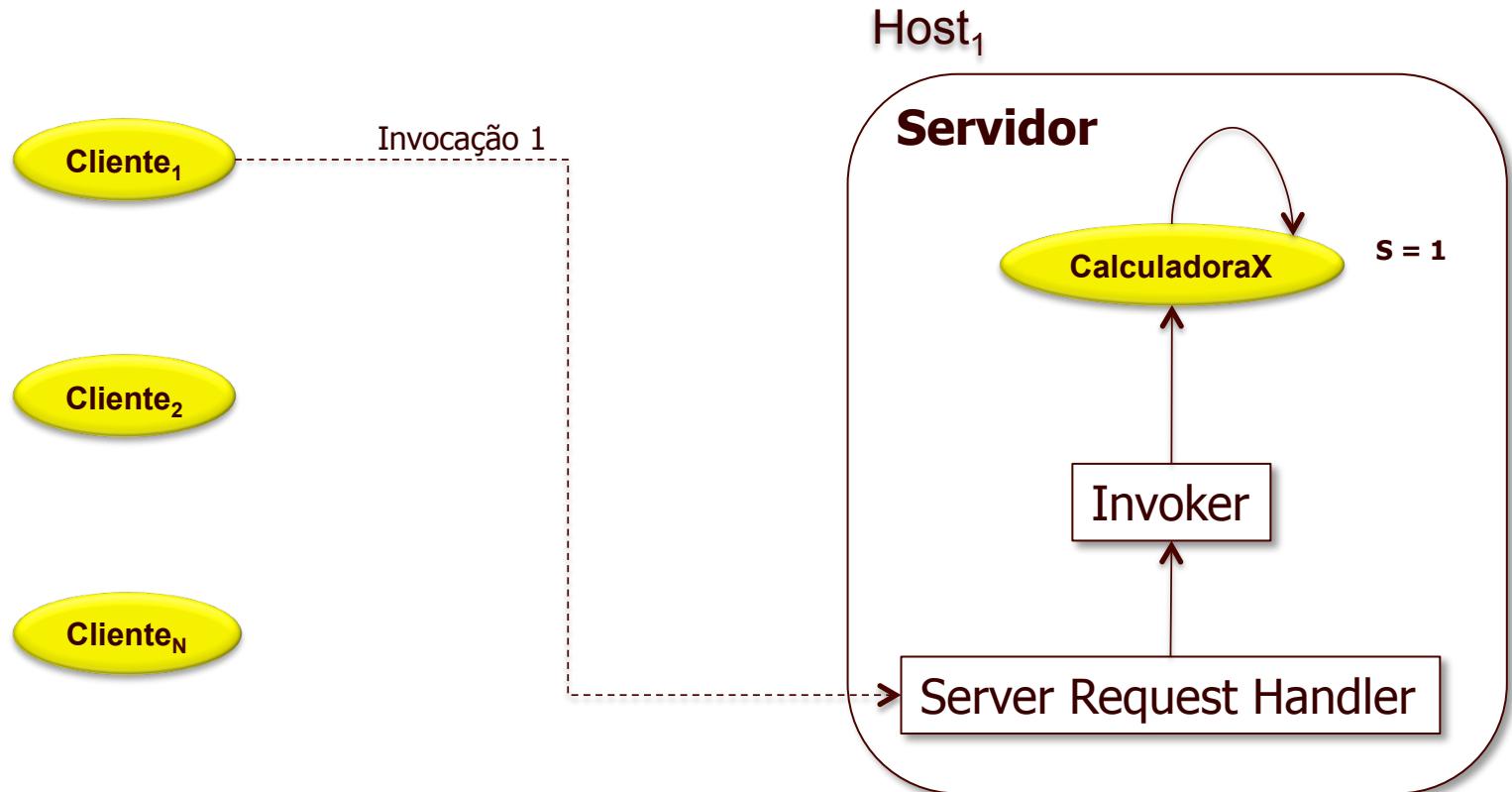
- Criar uma *Static instance* de um objeto remoto independente do **estado** e do **ciclo de vida** do cliente
- **Ativada** antes dos clientes e **desativada** quando o servidor parar

# Lifecycle Management Patterns:: *Static Instance*



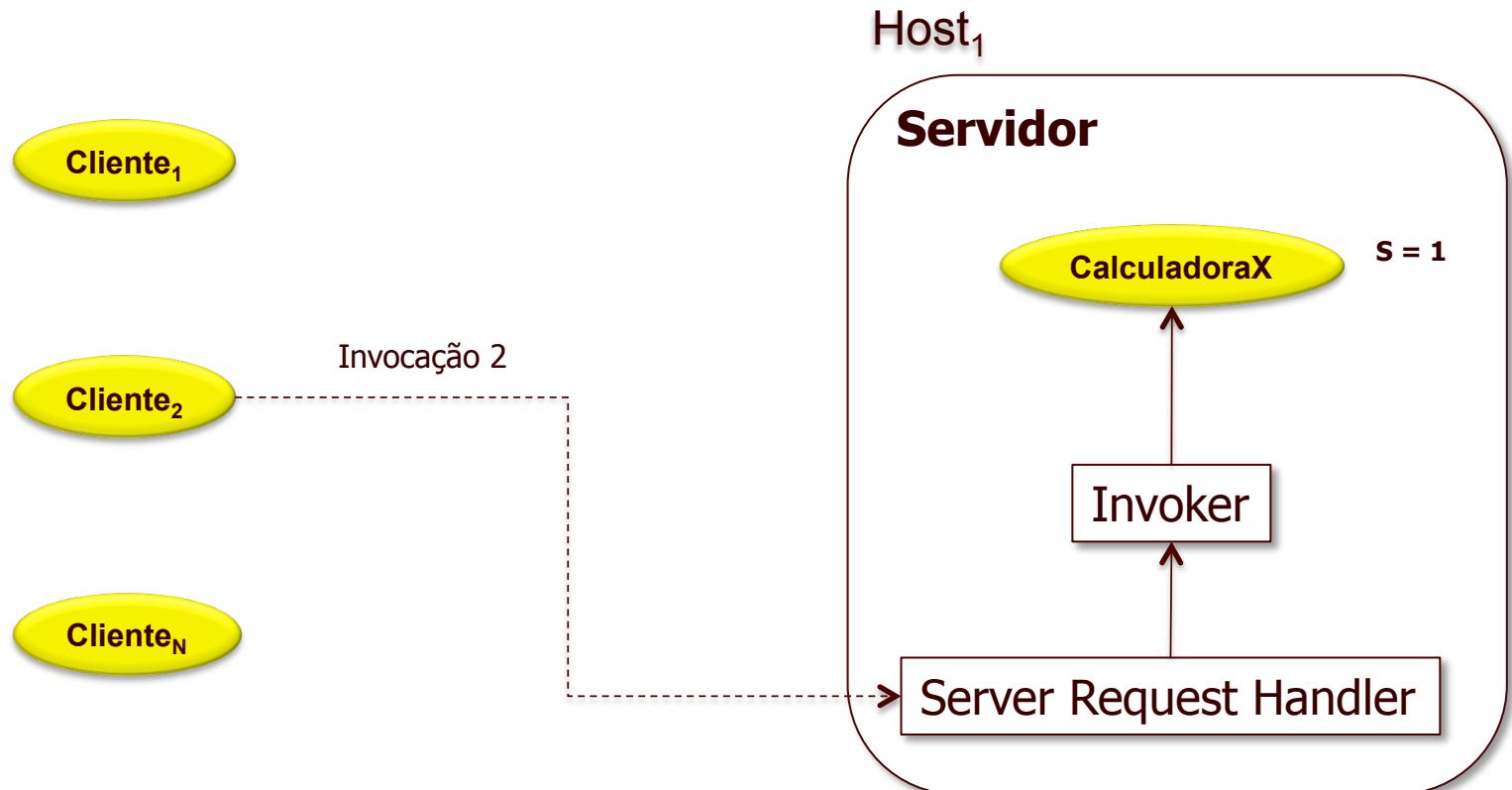
- ✓ Os recursos “consumidos” pela instância ficam **permanentemente alocados** enquanto o servidor estiver ativo.
- ✓ O estado da instância é compartilhado entre todos os clientes.

# Lifecycle Management Patterns:: *Static Instance*



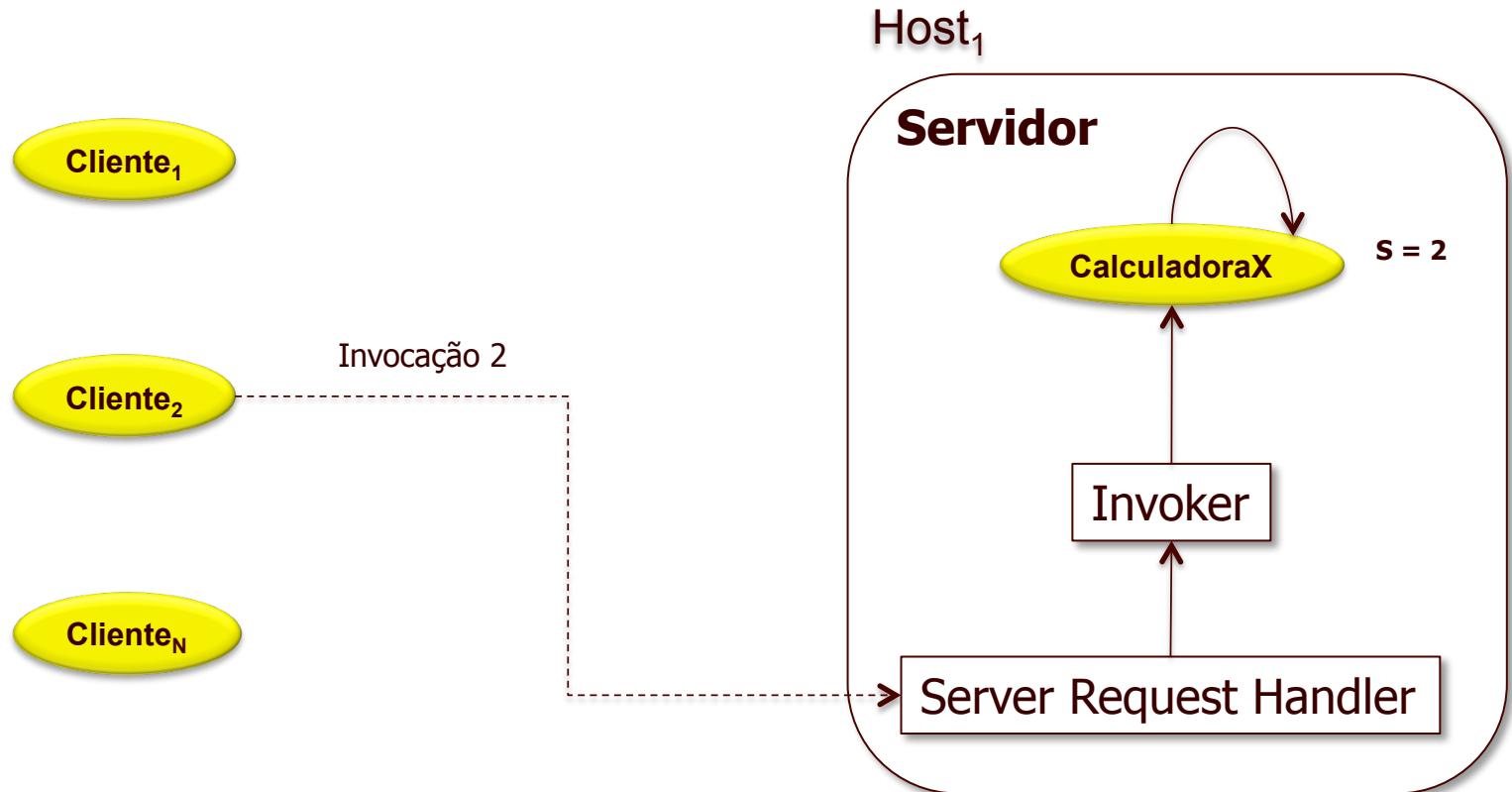
- ✓ Os recursos “consumidos” pela instância ficam **permanentemente alocados** enquanto o servidor estiver ativo.
- ✓ O estado da instância é compartilhado entre todos os clientes.

# Lifecycle Management Patterns:: *Static Instance*



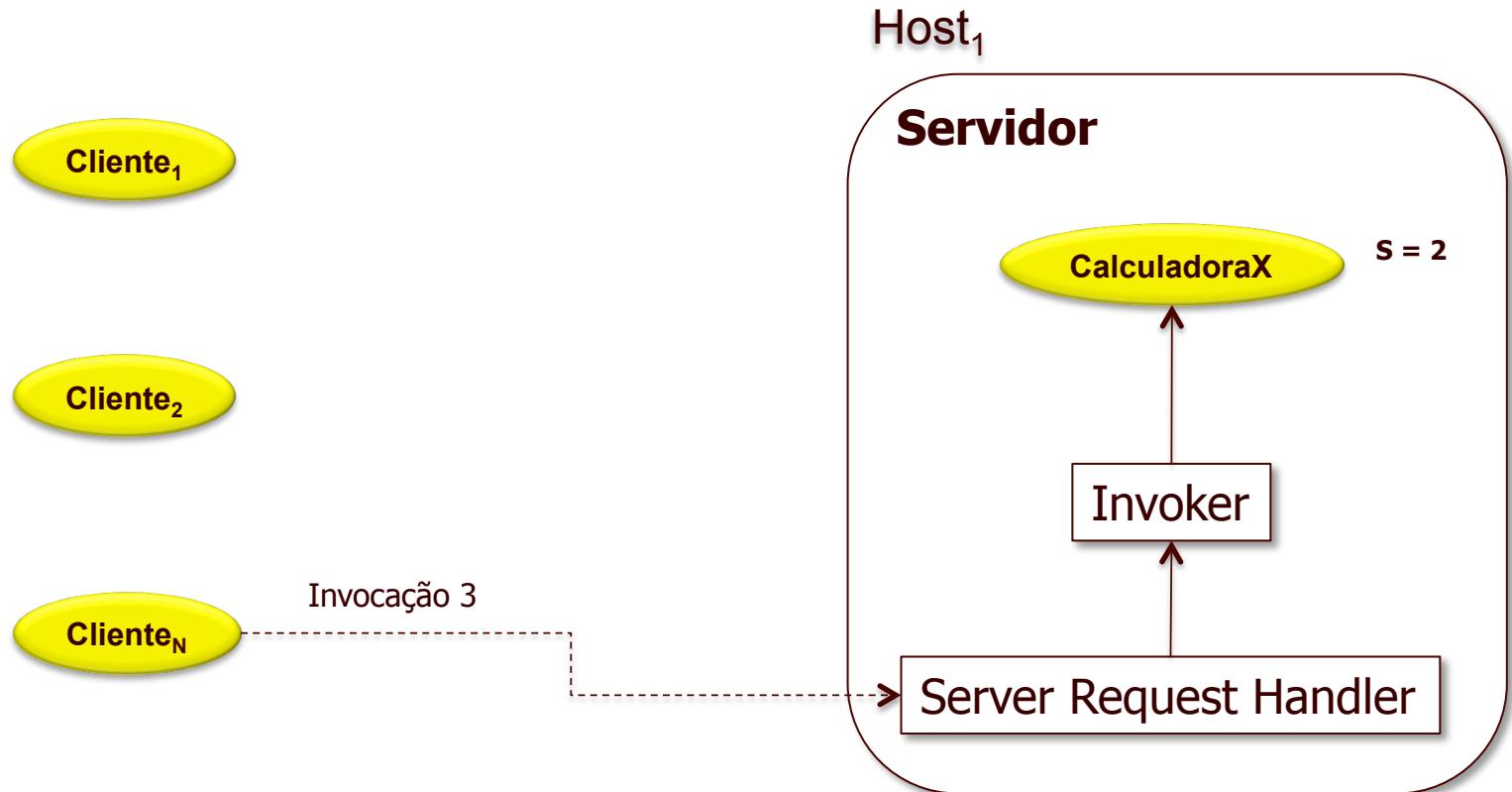
- ✓ Os recursos “consumidos” pela instância ficam **permanentemente alocados** enquanto o servidor estiver ativo.
- ✓ O estado da instância é compartilhado entre todos os clientes.

# Lifecycle Management Patterns:: *Static Instance*



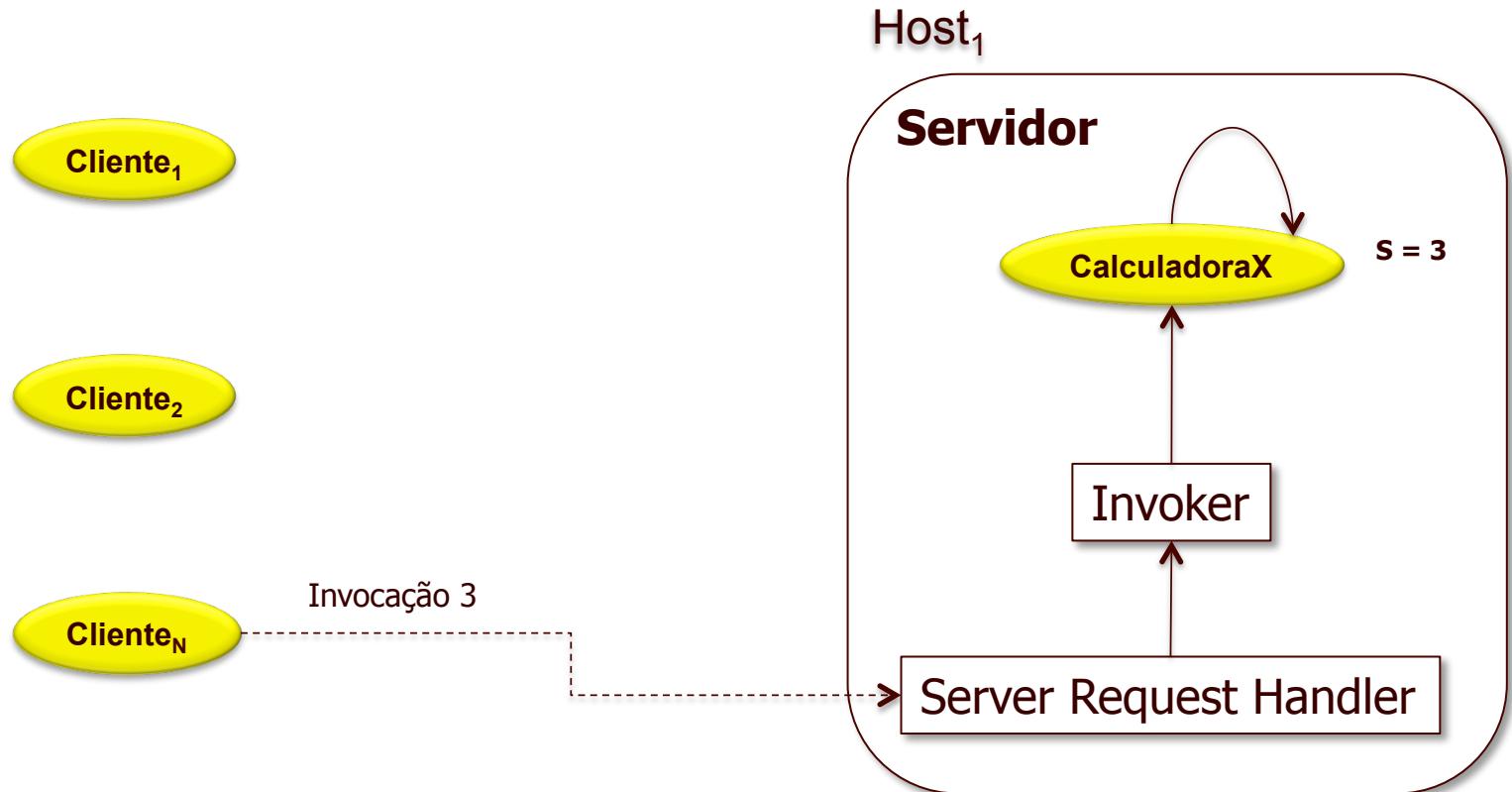
- ✓ Os recursos “consumidos” pela instância ficam **permanentemente alocados** enquanto o servidor estiver ativo.
- ✓ O estado da instância é compartilhado entre todos os clientes.

# Lifecycle Management Patterns:: *Static Instance*



- ✓ Os recursos “consumidos” pela instância ficam **permanentemente alocados** enquanto o servidor estiver ativo.
- ✓ O estado da instância é compartilhado entre todos os clientes.

# Lifecycle Management Patterns:: *Static Instance*



- ✓ Os recursos “consumidos” pela instância ficam **permanentemente alocados** enquanto o servidor estiver ativo.
- ✓ O estado da instância é compartilhado entre todos os clientes.

# Lifecycle Management Patterns:: *Per-Request Instance*

---

## ■ Contexto

- Quando o objeto remoto provê um serviço que não requer a manutenção do estado.

## ■ Problema

- Como melhorar a escalabilidade do acesso a objetos remotos?

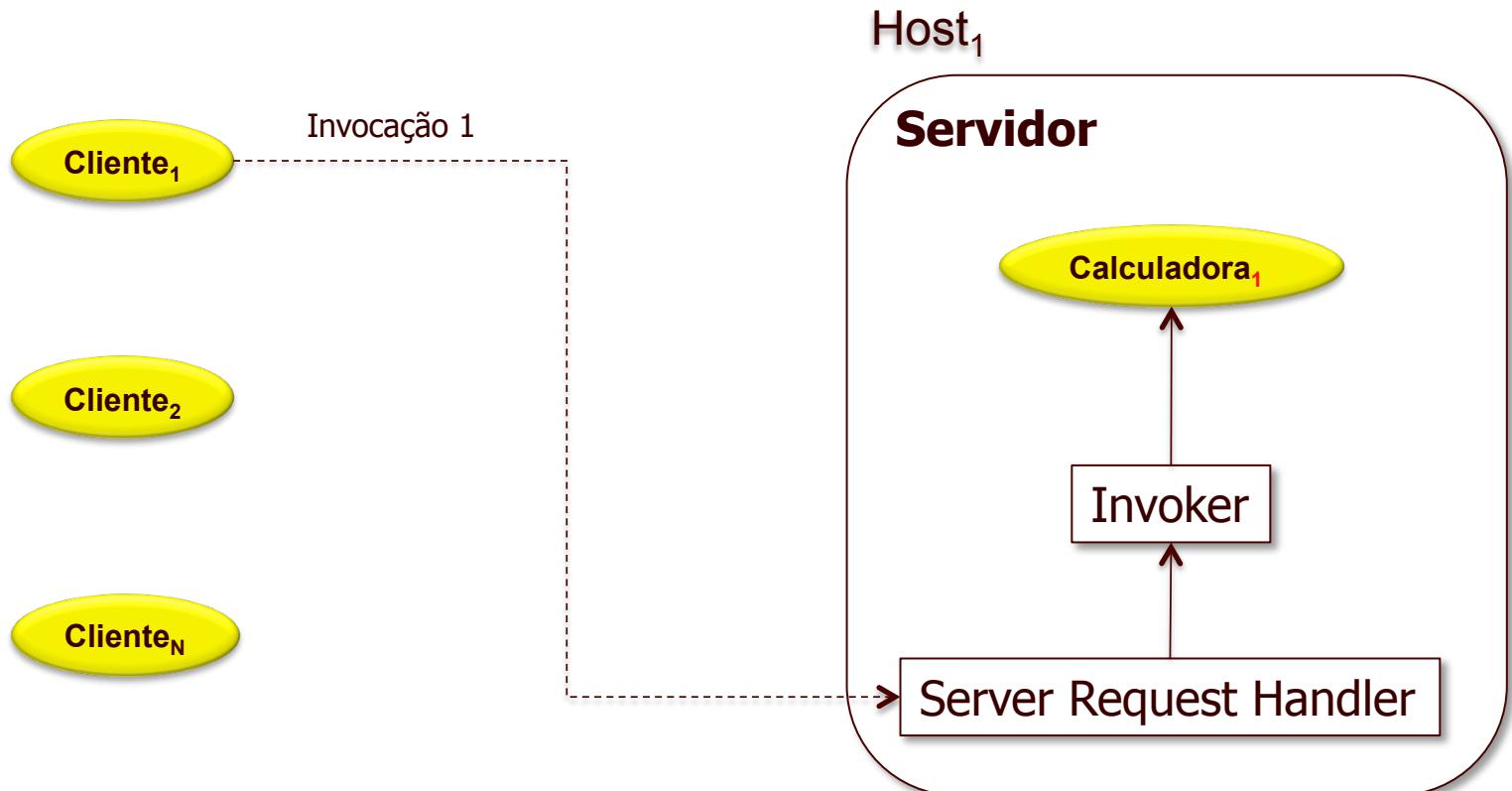
## ■ Solução 1

- Usar várias *Static instances*

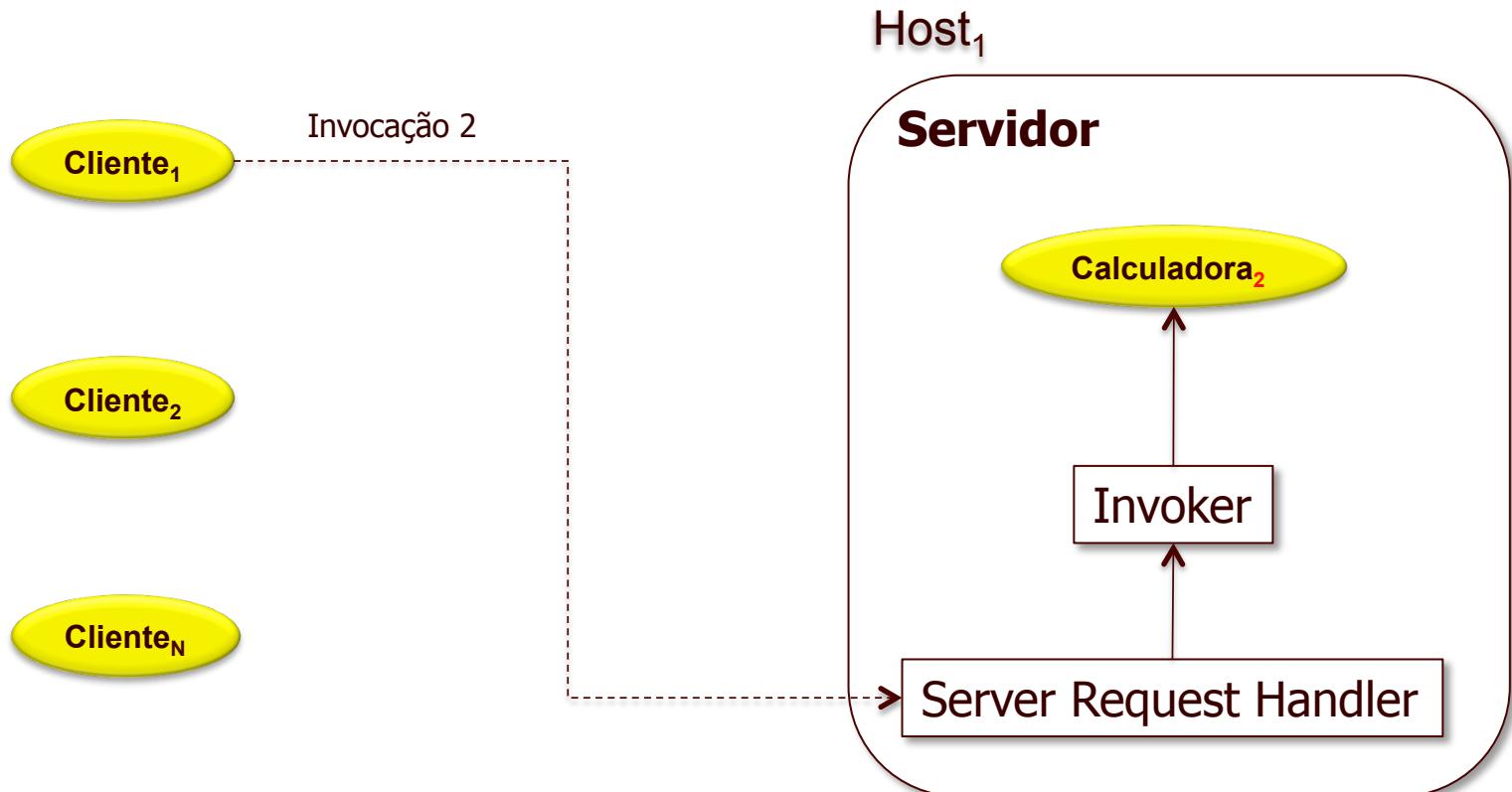
## ■ Solução 2

- Ativar um novo *servant* para cada nova invocação

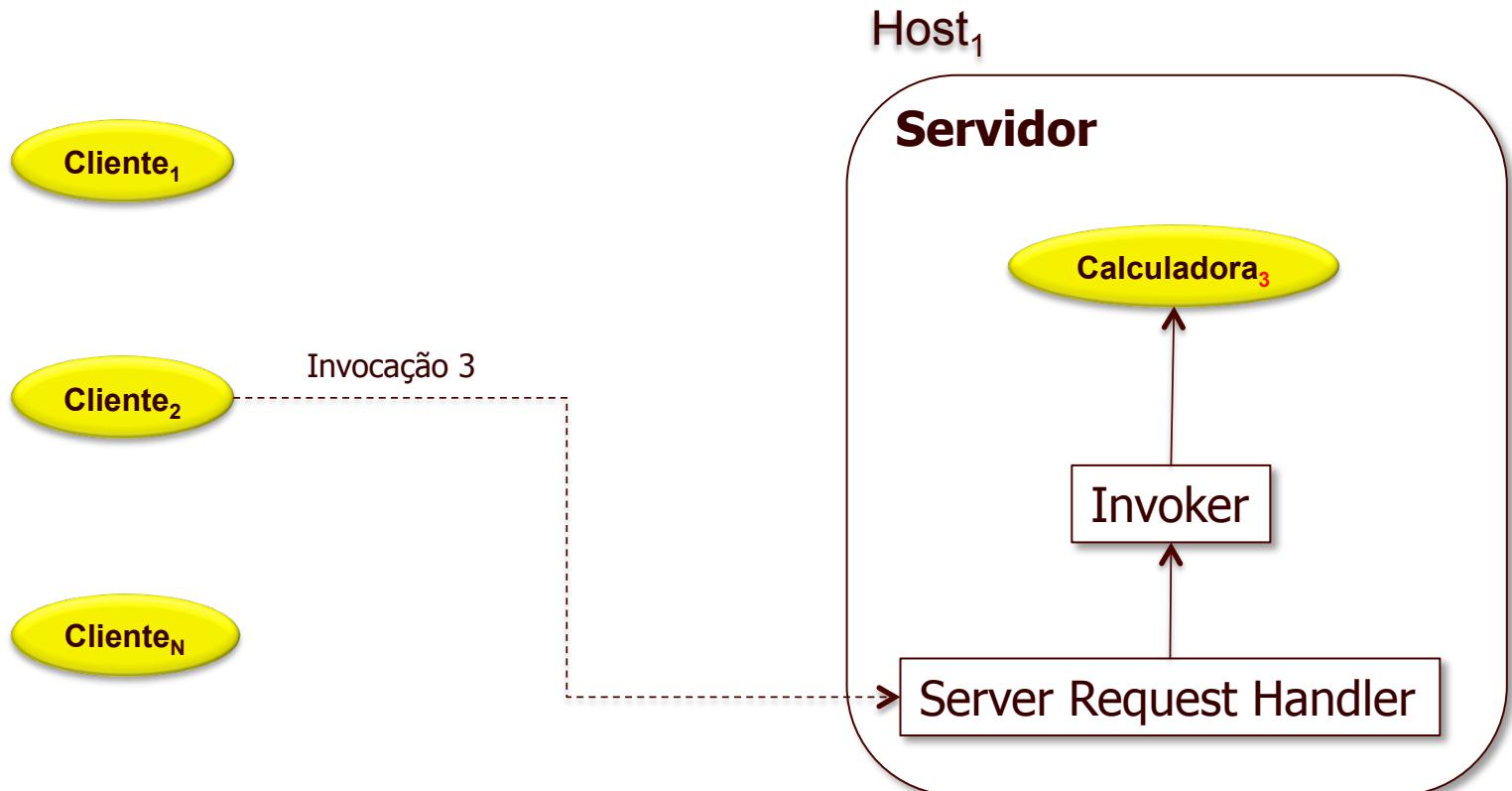
# Lifecycle Management Patterns:: *Per-Request Instance*



# Lifecycle Management Patterns:: *Per-Request Instance*



# Lifecycle Management Patterns:: *Per-Request Instance*



# Lifecycle Management Patterns:: *Client-Dependent Instance*

---

## ■ Contexto

- Clientes usam serviços fornecidos por objetos remotos.

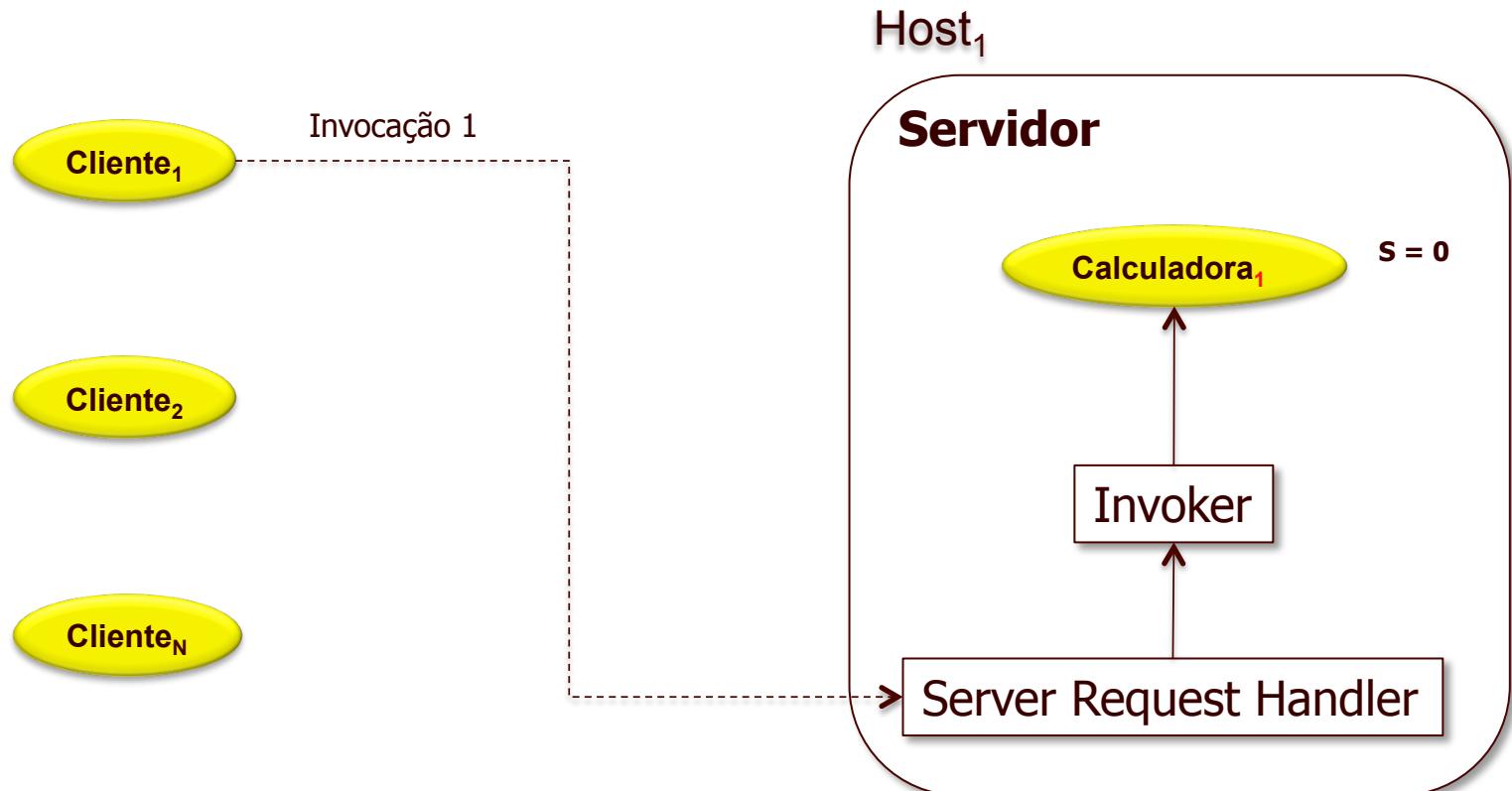
## ■ Problema

- Em situações onde o “objeto remoto estende a lógica do cliente”, onde manter o estado compartilhado do cliente/ objeto remoto?

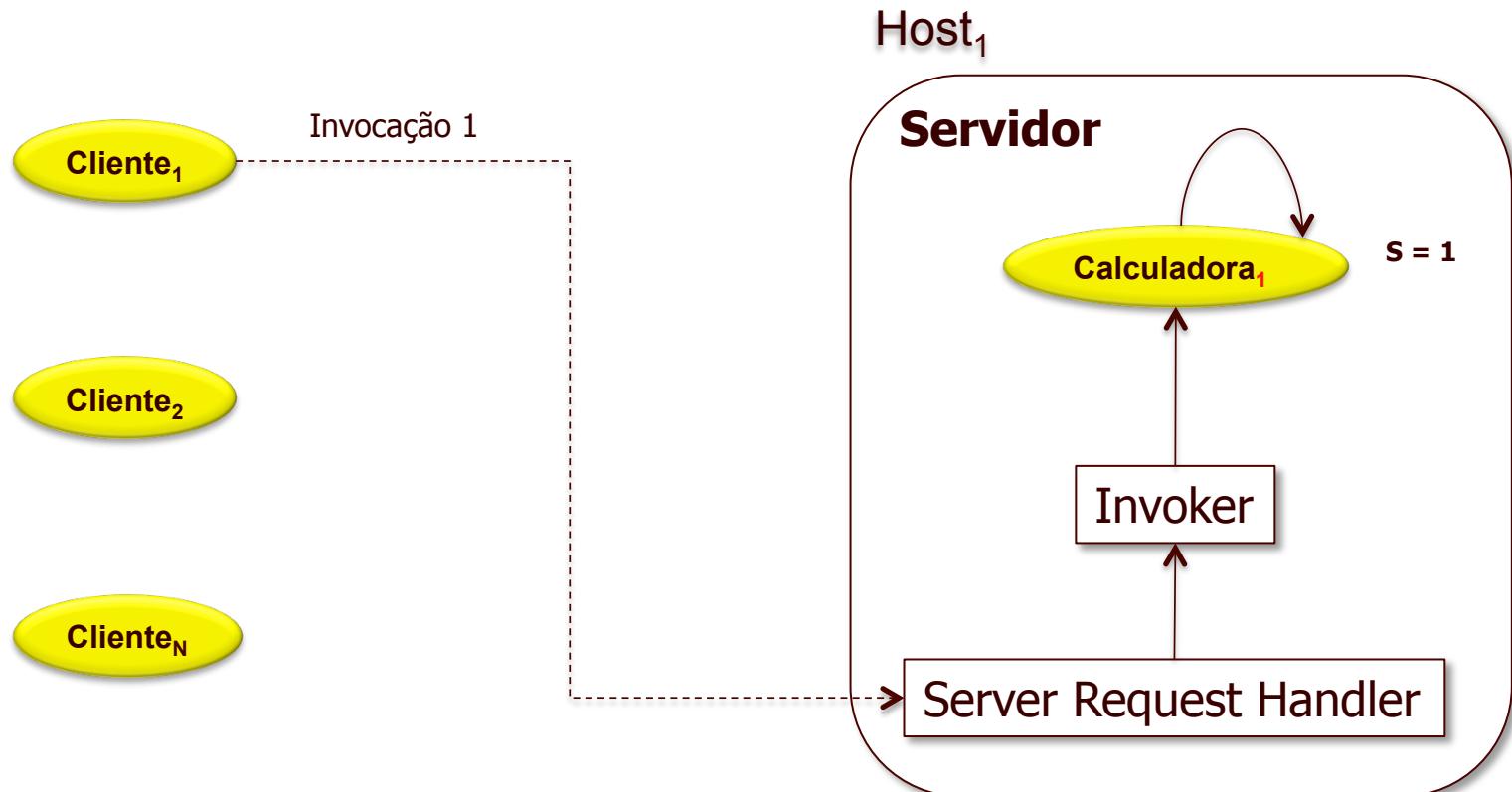
## ■ Solução

- Fornecer um objeto remoto cujo tempo de vida é **controlado pelo cliente**.
- Cliente **cria** *Client-dependent Instance* na invocação e **destrói** quando ela não for mais necessária.

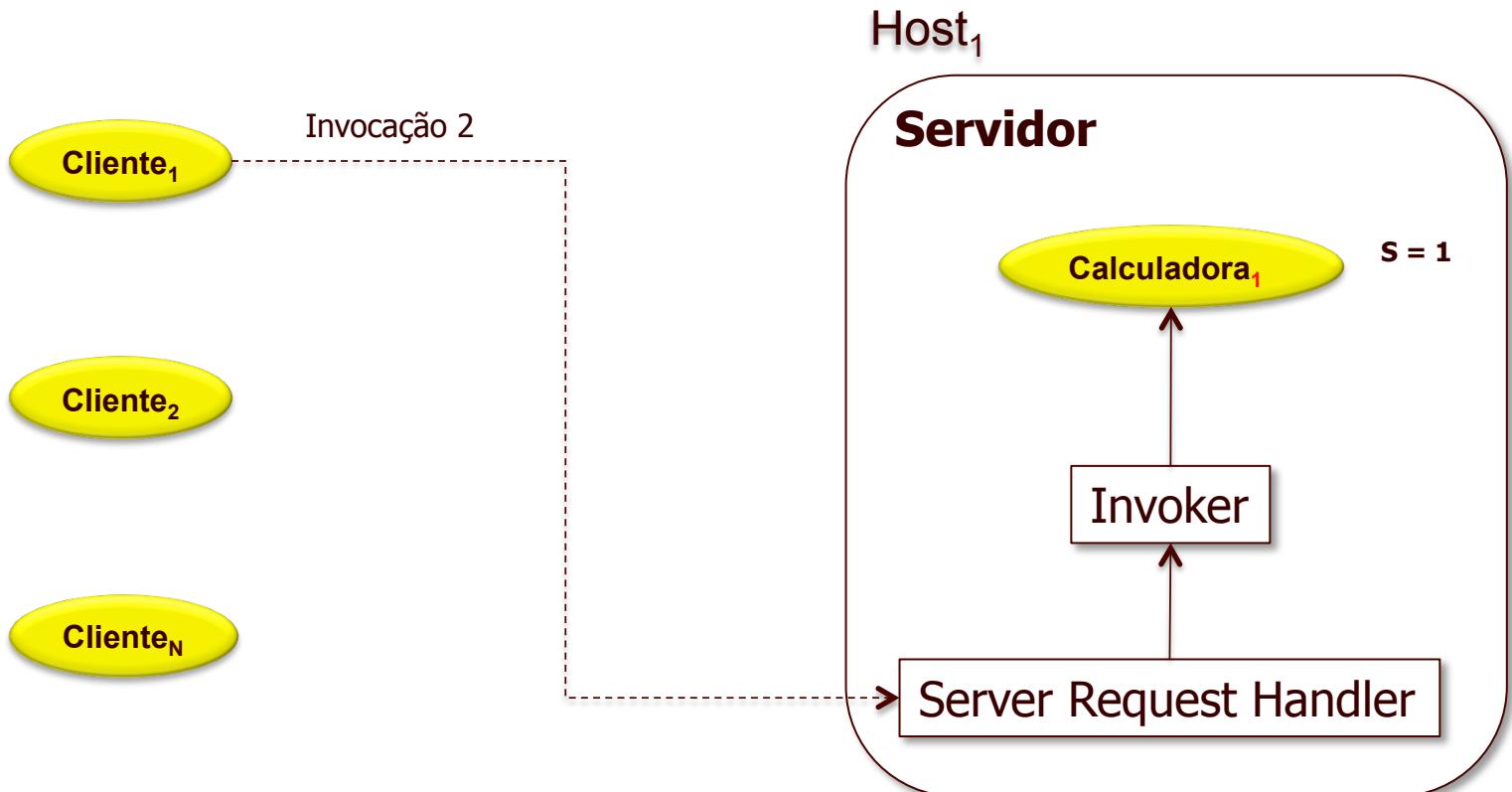
# Lifecycle Management Patterns:: *Client-Dependent Instance*



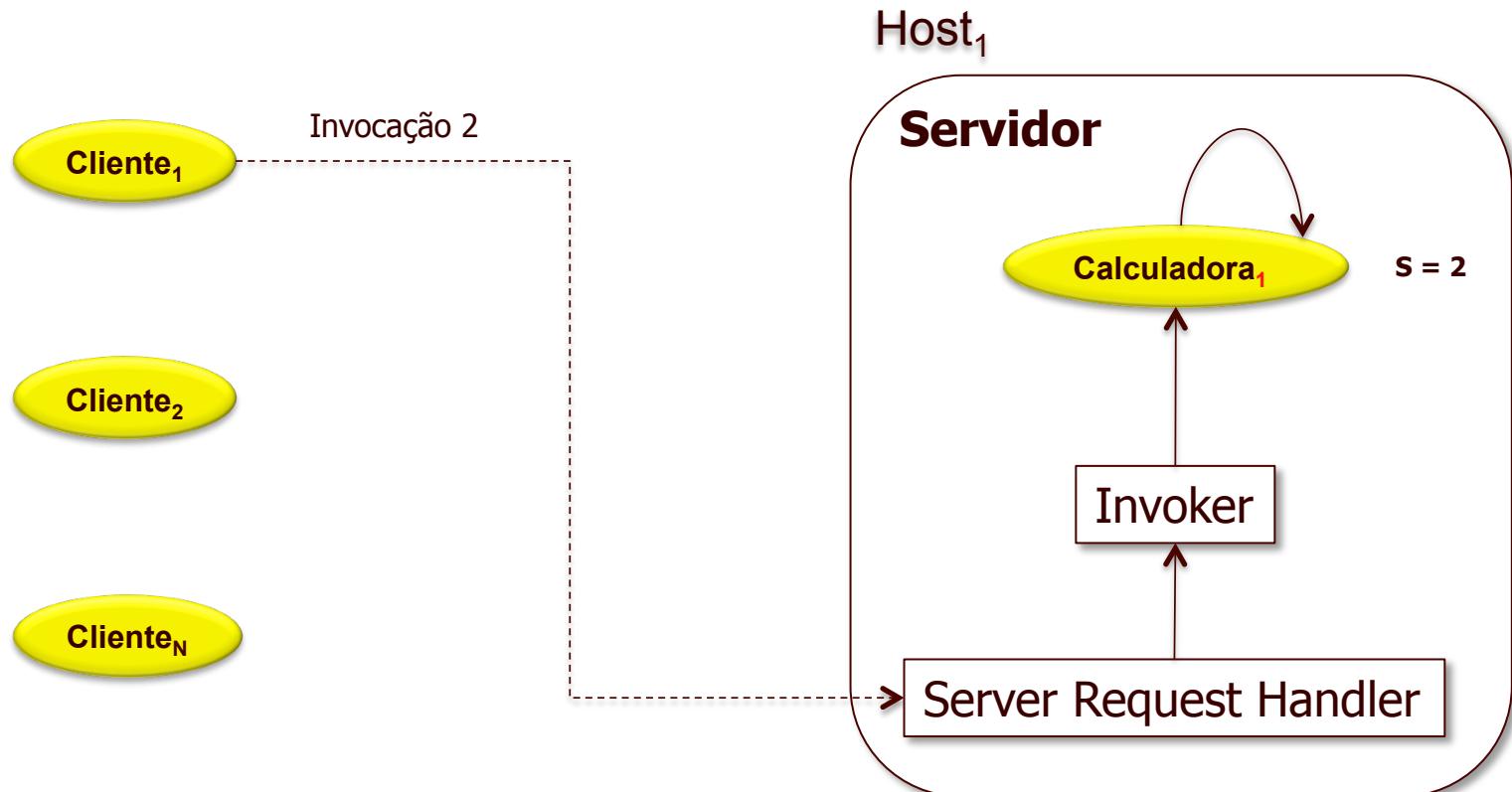
# Lifecycle Management Patterns:: *Client-Dependent Instance*



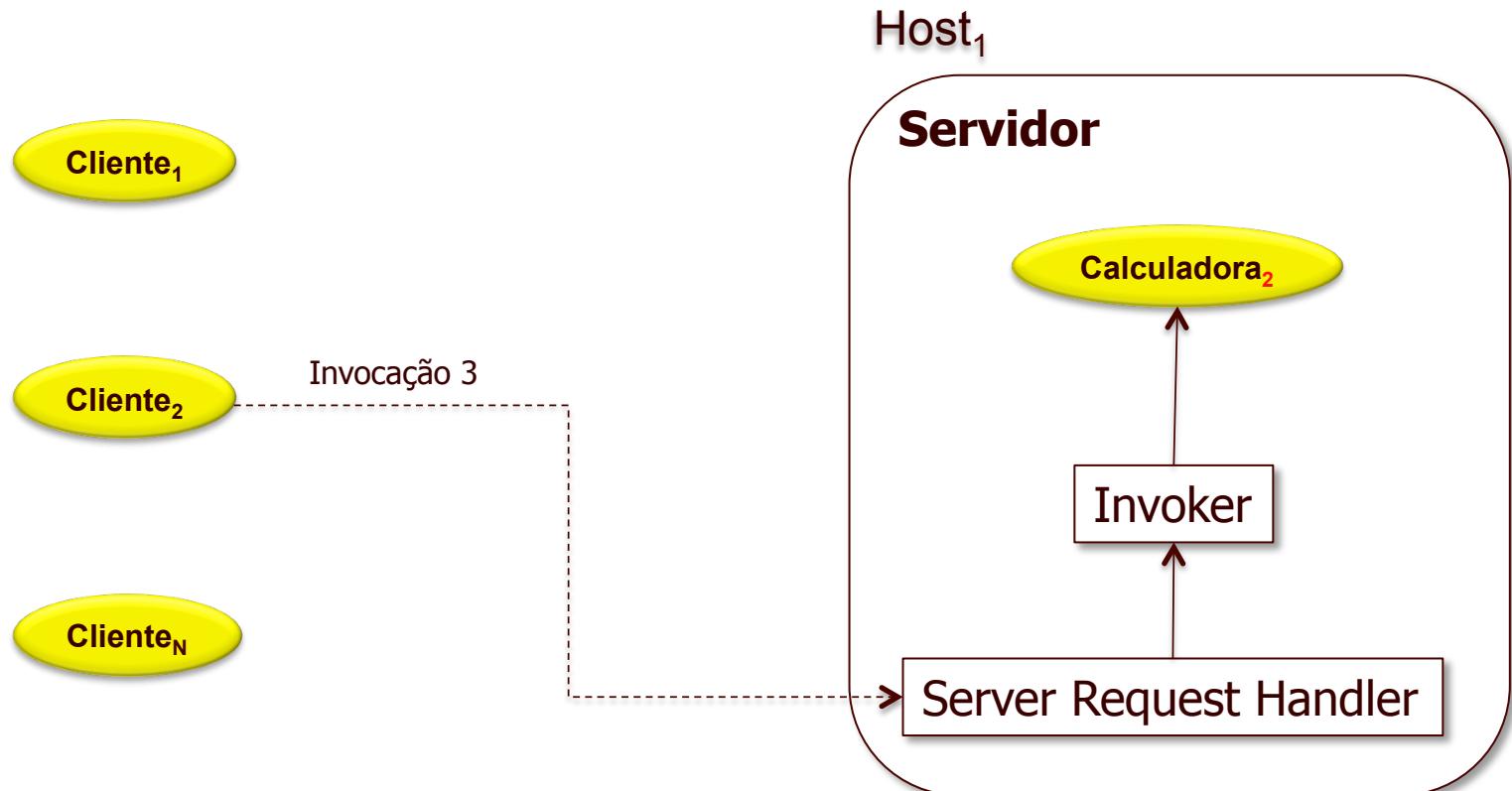
# Lifecycle Management Patterns:: *Client-Dependent Instance*



# Lifecycle Management Patterns:: *Client-Dependent Instance*



# Lifecycle Management Patterns:: *Client-Dependent Instance*



# Lifecycle Management Patterns:: *Lazy Acquisition*

---

## ■ Contexto

- As *Static Instances* precisam ser gerenciadas eficientemente (memória e processamento).

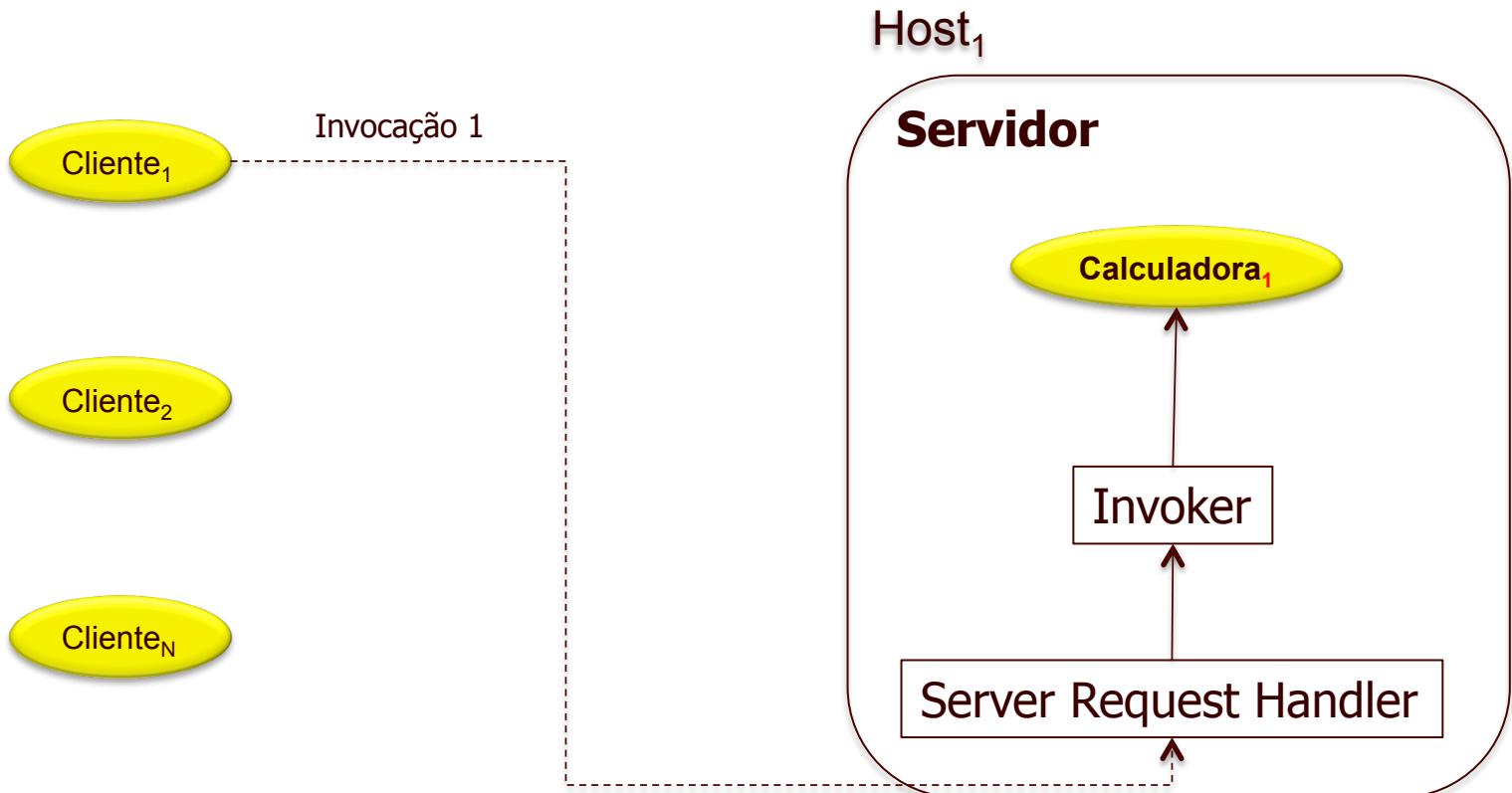
## ■ Problema

- Como gerenciar as *Static Instances* eficientemente?

## ■ Solução

- Instanciar os *servants* apenas quando os objetos remotos forem invocados pelo cliente

# Lifecycle Management Patterns:: *Lazy Acquisition*



# Lifecycle Management Patterns:: *Pooling*

---

## ■ Contexto

- Toda instância do objeto remoto consome recursos do servidor.

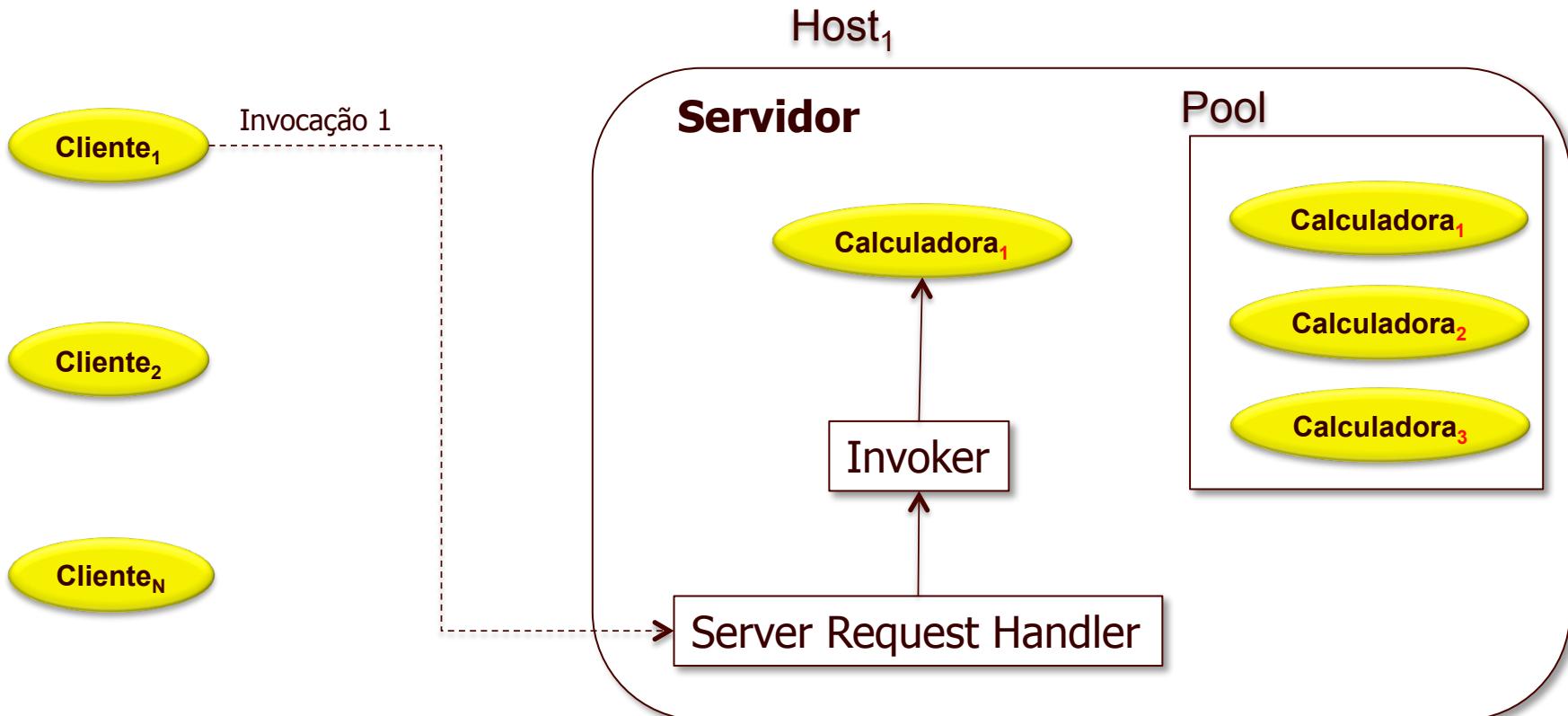
## ■ Problema

- Como gerenciar (criar/destruir) *servants* de forma eficiente?
- Considerando que: cada *servant* precisa ser (i) criado (memória precisa ser alocada), (ii) inicializado e (iii) registrado

## ■ Solução

- Criar um *pool* de *servants* para cada objeto remoto.
- Para cada invocação do objeto remoto:
  - (i) um *servant* é obtido do *pool*,
  - [(ii) o estado do objeto remoto é “colocado” no *servant*,]
  - (iii) o *servant* “atende” a invocação,
  - [(iv) o estado do objeto remoto é “persistido”,] e
  - (v) o *servant* é colocado de volta no *pool*.

# Lifecycle Management Patterns:: *Pooling*



# Lifecycle Management Patterns:: *Leasing*

---

## ■ Contexto

- Clientes consomem recursos do servidor, e.g., através do uso de *Client-Dependent Instances*

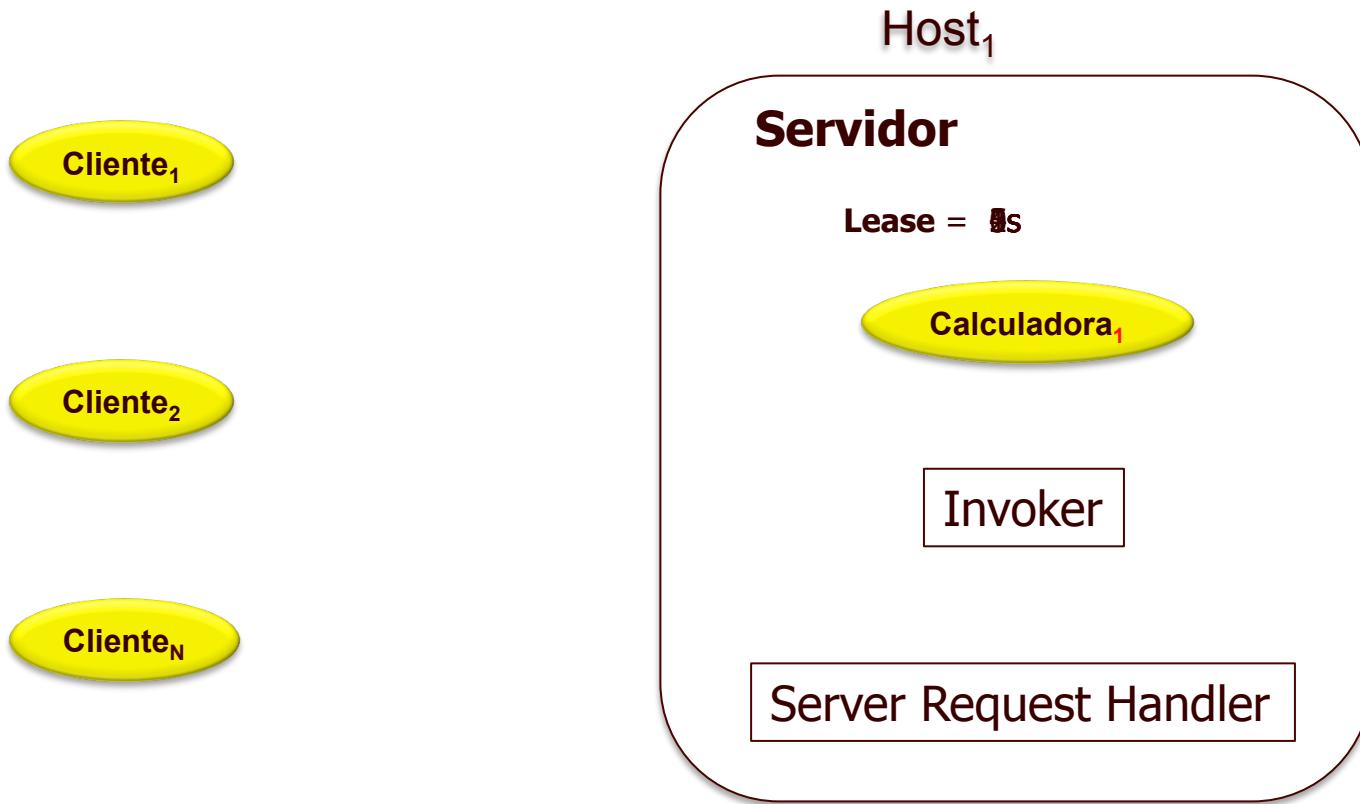
## ■ Problema

- Em que momento objetos remotos e seus *servants* podem ser “liberados” para não mais consumirem recursos do servidor?
- *Lifecycle Manager* e objetos remotos não sabem quando os clientes irão acessá-los.

## ■ Solução

- Associar um **tempo de vida** (*lease*) para o uso do objeto remoto pelo cliente.
- Quando todos os *leases* tiverem **expirado**, o objeto remoto pode ser **destruído**

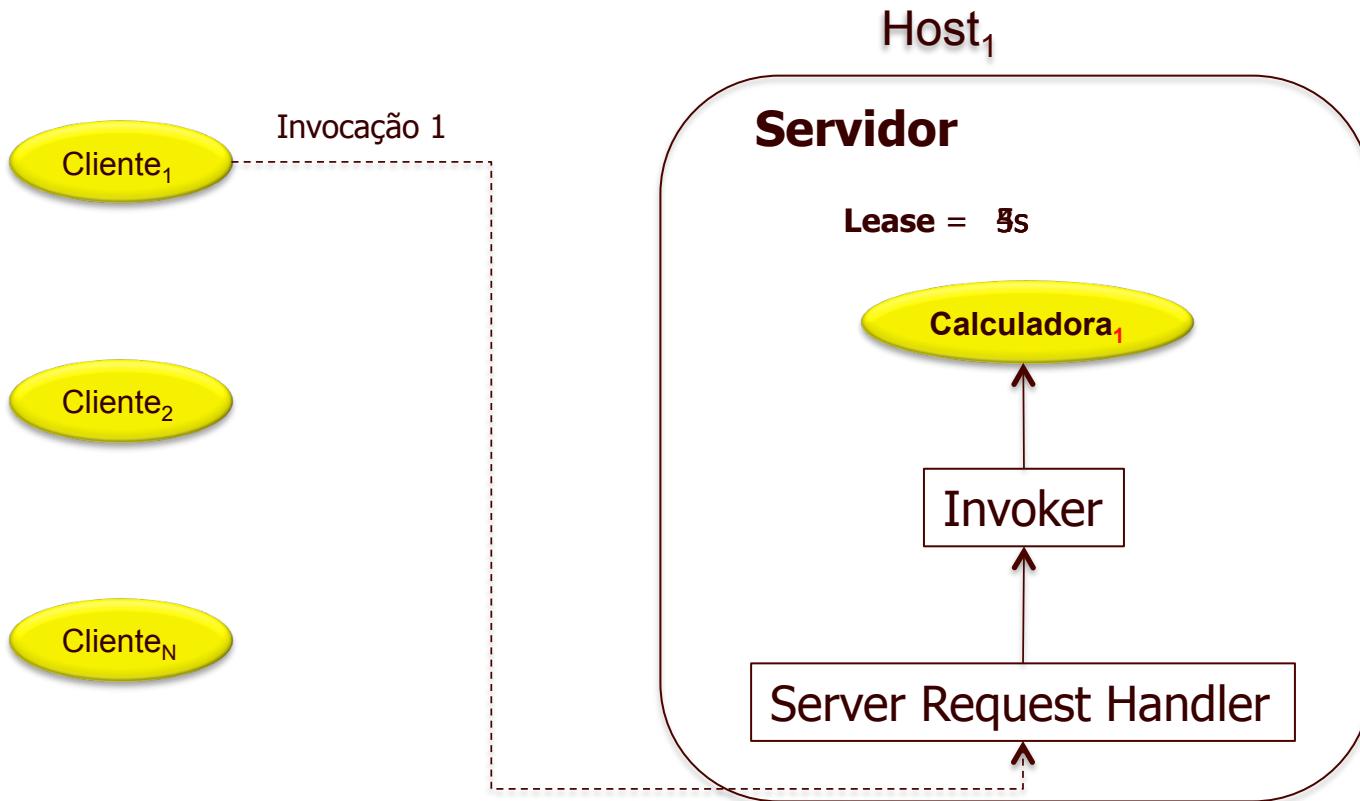
# Lifecycle Management Patterns:: *Leasing*



Renovação do *lease*:

1. O *lease* é renovado a cada **invocação**, i.e., o *lease* precisa ser maior do que o intervalo entre as invocações.
2. **Cliente** renova o *lease* explicitamente.
3. **Middleware** avisa que o *lease* irá expirar e o cliente define uma extensão do *lease*.

# Lifecycle Management Patterns:: *Leasing*



Renovação do *lease*:

1. O *lease* é renovado a cada **invocação**, i.e., o *lease* precisa ser maior do que o intervalo entre as invocações.
2. **Cliente** renova o *lease* explicitamente.
3. **Middleware** avisa que o *lease* irá expirar e o cliente define uma extensão do *lease*.

# Lifecycle Management Patterns:: *Passivation*

---

## ■ Contexto

- A aplicação servidora fornece objetos remotos *stateful* (*Client-dependent Instance* e *Static Instances*)

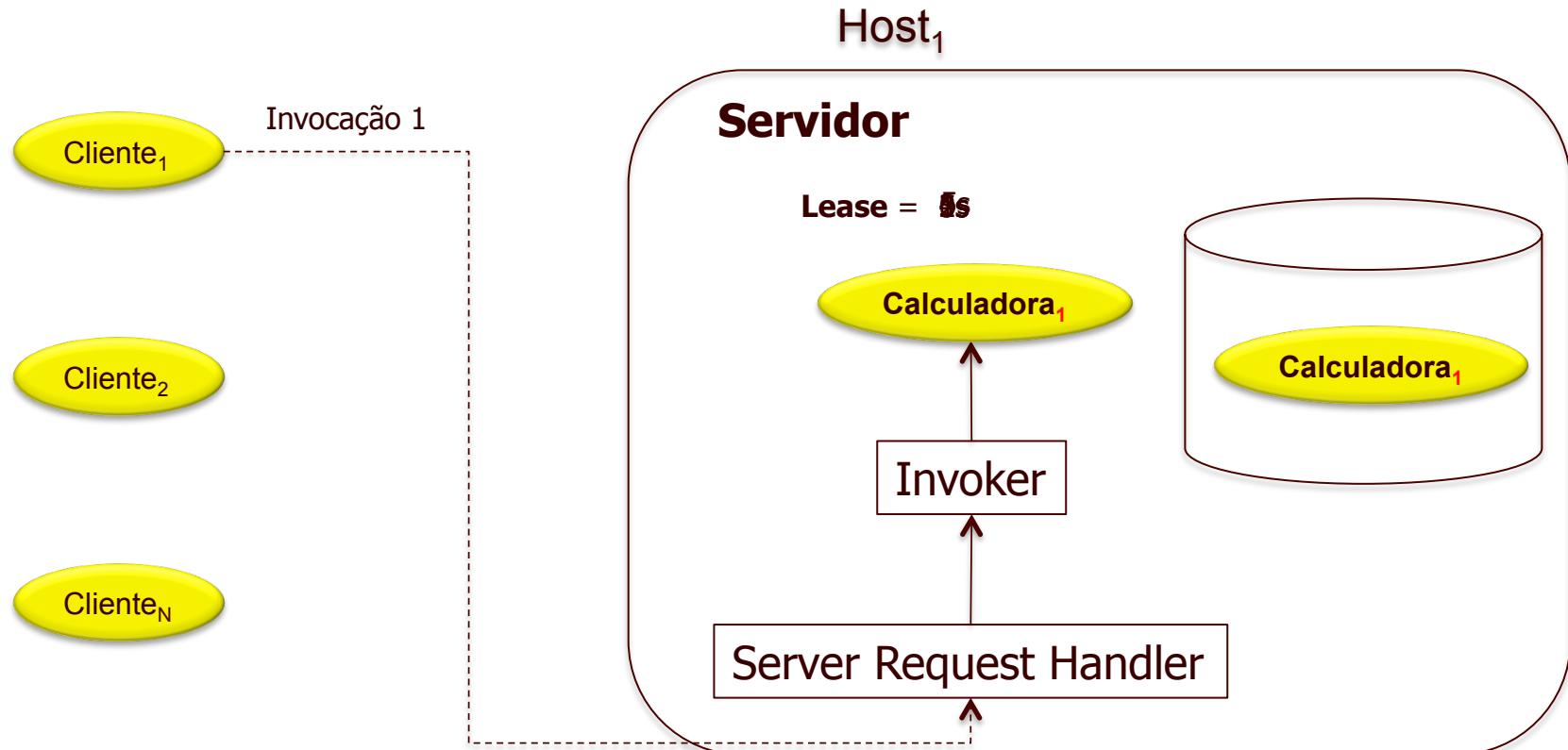
## ■ Problema

- Como evitar que *servants* (instâncias dos objetos remotos em execução) consumam recursos quando há longos períodos sem que sejam invocados?

## ■ Solução

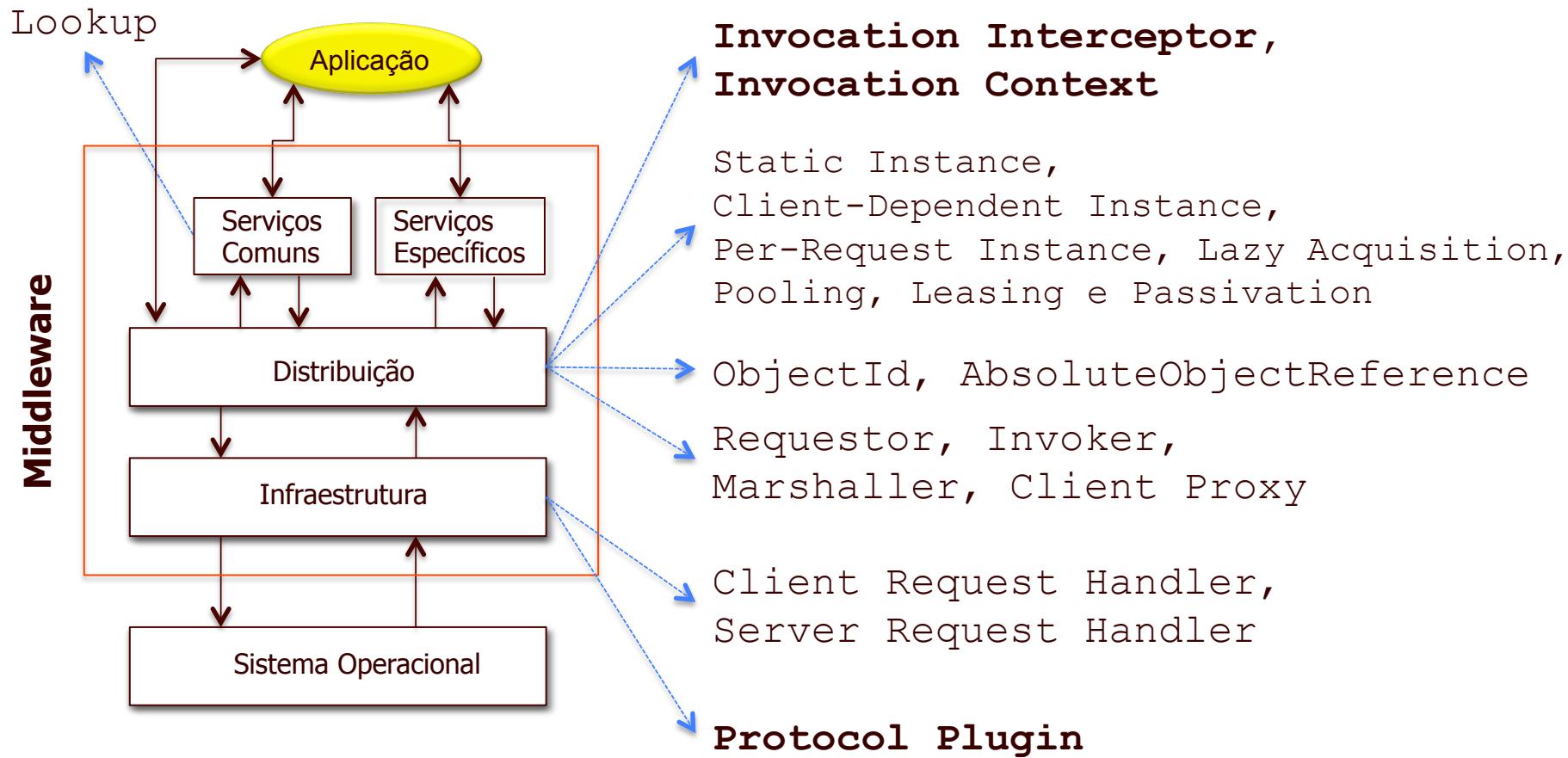
- Remover os *servants* da memória quando eles não forem acessados por um período pré-definido.
- Quando o *Invoker* recebe uma invocação para um objeto remoto “passivado”, o *Lifecycle Manager*:
  - Cria um novo *servant*
  - Inicializa o novo *servant* com o estado persistido do objeto

# Lifecycle Management Patterns:: *Passivation*



# Padrões de Extensão

# Remoting Patterns:: Padrões de Extensão



# Padrões de Extensão

## ■ Assume que o middleware é organizado em camadas

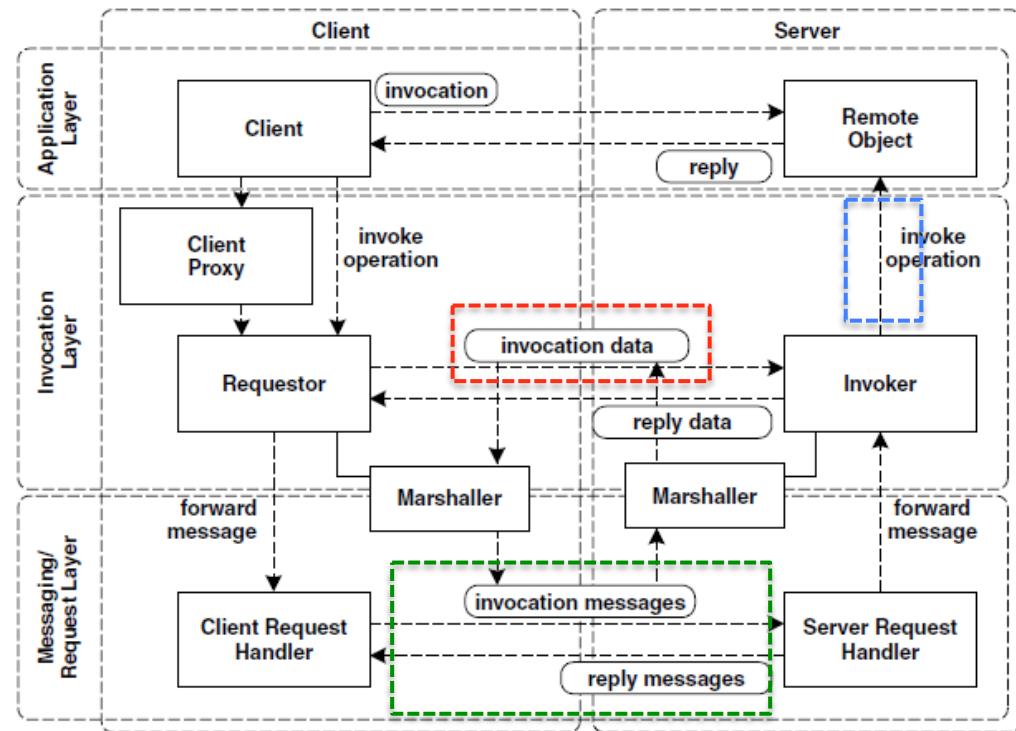
- Invocação
- Mensageria

## ■ Cada camada depende apenas das camadas adjacentes

## ■ As extensões são feitas estendendo-se uma+ camadas

## ■ 03 padrões

- Invocation Interceptor
- Invocation Context
- Protocol Plug-In



# Padrões de Extensão:: *Invocation Interceptor*

## ■ Contexto

- Aplicações distribuídas precisam transparentemente de novos serviços (e.g., log)

## ■ Problema

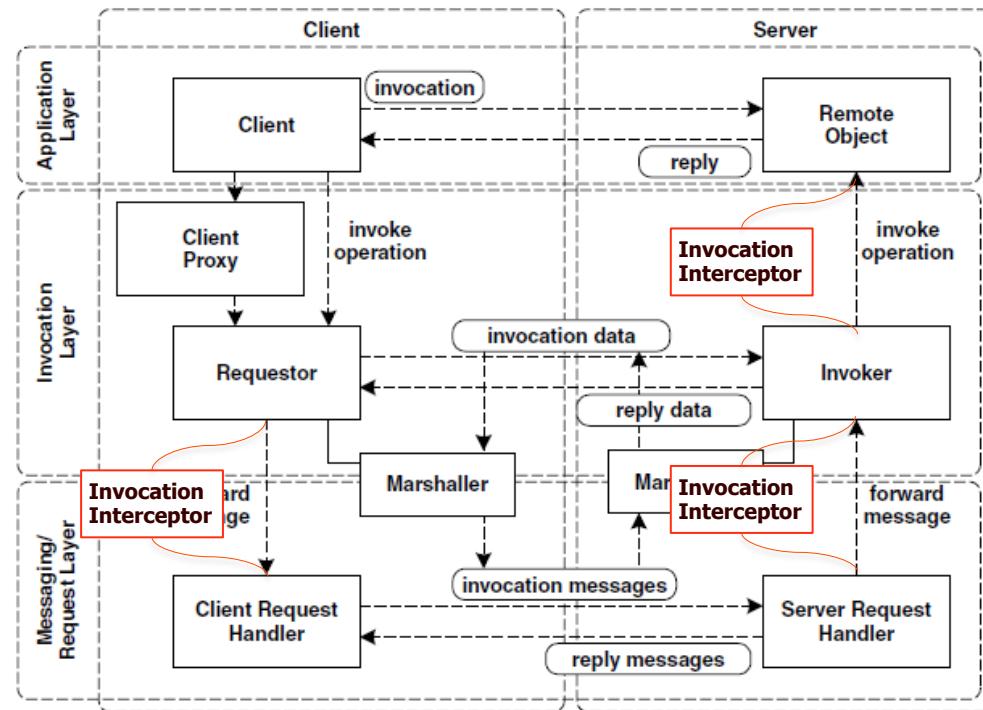
- **Como integrar de forma transparente (sem que clientes e objetos remotos saibam) novos serviços à aplicação distribuída?**

## ■ Solução 1

- Clientes e objetos remotos “cuidam disto” (neste caso não há transparência).

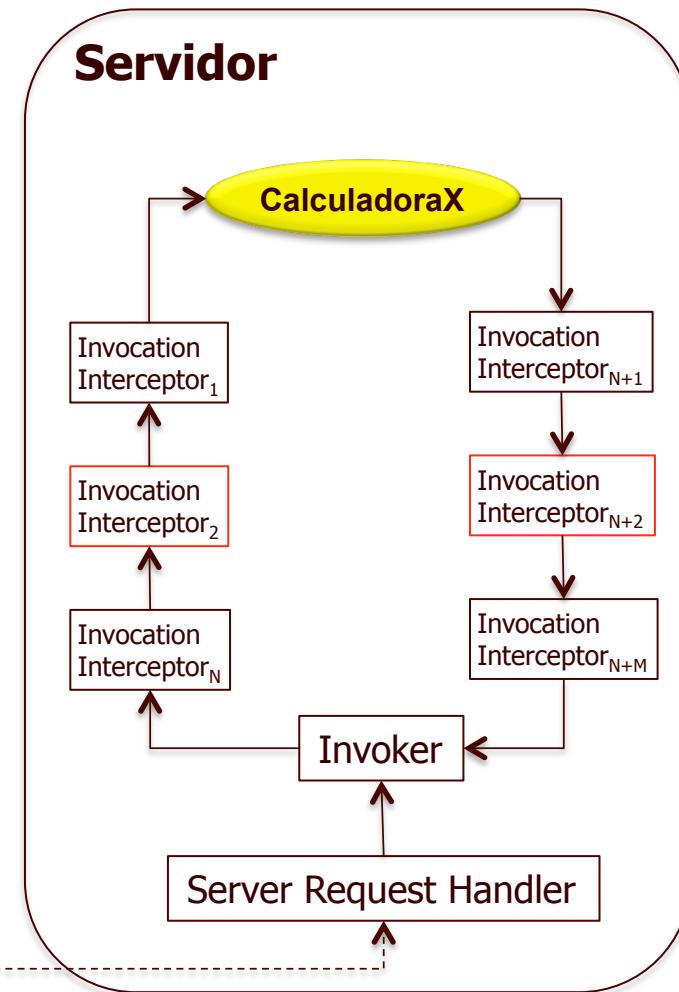
## ■ Solução 2

- Criar “ganchos” no caminho percorrido por uma invocação (e.g., no *Invoker* e *Requestor*) e “*plugar*” *Invocation Interceptors*
- *Invocation Interceptors* podem ser invocados antes/depois da invocação/ resposta



# Padrões de Extensão:: *Invocation Interceptor*

- ✓ Os interceptadores são **independentes** de um cliente/objeto remoto particular
- ✓ Os interceptadores são normalmente dependentes entre si.
- ✓ Estratégias de composição
  - ✓ **Orquestração** (e.g., feita pelo *Invoker*)
  - ✓ **Coreografia** (e.g., cada *Interceptor* invoca o próximo)
- ✓ Os *Invocation Interceptors* são configurados por (i) arquivo de configuração (ii) programaticamente (dinâmica)
- ✓ O uso de Aspectos pode ser uma alternativa ao *Invocation Interceptors*



# Padrões de Extensão:: *Invocation Context*

## ■ Contexto

- Serviços são plugados no middleware através de *Invocation Interceptors*.

## ■ Problema

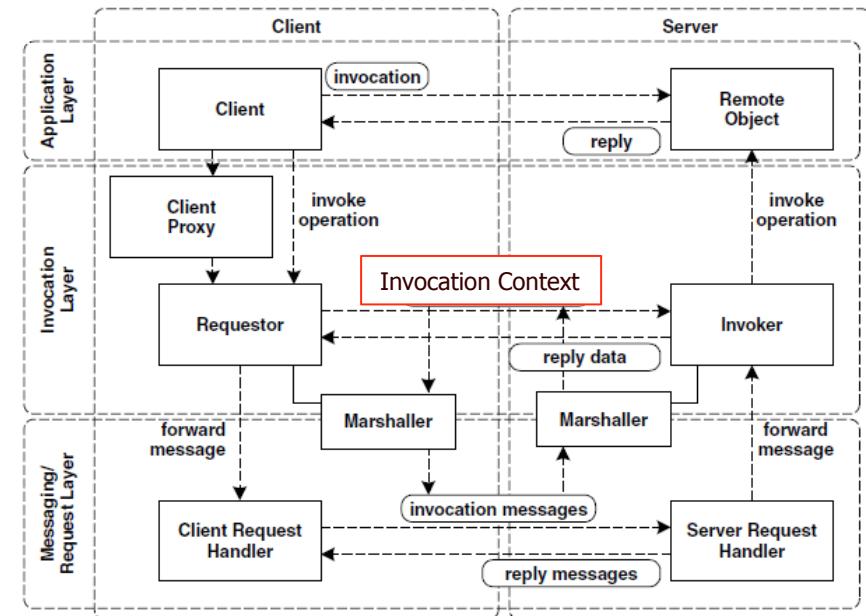
- **Como inserir informações (e.g., credenciais) que possam ser usadas por *Invocation Interceptors*?**
- Considerando que: as informações que constam em uma invocação incluem normalmente o nome da **operação**, o **Object Id** e **parâmetros**.

## ■ Solução 1

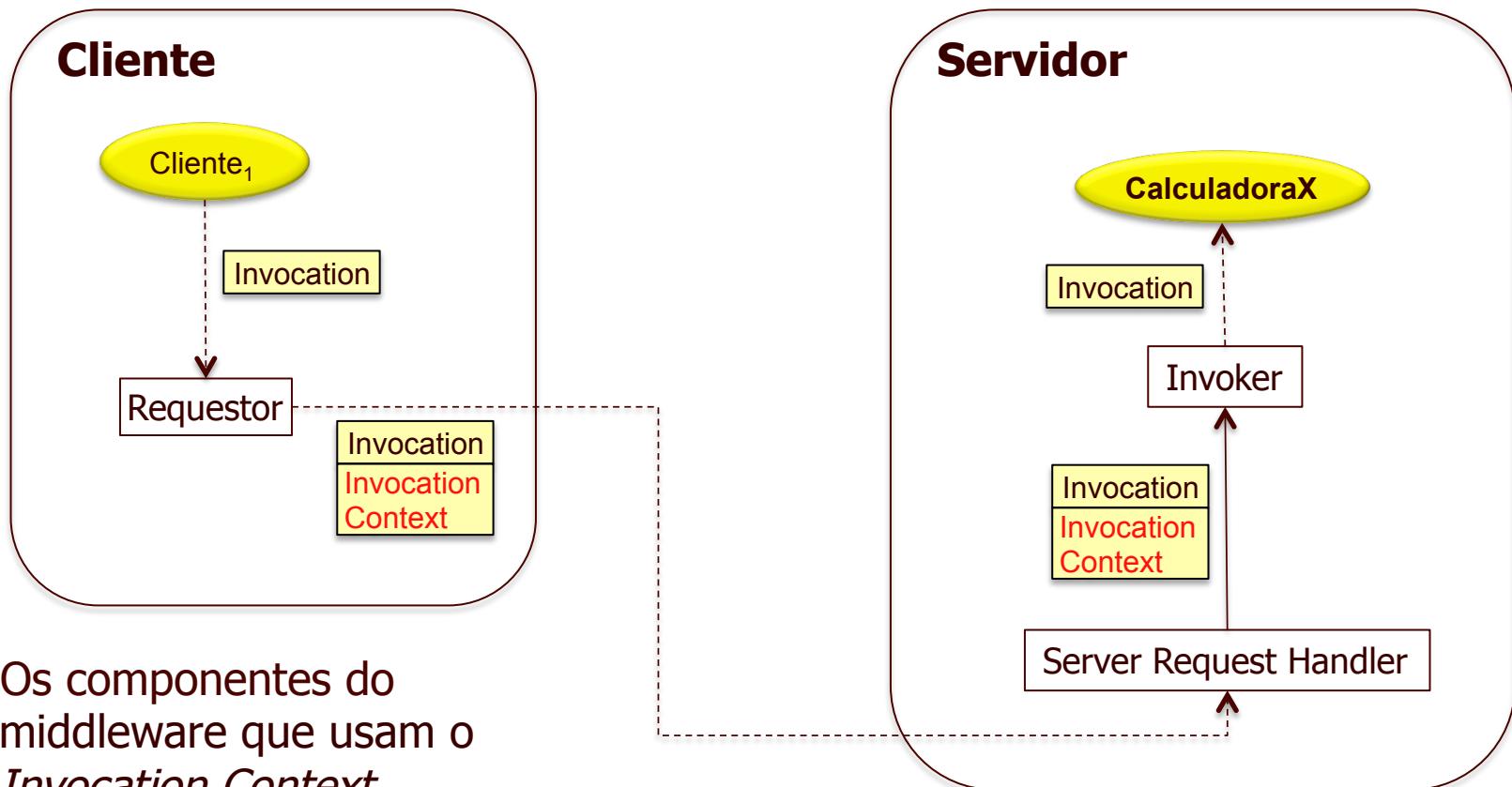
- Adicionar estas informações nas assinaturas das operações.

## ■ Solução 2

- Inserir informações de contexto (*Invocation Context*) em toda invocação usando *Invoker*, *Requestor* ou *Invocation Interceptors*

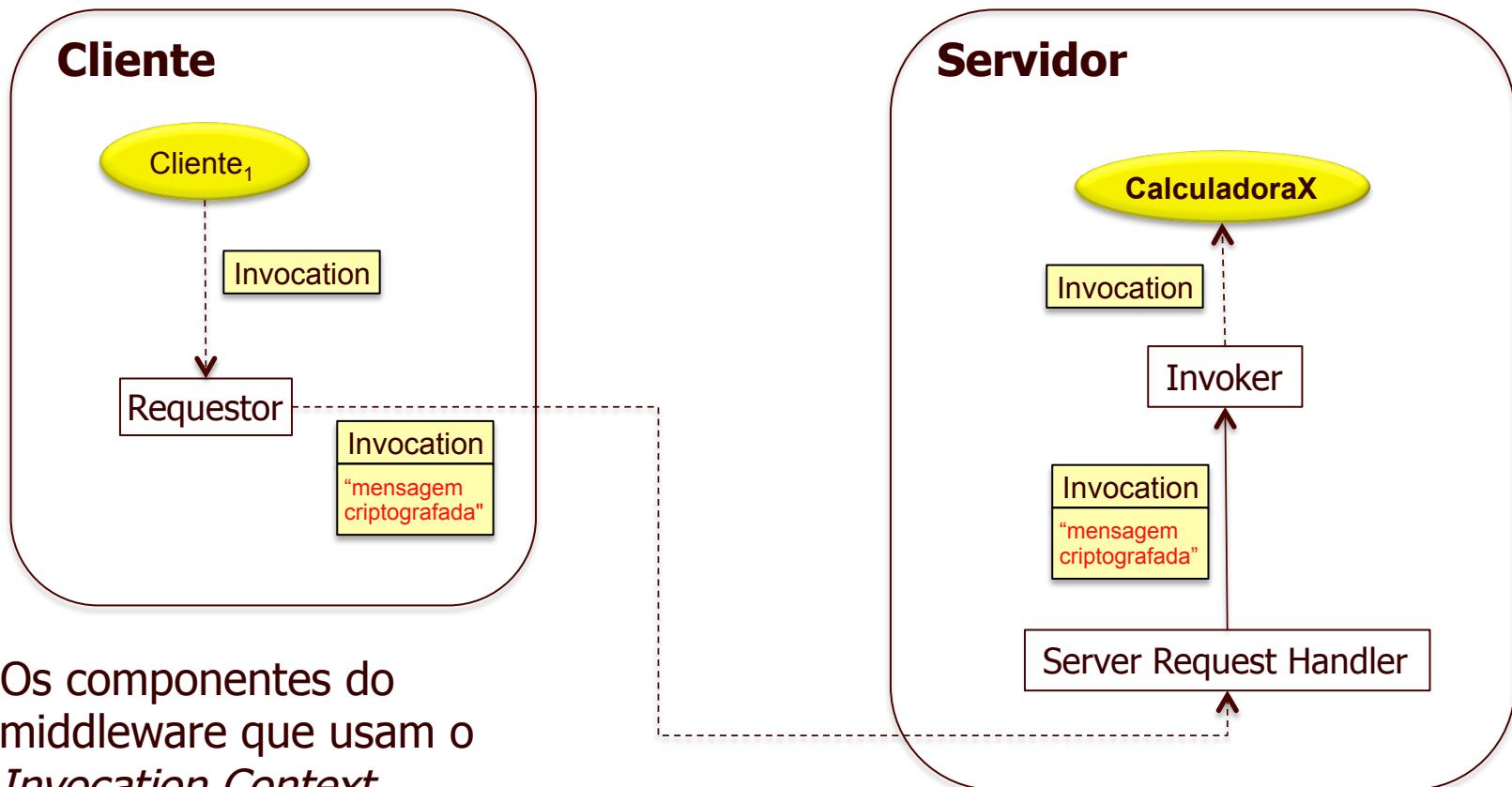


# Padrões de Extensão:: *Invocation Context*



- ✓ Os componentes do middleware que usam o *Invocation Context* precisam **concordar** no formato do **contexto**.

# Padrões de Extensão:: *Invocation Context*



- ✓ Os componentes do middleware que usam o *Invocation Context* precisam **concordar** no formato do **contexto**.

# Padrões de Extensão:: *Protocol Plug-In*

## ■ Contexto

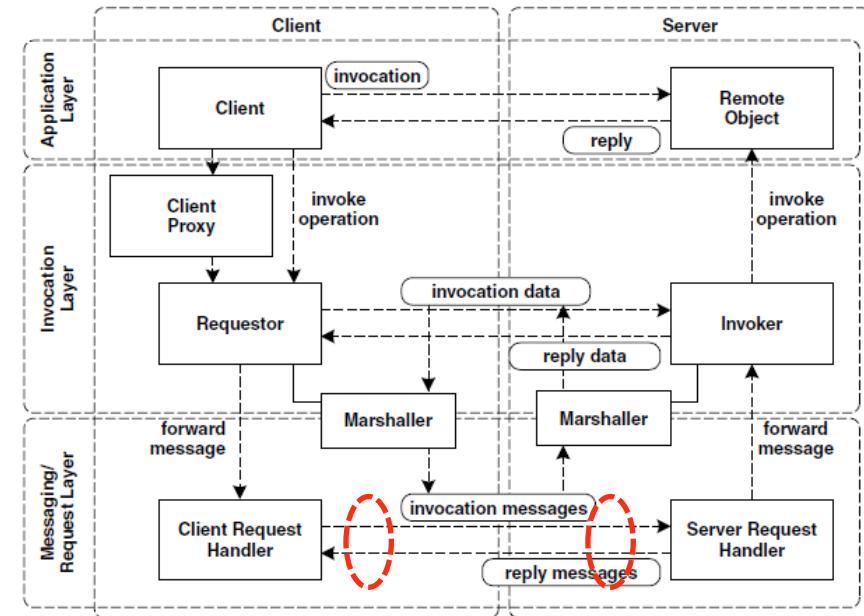
- Os componentes *Client/Server Request Handlers* usam o mesmo protocolo de comunicação.

## ■ Problema

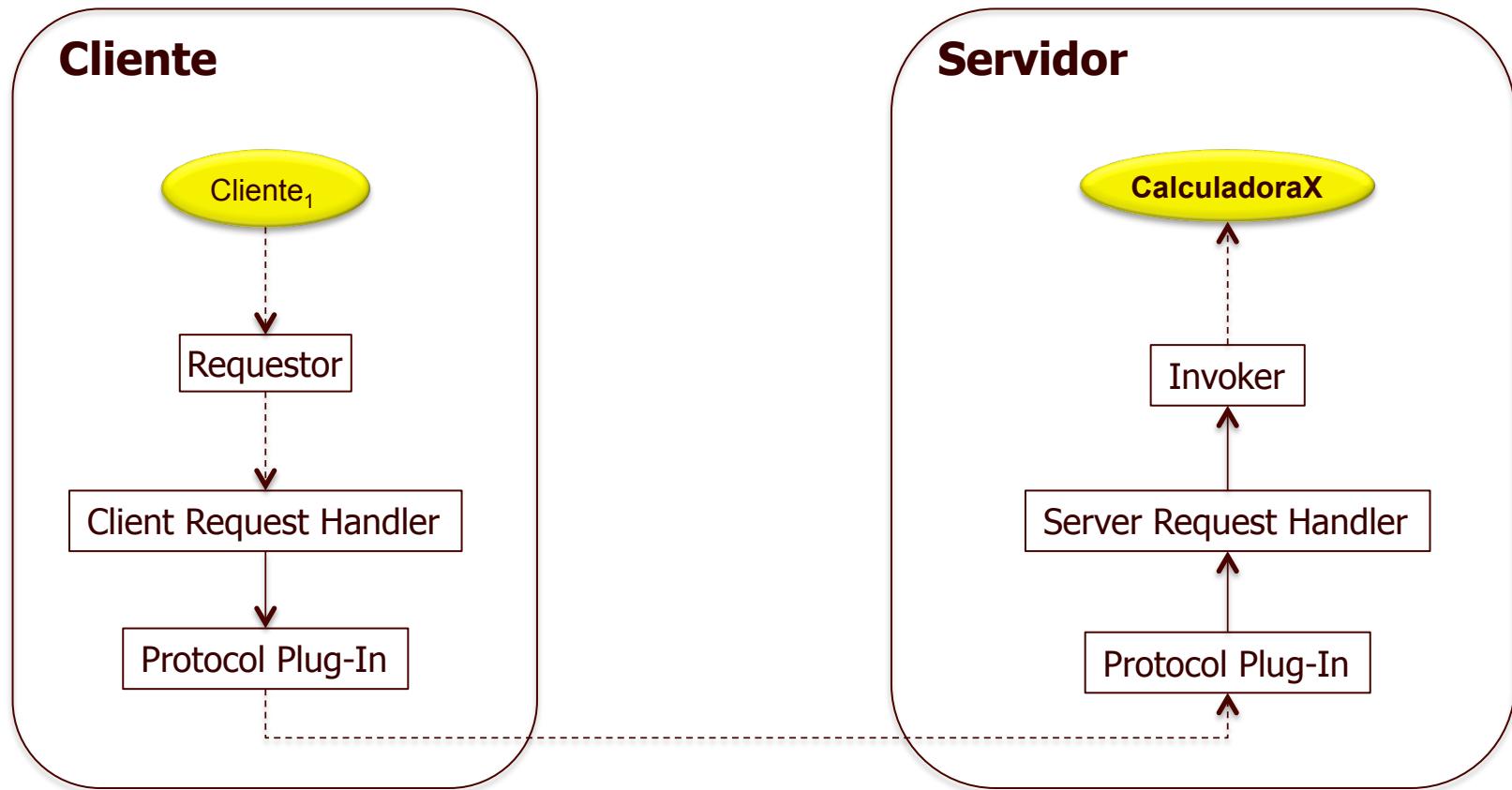
- Como resolver a (eventual) heterogeneidade de protocolos de comunicação usados pelo *Client Request Handler* e *Server Request handler*?

## ■ Solução

- Usar o *Protocol Plug-in* para estender os *Client/Server-Request Handlers*.
- O *Protocol Plug-in* deve fornecer uma interface que permita sua configuração.



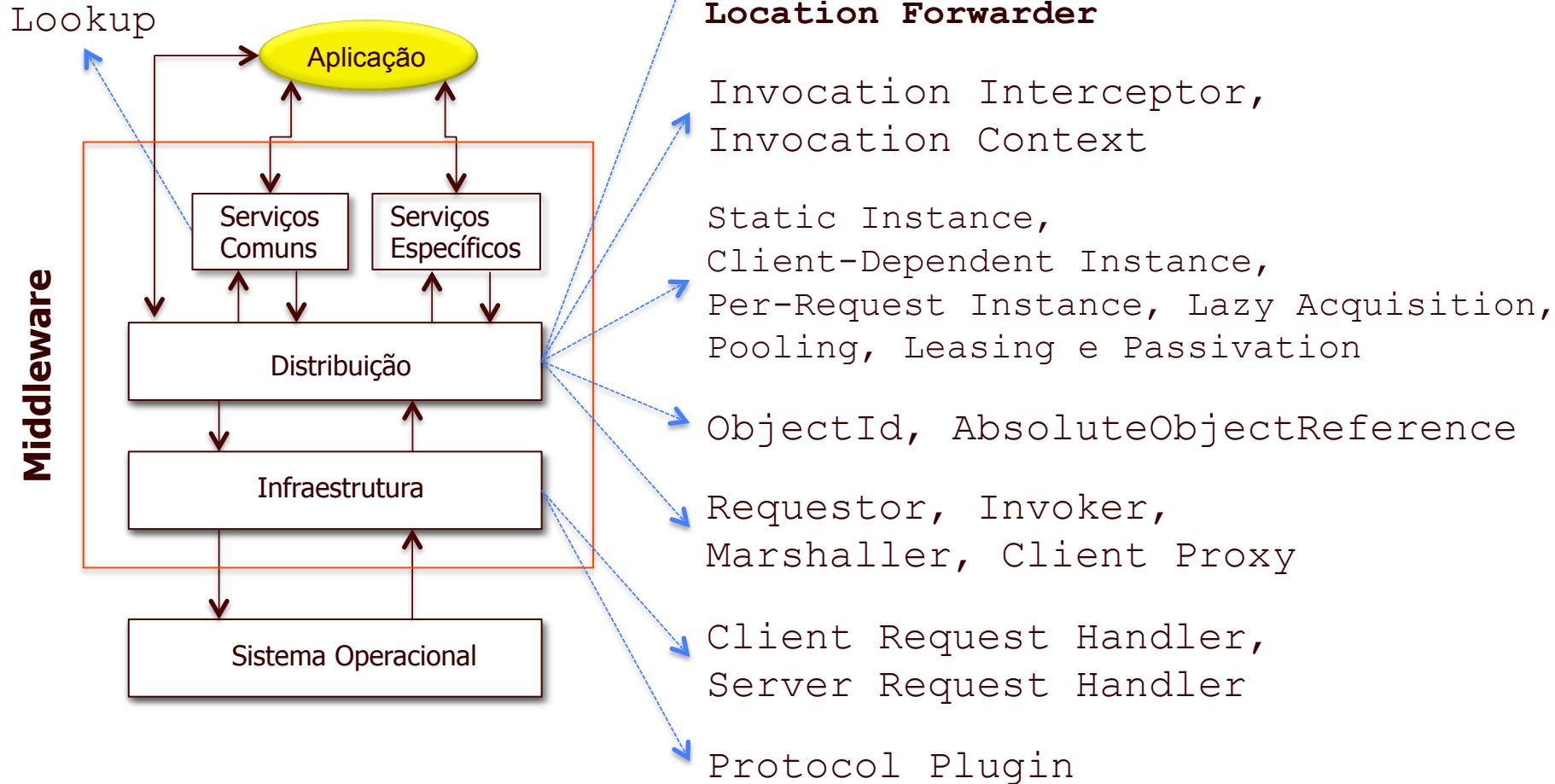
# Padrões de Extensão:: *Protocol Plug-In*



✓ O *Protocol Plug-in* pode ser definido através de *Invocation Interceptors*.

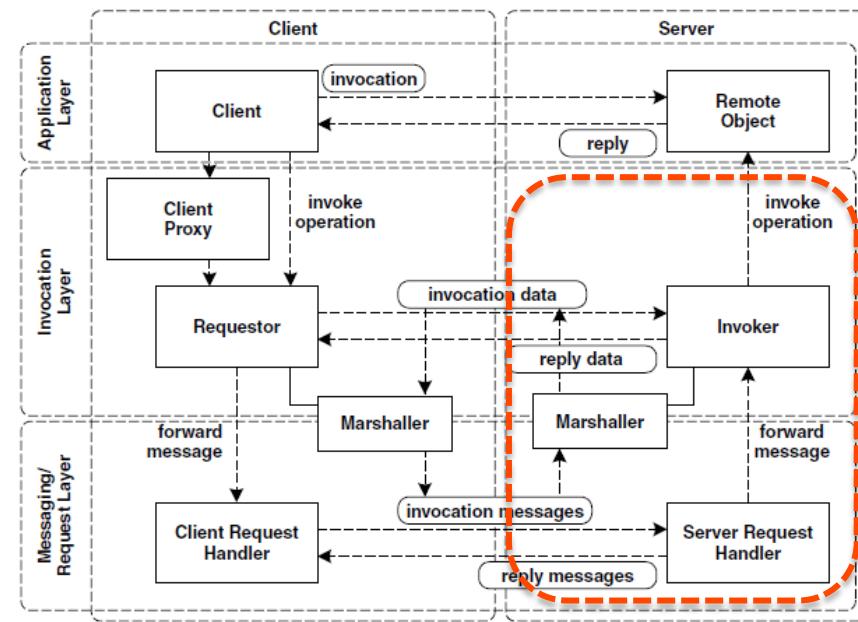
# Padrões de Infraestrutura Estendida

# Remoting Patterns:: Infraestrutura Estendida



# Padrões de Infraestrutura Estendida

- Implementam características avançadas de um middleware
- 05 padrões
  - Lifecycle Manager
  - Configuration Group
  - Local Object
  - QoS Observer
  - Location Forwarder



# Padrões de Infraestrutura Estendida:: *Lifecycle Manager*

## ■ Contexto

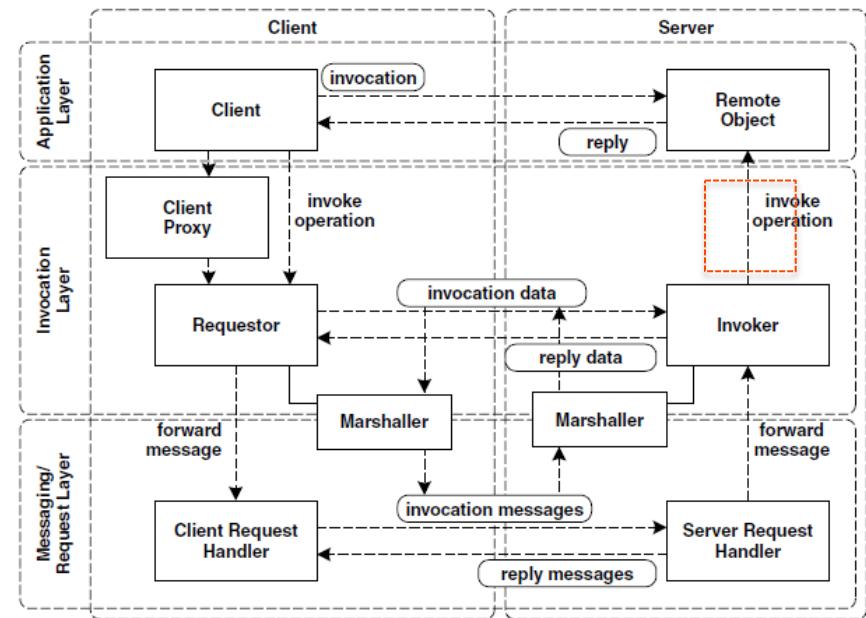
- O servidor precisa gerenciar diferentes tipos de ciclos de vida dos objetos remotos

## ■ Problema

- Como os desenvolvedores de aplicações distribuídas podem **customizar** as estratégias de ciclo de vida dos objetos remotos?

## ■ Solução

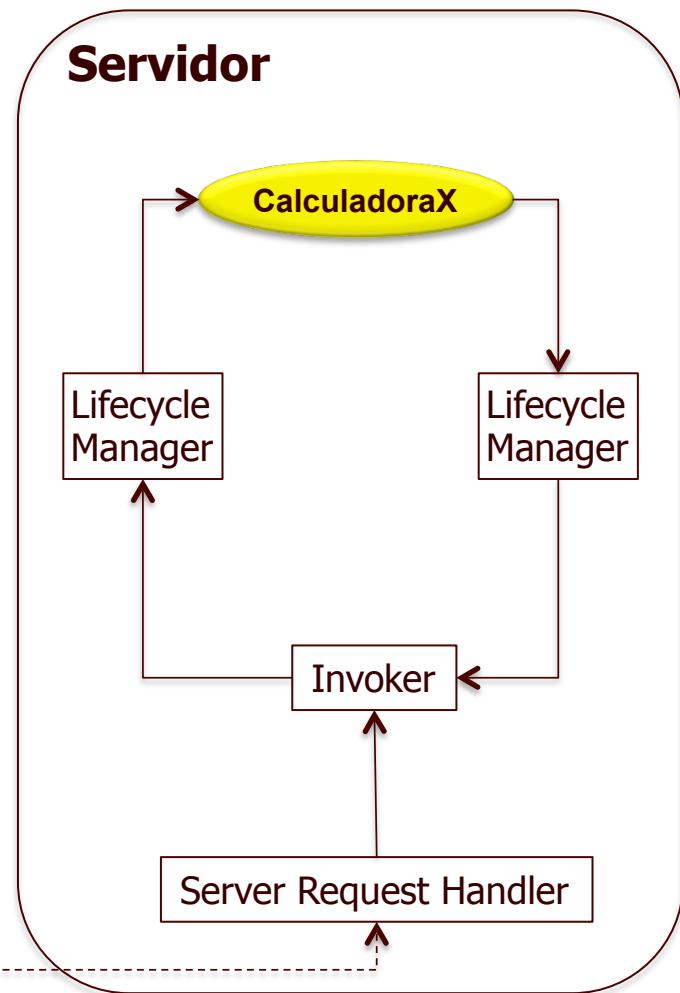
- Usar o *Lifecycle Manager* para gerenciar o ciclo de vida dos objetos remotos e *servants*.
- *Servants* que precisam estar cientes dos eventos de gerenciamento devem fornecer operações de “callback”



# Padrões de Infraestrutura Estendida:: *Lifecycle Manager*



- ✓ O *Lifecycle Manager* poder ser implementado como parte do *Invoker* ou como um *Invocation Interceptor*
- ✓ Como o *Lifecycle Manager* é invocado **antes e depois** da invocação, ele pode gerenciar a criação, inicialização e destruição dos *servants*
- ✓ Há um impacto no **desempenho** da aplicação, pois o *Lifecycle Manager* é invocado a cada invocação do objeto remoto.



# Padrões de Infraestrutura Estendida:: Configuration Group

## Contexto

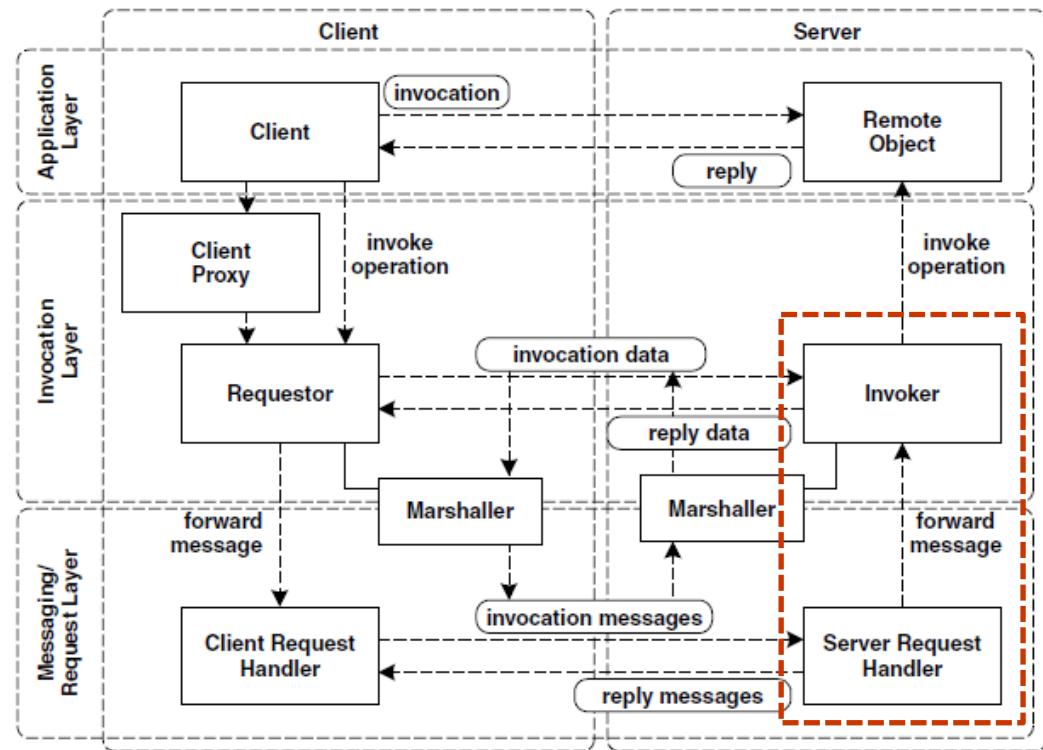
- Objetos remotos no servidor que devem ser configurados de forma similar, e.g., objetos gerenciados pelo mesmo *Configuration Manager*, objetos que usam o mesmo *Protocol Plug-in*.

## Problema

- Como configurar os objetos remotos de “forma prática”, e.g., evitando a configuração individual de cada objeto?

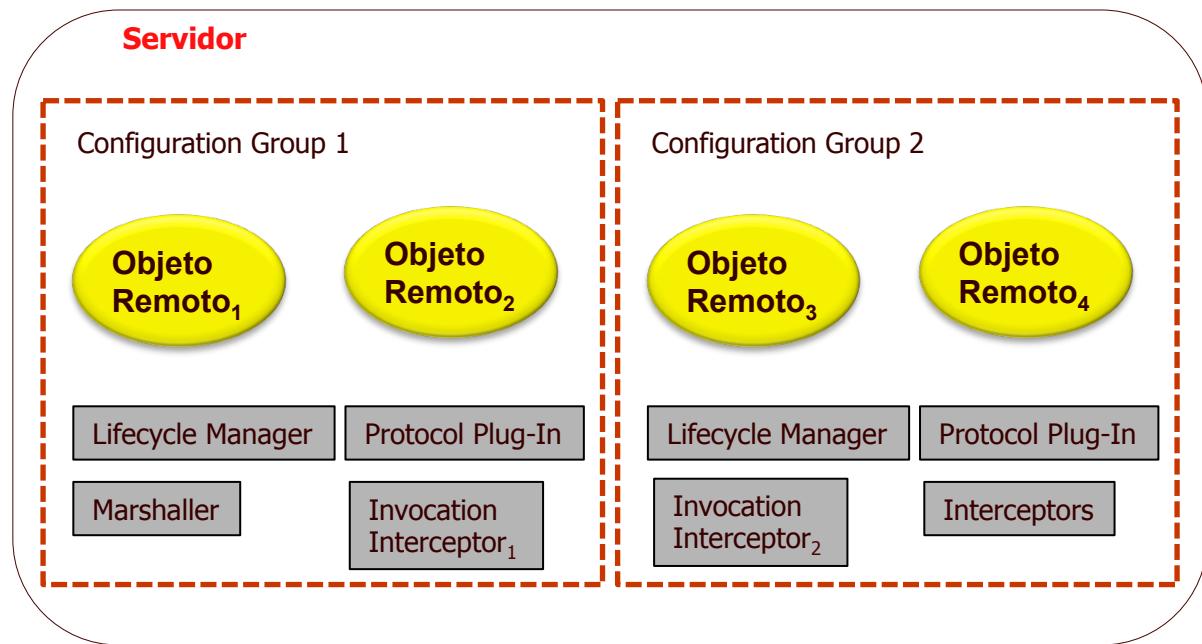
## Solução

- Agrupar objetos remotos que possuam configurações **similares** em *Configuration Groups*



# Padrões de Infraestrutura Estendida:: Configuration Group

- ✓ A implementação do *Configuration Group* é tipicamente associada ao **Server Request Handler** e **Invokers**.
- ✓ Quando os objetos possuem configurações muito distintas entre si, o uso do *Configuration Group* pode acarretar uma **complexidade** adicional que pode ser **desnecessária**.



# Padrões de Infraestrutura Estendida:: *Local Object*

## Contexto

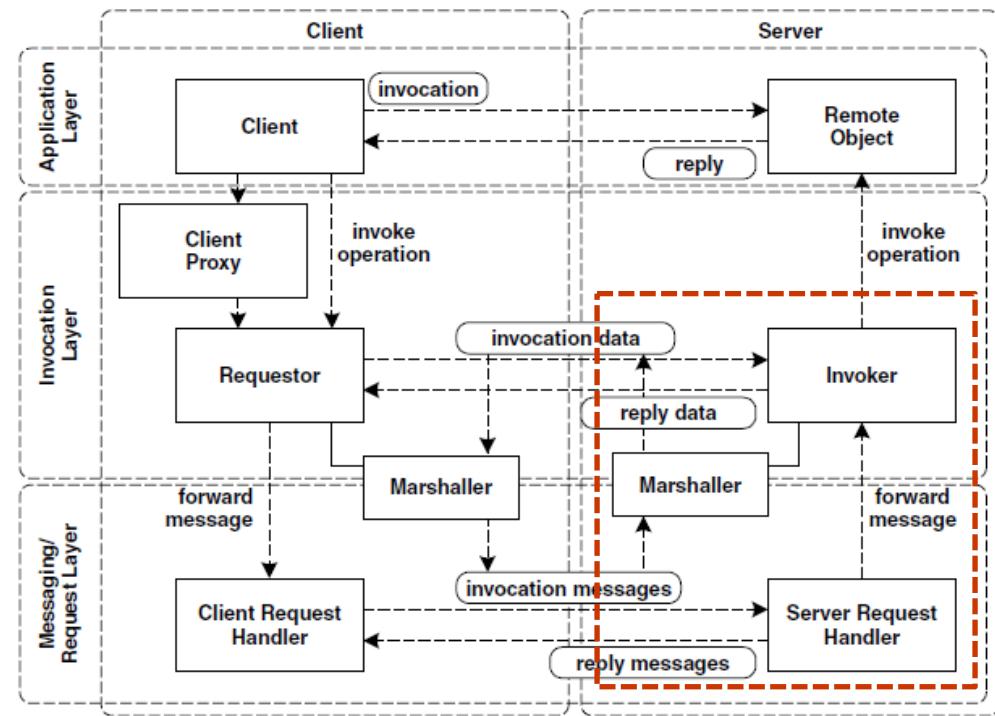
- O middleware precisa fornecer interfaces para configuração/acesso (e.g., *Lifecycle Manager*, *Protocol Plugin*, *Invocation Interceptors*) aos seus próprios componentes.

## Problema

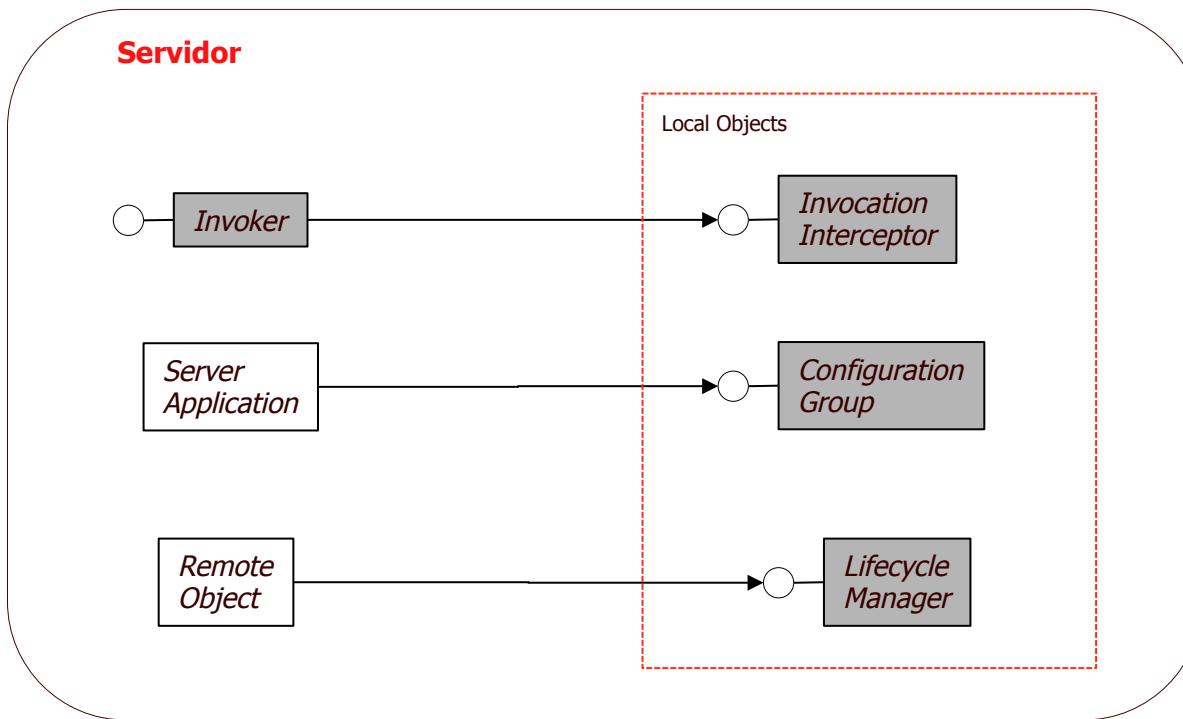
- Como fornecer interfaces de gerenciamento que não devem ser acessíveis remotamente, mas precisam usar o mesmo **modelo de programação** (e.g., semântica de invocação) dos objetos remotos?
- Considerando que: o acesso remoto pode causar **inconsistências** no componente e/ou **violação** de acesso.

## Solução

- Fornecer *Local Objects* aos desenvolvedores das aplicações no cliente/servidor que permitam o acesso à **configuração** dos **componentes** do middleware



# Padrões de Infraestrutura Estendida:: *Local Object*



- ✓ Para evitar o acesso remoto, precisa-se garantir que é **impossível** obter um *Absolute Object Reference* do *Local Object*.
- ✓ Estes objetos **não** podem ser acessados remotamente
- ✓ *Local Object* podem ser registrados no *Lookup*

# Padrões de Infraestrutura Estendida:: QoS Observer

## ■ Contexto

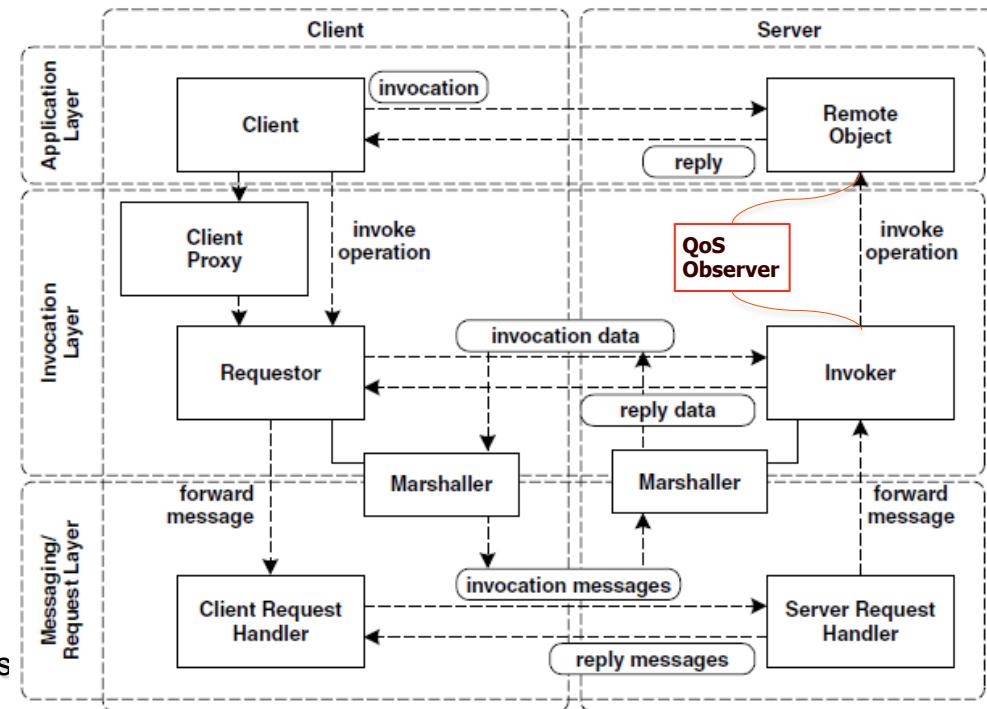
- O desenvolvedor da aplicação distribuída deseja gerenciar (monitorar/ controlar) a qualidade de serviço dos objetos remotos.
  - e.g., o **monitoramento** de um objeto remoto pode indicar que o tempo de resposta está aumentando e, para garantir a qualidade do serviço provido, novas invocações **passam a não ser aceitas**.

## ■ Problema

- **Como permitir que os desenvolvedores das aplicações distribuídas possam monitorar/ controlar a qualidade de serviço fornecida pelos objetos remotos?**

## ■ Solução

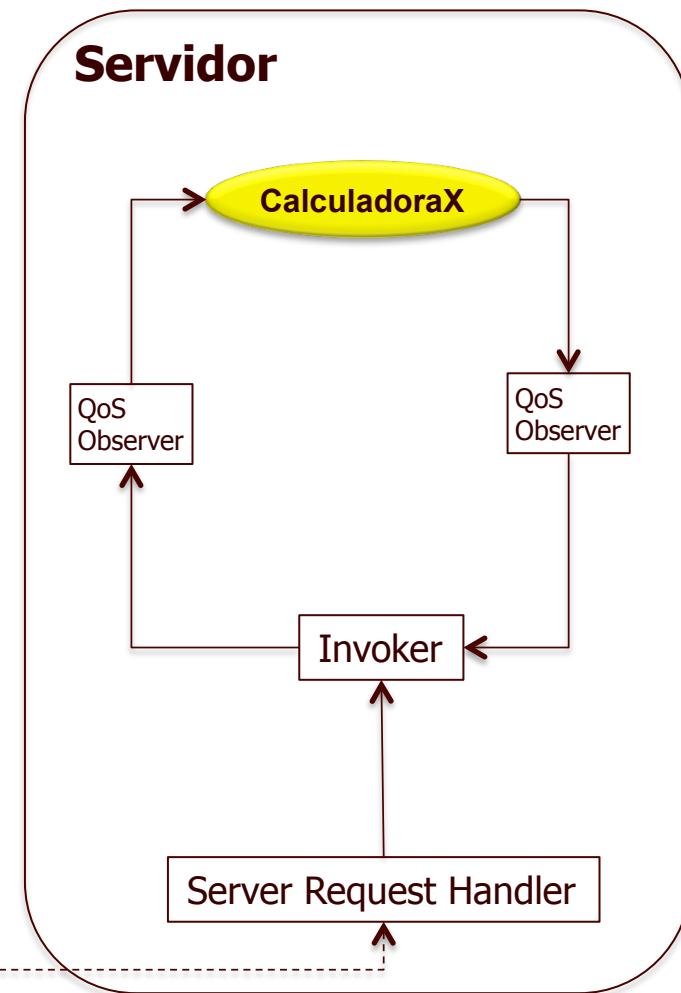
- Fornecer “**ganchos**” nos componentes do middleware que permitem ao desenvolvedores de aplicações distribuídas plugarem **QoS Observers**.
- **QoS Observers** são **específicos** da aplicação



# Padrões de Infraestrutura Estendida:: QoS Observer



- ▶ *QoS Observer* pode não ser parte do middleware.
- ▶ Quem pode ser monitorado: Objeto remoto, *Server Request Handler*, *Marshaller*, *Invoker*, *Requestor*, *Lifecycle Manager*, *Client Request Handler*, outros.
- ▶ O que pode ser monitorado: Inicio/fim da invocação, estabelecimento de conexões, início/fim do marshalling, nome da operação sendo invocada, outros.
- ▶ Os componentes do middleware precisam possuir uma interface de gerenciamento que permitam os *QoS Observers* obterem informações sobre eles.
- ▶ *QoS Observers* são tipicamente definidos como *Local Objects*.



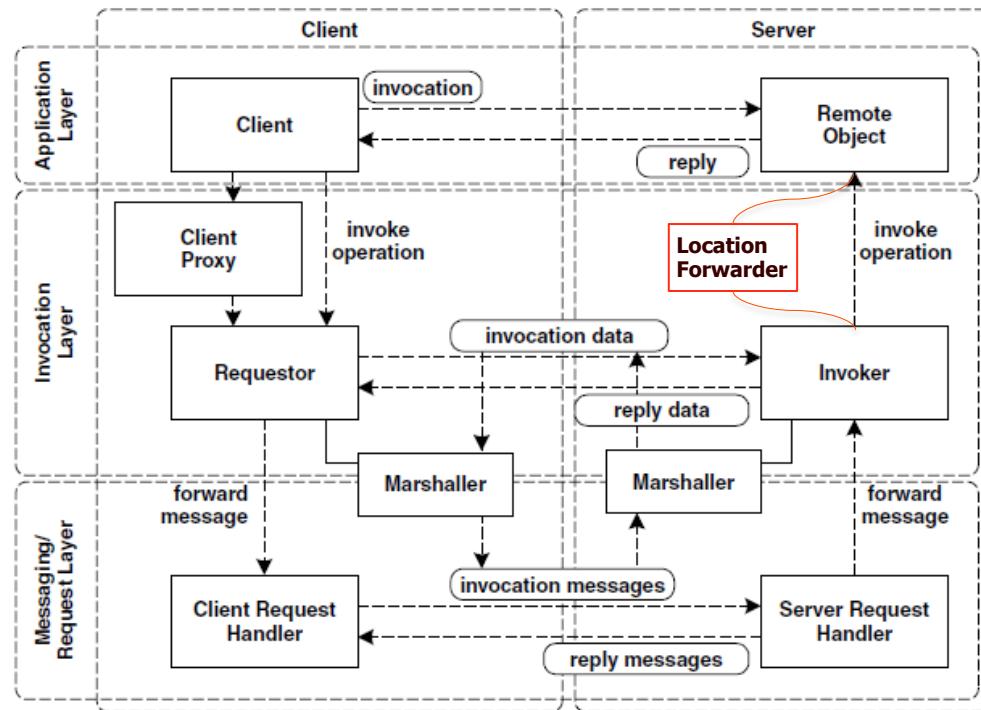
# Padrões de Infraestrutura Estendida:: *Location Forwarder*

## ■ Contexto

- Invocações que chegam ao *Invoker* e precisam ser encaminhadas ao Objeto Remoto

## ■ Problema

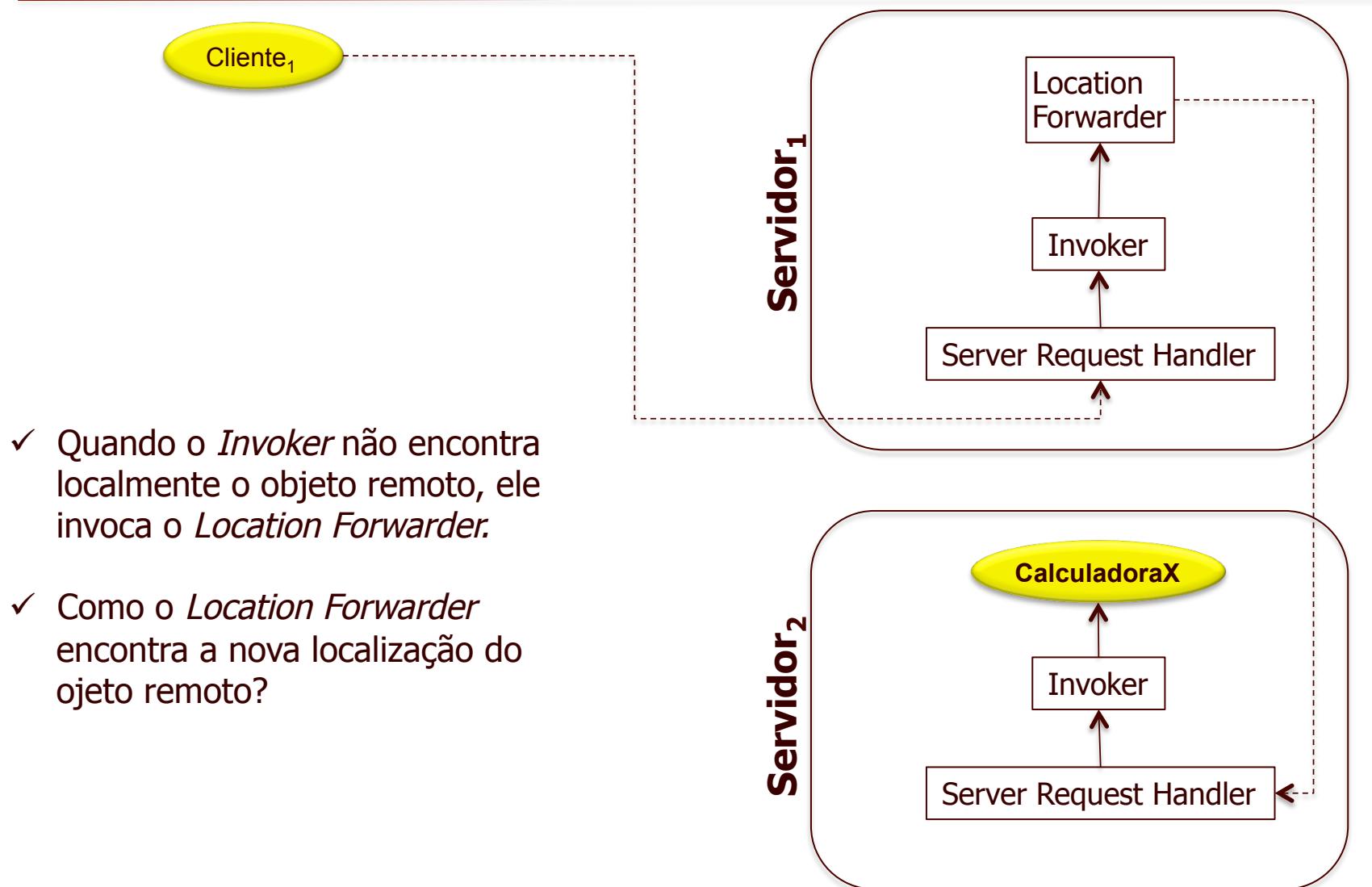
- Como encaminhar transparentemente requisições a objetos remotos que não mais se encontram no servidor?
- Considerando que (i) o objeto pode ter **migrado** porque a quantidade de recursos no servidor inicial não era mais suficiente, e (ii) mecanismos de tolerância a falhas e balanceamento de carga criam **réplicas** do objeto e estas réplicas precisam ser acessadas como se fossem um único objeto.



## ■ Solução

- Usar um *Location Forwarder* que encaminha a invocação ao objeto remoto em outro servidor.

# Padrões de Infraestrutura Estendida:: *Location Forwarder*



# Padrões de Infraestrutura Estendida:: *Location Forwarder*

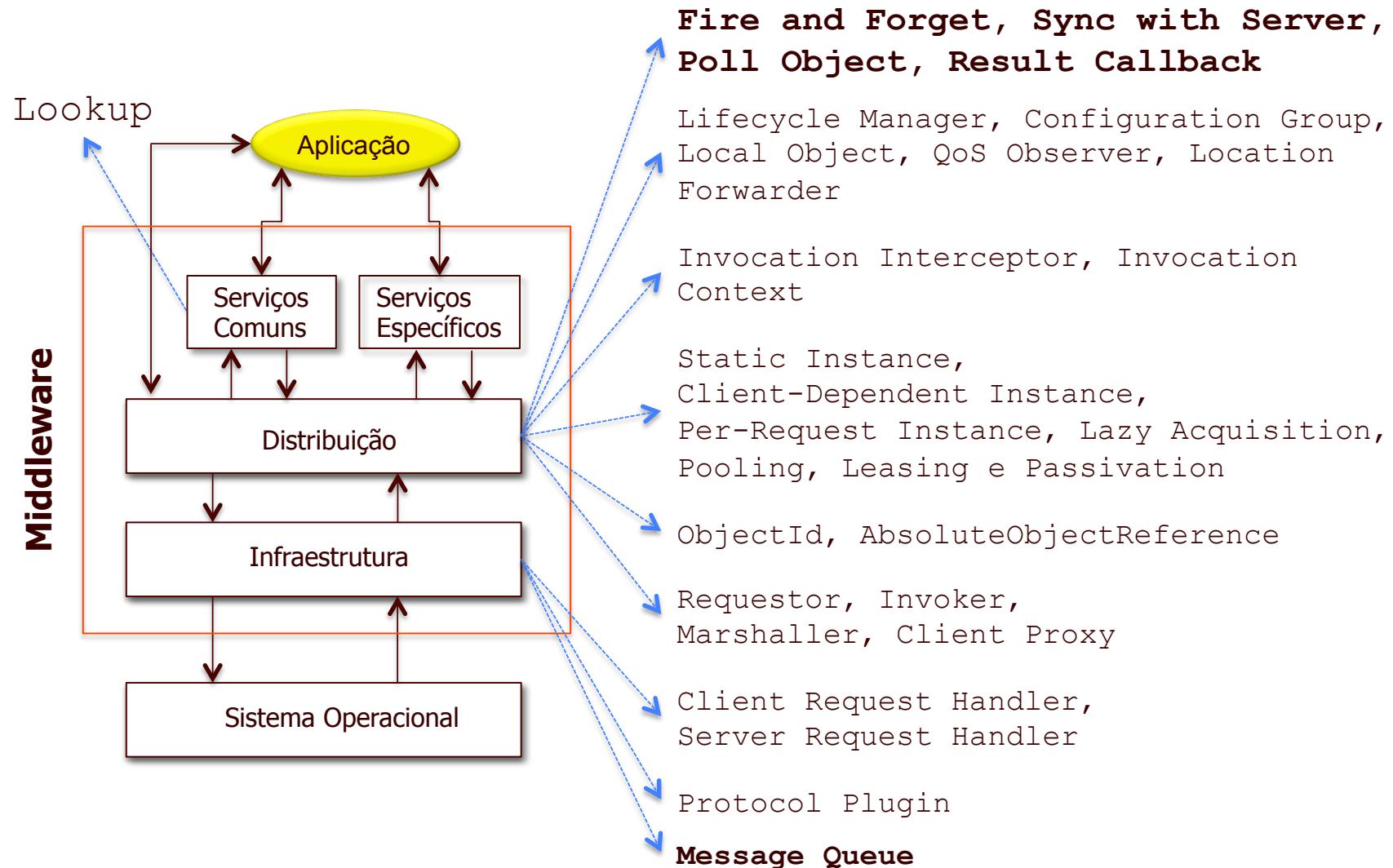
---

## ■ Variantes

- *Location Forwarder* informa ao **cliente** a nova localização do objeto remoto
- *Location Forwarder* pode funcionar como um **balanceador** de carga redirecionando as invocações usando algum algoritmo de平衡amento
- [em aplicações tolerantes a falhas] *Location Forwarder* pode **coordenar** a invocação a vários objetos remotos
- Quando um objeto remoto falha, o *Location Forwarder* pode passar ao cliente a localização de alguma réplica que esteja funcionando.

# Padrões de Invocação Assíncrona

# Remoting Patterns:: Padrões de Invocação Assíncrona



# **Padrões de Invocação Assíncrona**

---

- **Cliente não precisa esperar uma resposta do servidor para continuar a sua operação**
- **Usados tipicamente para melhorar o desempenho da aplicação distribuída**
- **Padrões comportamentais e estruturais**
  - Fire and Forget (comportamental)
  - Sync with Server (comportamental)
  - Poll Object (estrutural)
  - Result Callback (estrutural)
  - Message Queue (estrutural)

# Invocação Assíncrona:: *Fire and Forget*

## Contexto

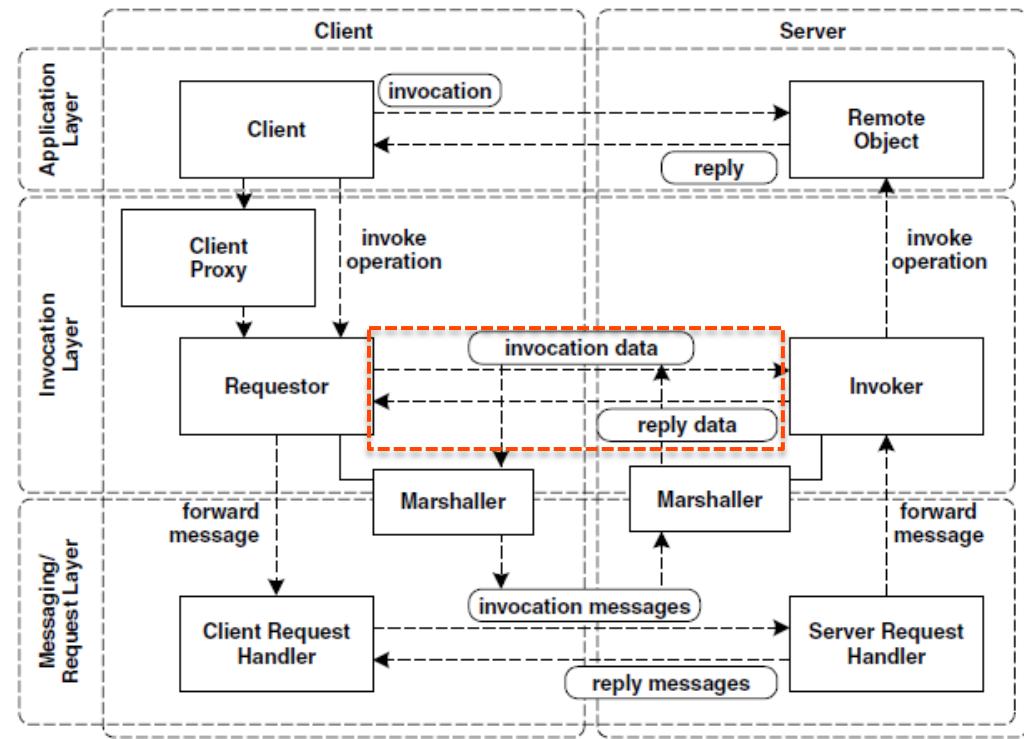
- Objetos remotos que possuem operações que não retornam valores/exceções

## Problema

- Como tratar invocações que não retornam valores/exceções?

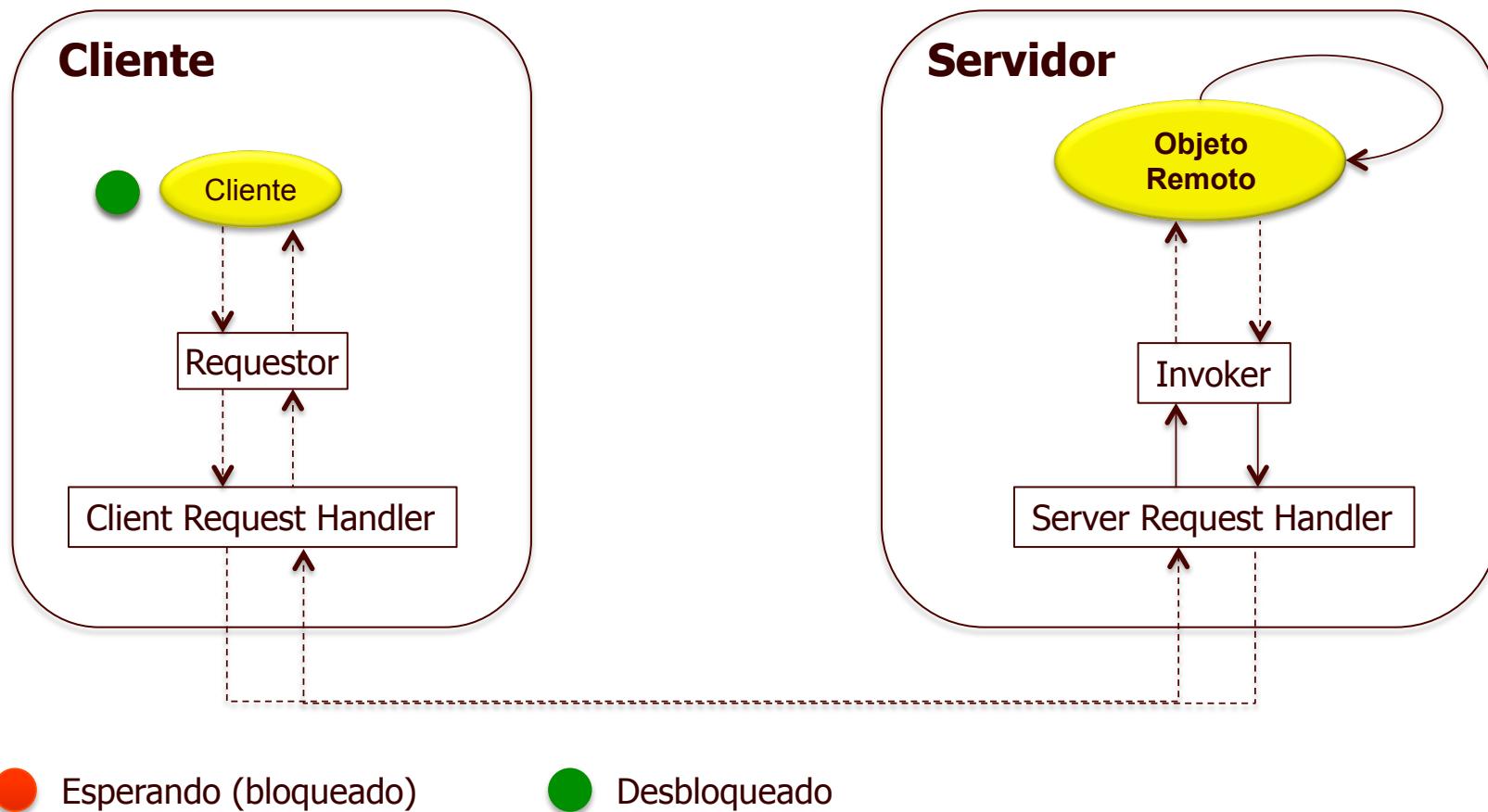
## Solução

- Implementar operações *Fire and Forget*
- *Requestor* envia a invocação e retorna o controle ao cliente imediatamente



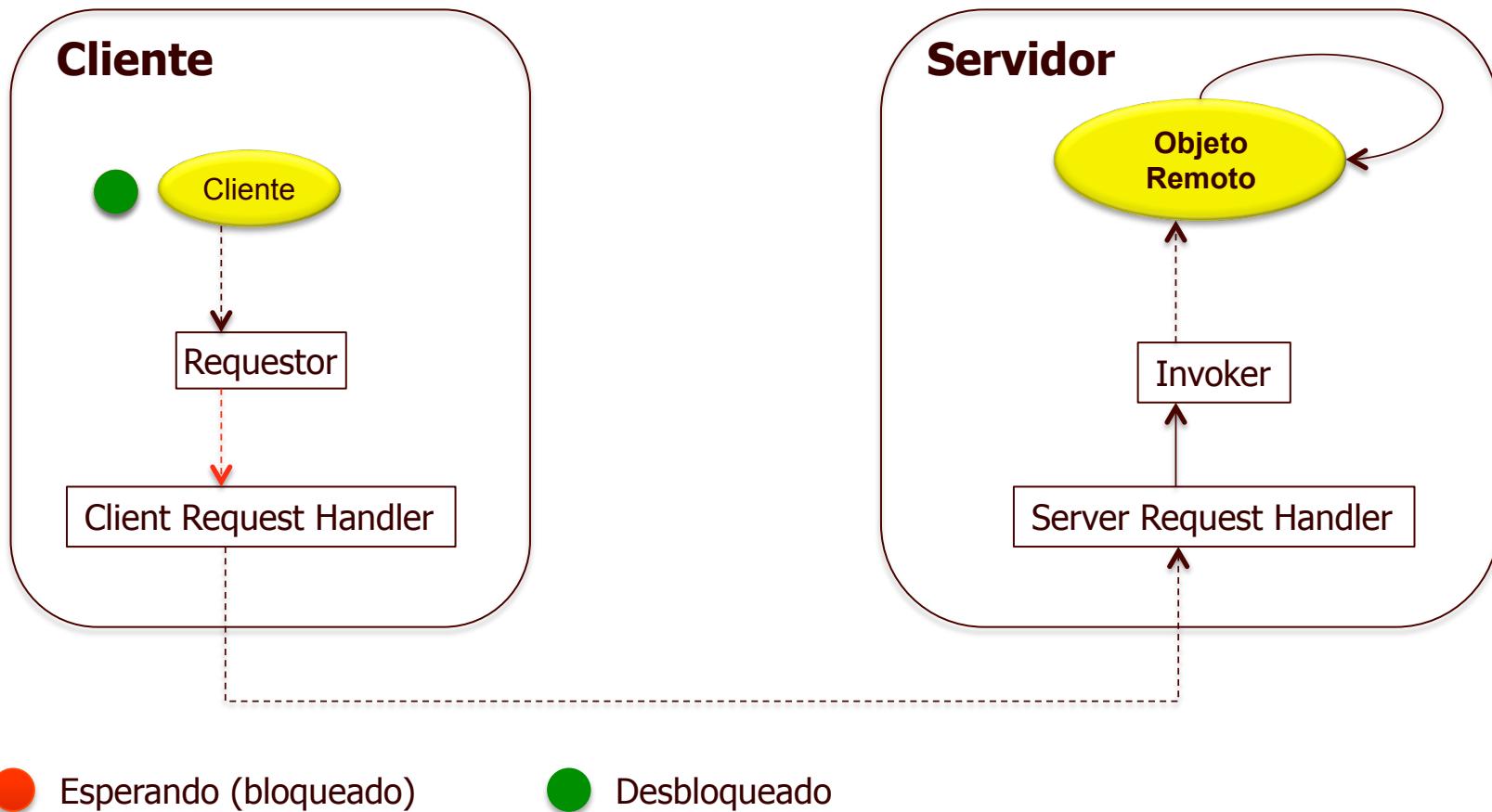
# Padrões de Extensão:: *Fire and Forget*

Execução Síncrona ("Tradicional")



# Padrões de Extensão:: *Fire and Forget*

## Execução Assíncrona (Fire and Forget)



# Invocação Assíncrona:: Sync with Server

## Contexto

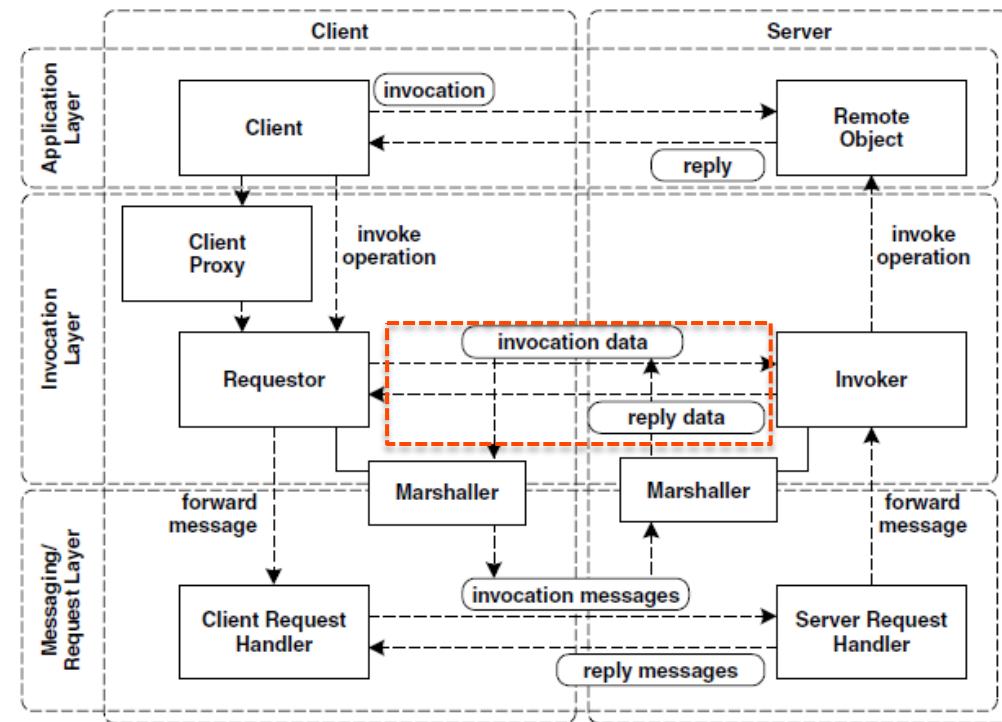
- Objetos remotos que possuem operações que não retornam valores/exceções

## Problema

- Como tratar invocações que não retornam valores/exceções, **mas cujos clientes precisam ter garantias de que a invocação chegará ao servidor?**

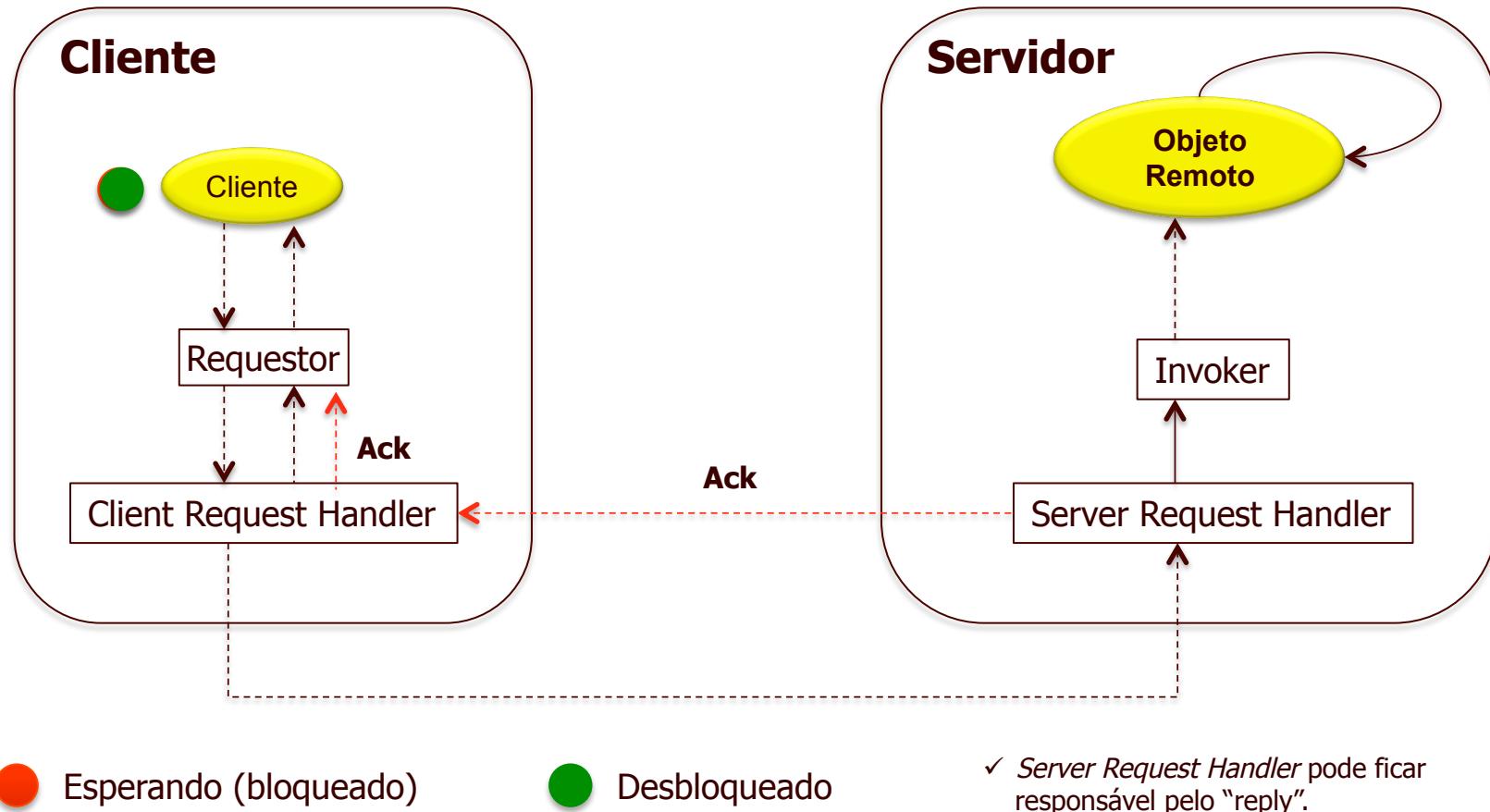
## Solução

- Implementar operações *Sync with Server*
- ***Requestor*** envia a invocação, **espera por uma confirmação do servidor** e retorna o controle ao cliente imediatamente (não espera pela execução da invocação)



# Padrões de Extensão:: Sync with Server

## Execução Assíncrona (Sync with Server)



# Invocação Assíncrona:: *Poll Object*

## Contexto

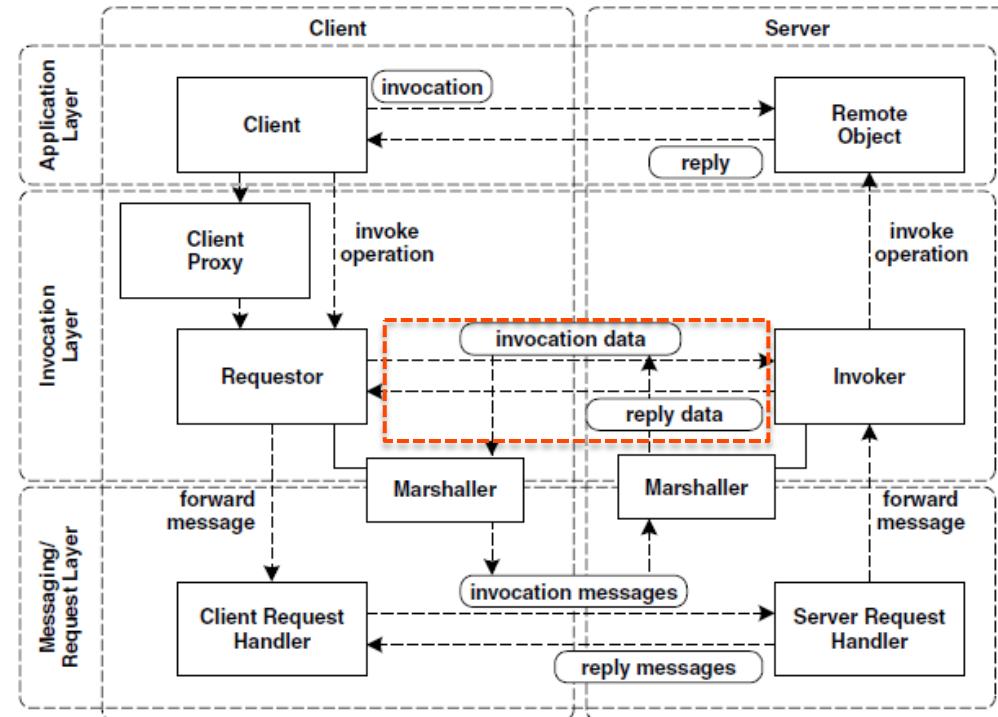
- Invocações aos objetos remotos podem executar **assincronamente**, mas o cliente depende do resultado (não de imediato) para realizar outras tarefas.

## Problema

- Como permitir que clientes invoquem operações remotas, sem precisar ficar esperando pelos resultados?

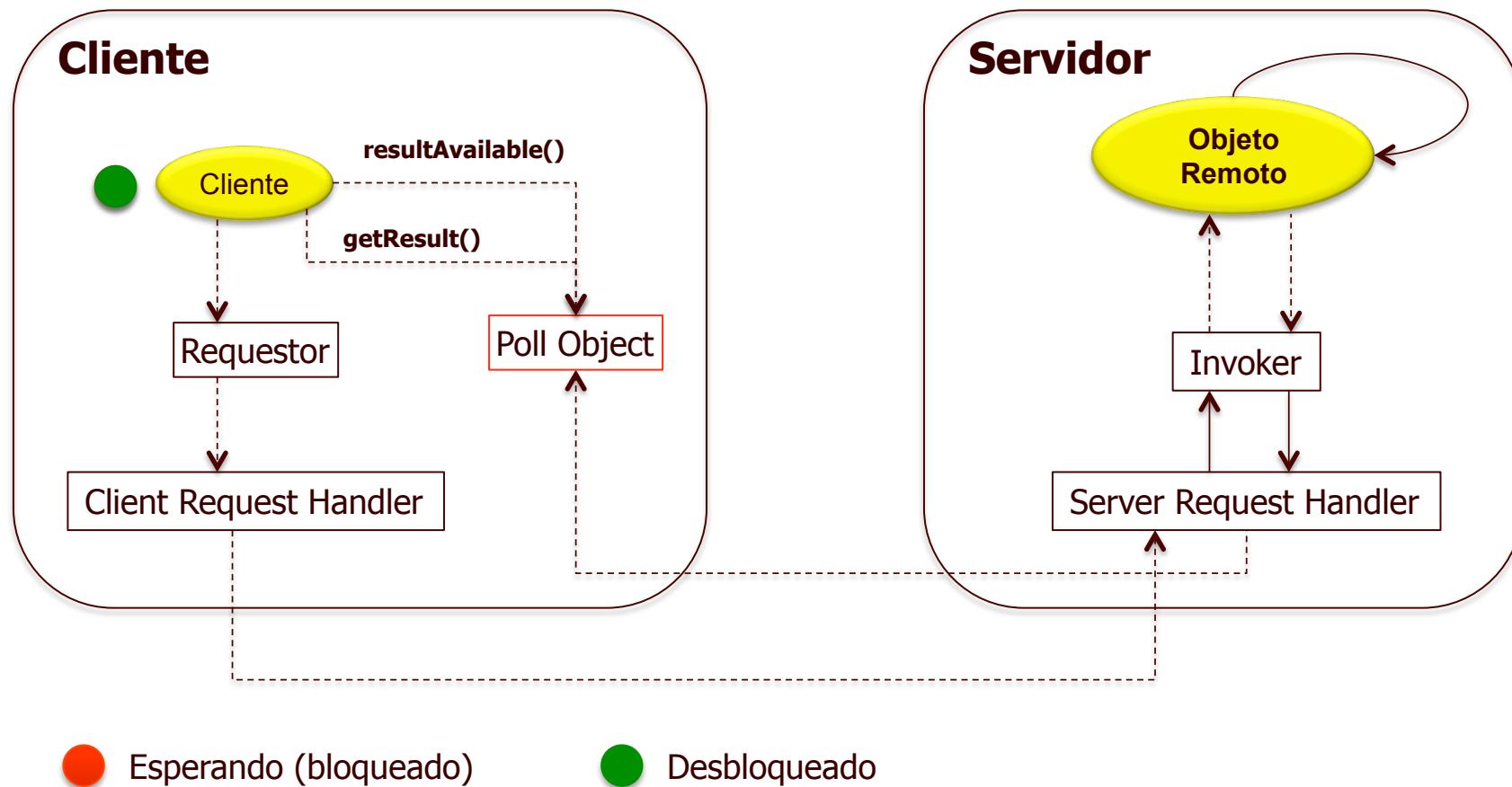
## Solução

- Implementar **Poll Objects** que se “responsabilizam” em receber as respostas das invocações remotas
- Cliente “de tempos em tempos” consulta o *Poll Object* para saber se a resposta já chegou.



# Padrões de Extensão:: *Poll Object*

## Execução Assíncrona (Poll Object)



# Invocação Assíncrona:: *Poll Object*

---

- **Poll Object depende da interface do objeto remoto**
  - Um tipo de Poll Object para cada operação remota
  - Um Poll Object para cada objeto remoto e um `getResult()` para cada operação remota
  - Um Poll Object genérico
- **Bom em situações em que o tempo de resposta é curto, mas longo o suficiente para permitir que o cliente realize outras operações.**
- **Em situações de longo período de espera, o Result Callback é mais adequado**

# Invocação Assíncrona:: *Result Callback*

## Contexto

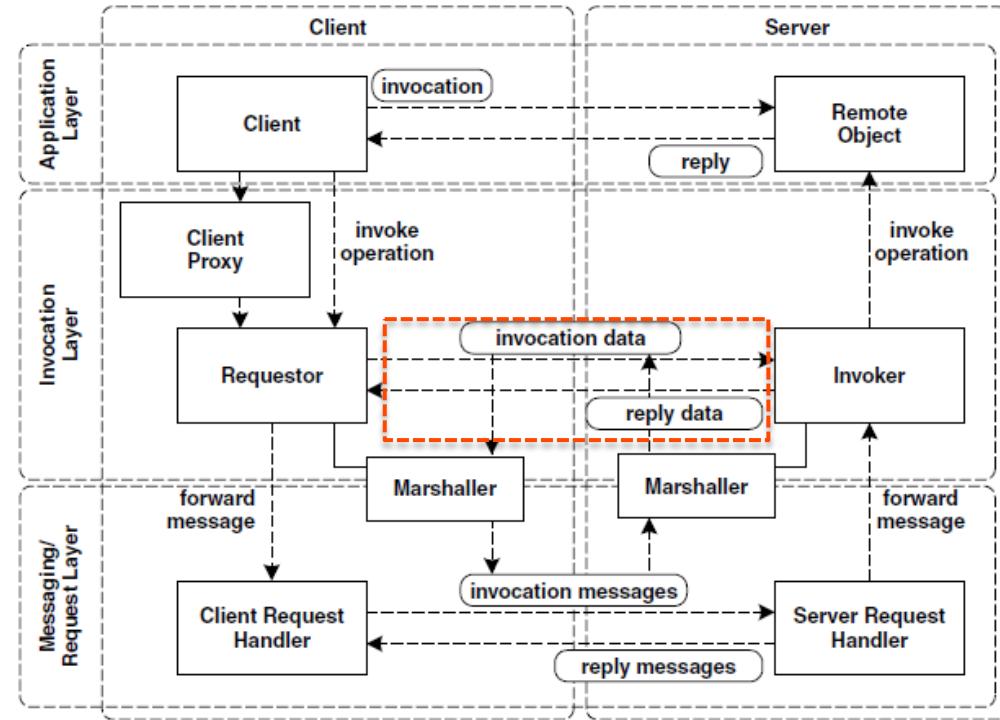
- Objetos remotos que retornam valores e cujo retorno pode ser tratado **assincronamente**.

## Problema

- Como permitir que clientes (i) invoquem operações remotas, (ii) sem precisar ficar esperando pelos resultados, (iii) mas quando estes resultados estiverem prontos o cliente precisa ser informado imediatamente?

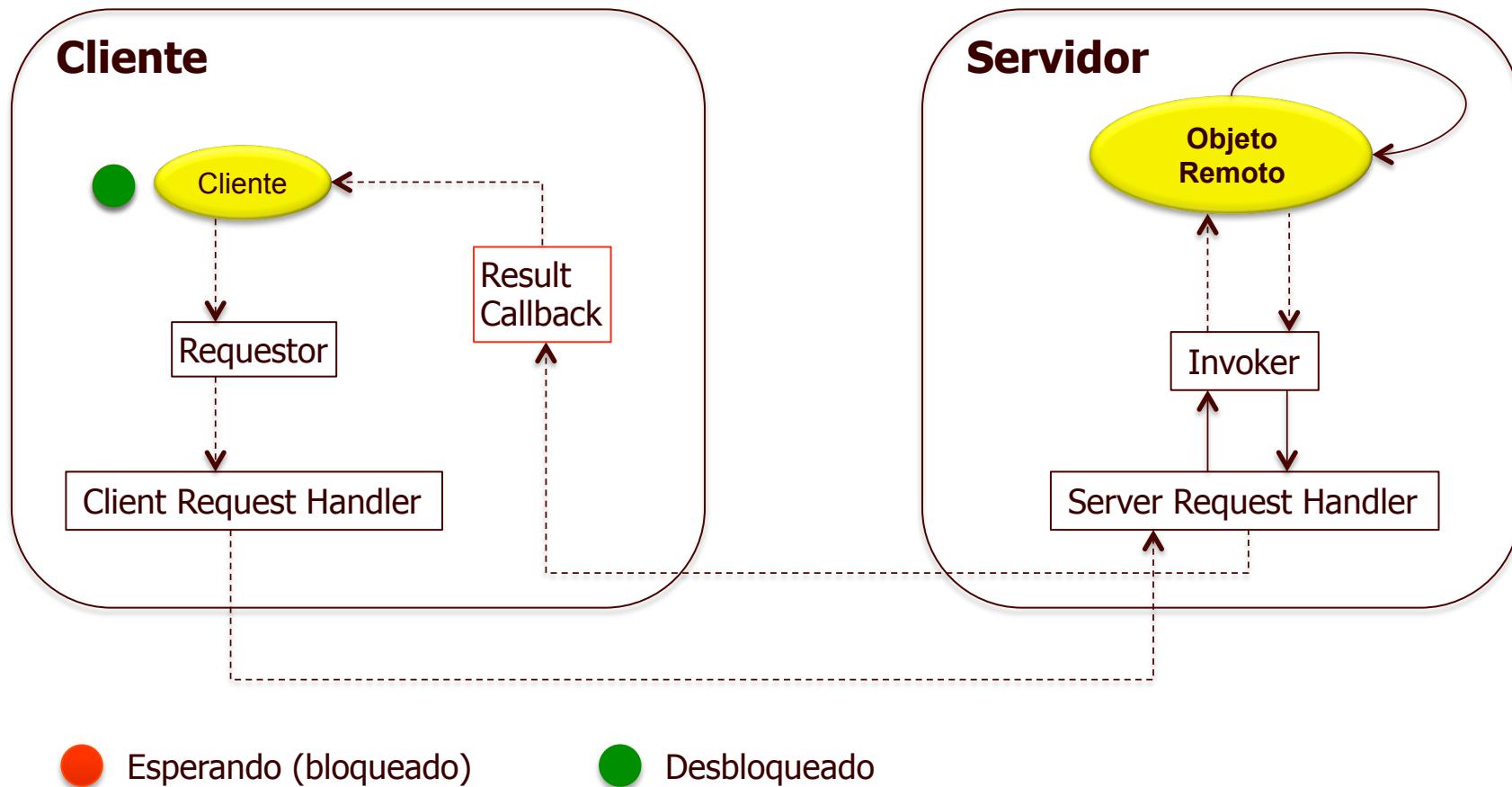
## Solução

- Fornecer uma interface *callback* para invocações remotas no cliente
- O controle é repassado imediatamente ao cliente tão logo a invocação tenha sido enviada ao servidor



# Invocação Assíncrona:: *Result Callback*

## Execução Assíncrona (Result Callback)



# Padrão de Projeto: Message Queue

## Contexto

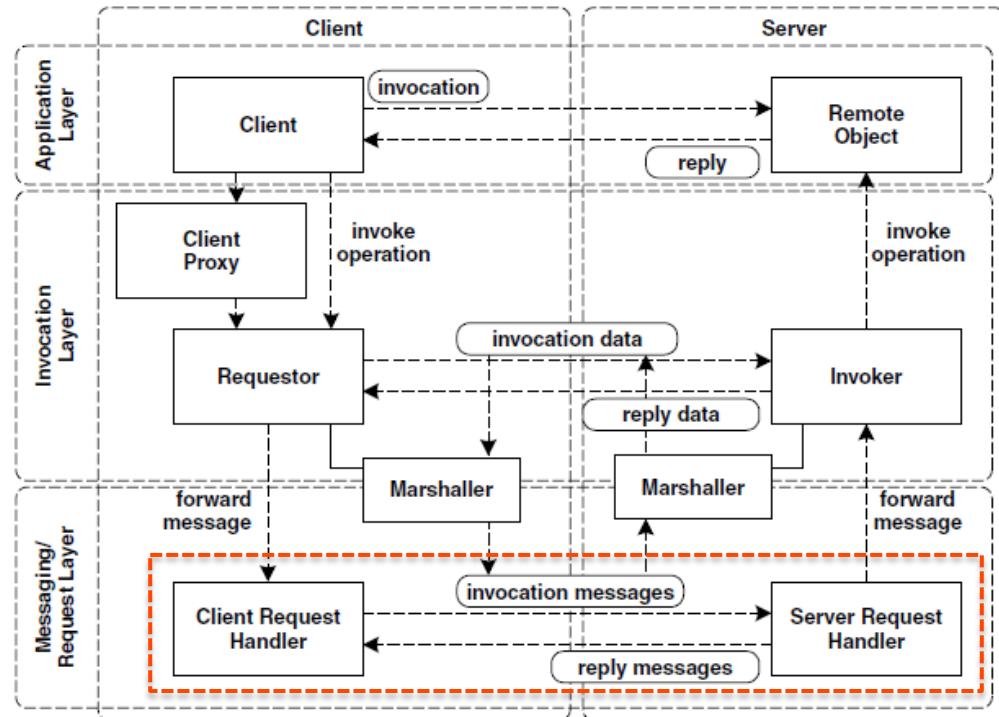
- O resultado da invocação pode ser tratado assincronamente

## Problema

- Poll Object e Result Callback precisam de um thread separado para receber os resultados assincronamente
- Nenhuma das variantes assíncronas pode tratar a ordem de invocação/recebimento das mensagens

## Solução

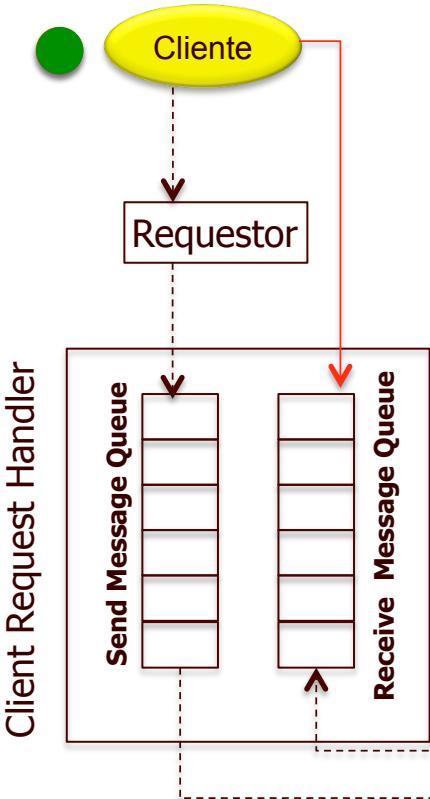
- Adicionar **filas** de mensagens aos **Request handlers**



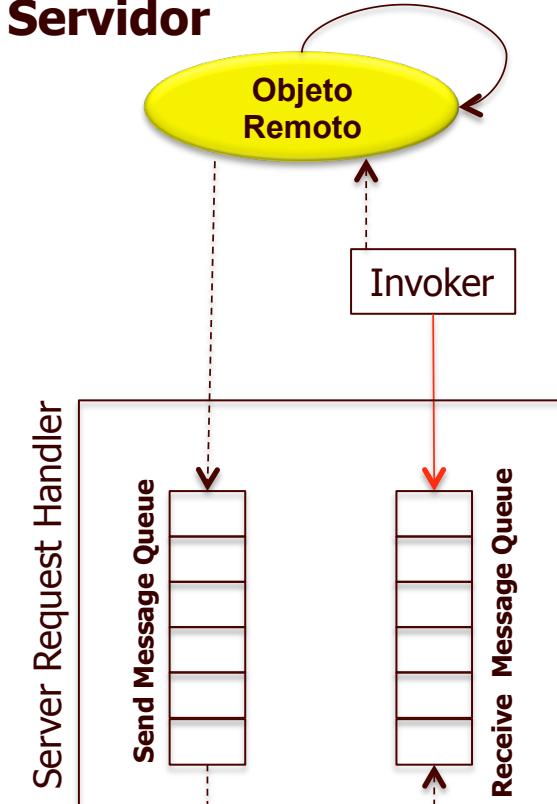
# Invocação Assíncrona:: Message Queue

## Execução Assíncrona (Message Queue)

### Cliente



### Servidor



---

# **Fim dos Slides**