
Instituto Superior de Engenharia de Lisboa

Área Departamental de Engenharia de Eletrónica e Telecomunicações e de Computadores

Mestrado em Engenharia Informática e de Computadores (MEIC)

Mestrado em Engenharia Informática e Multimédia (MEIM)

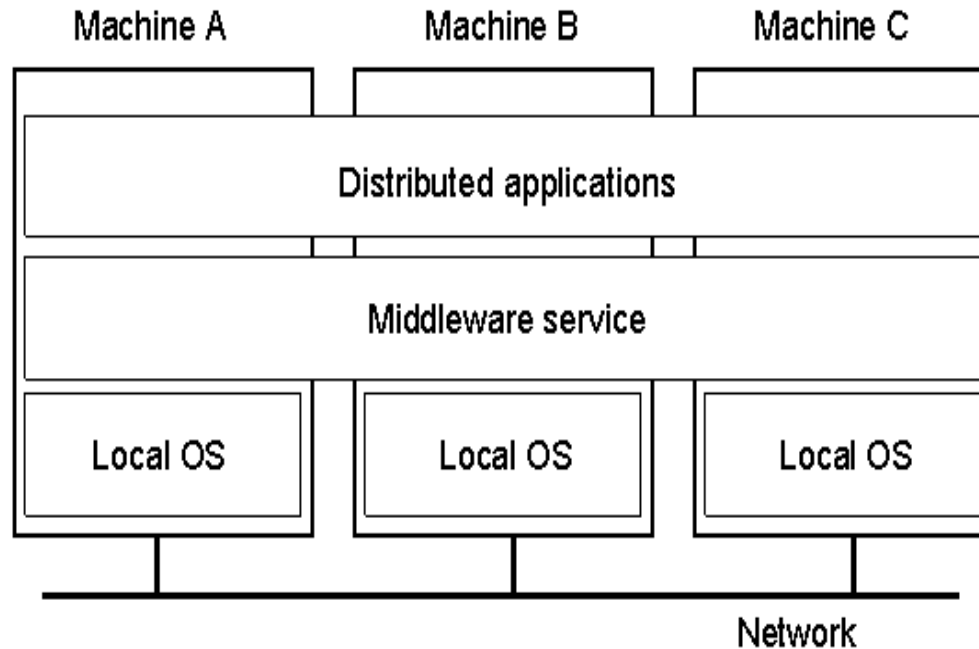
- **Desenvolvimento de aplicações distribuídas:**
 - ❖ **Paradigmas *Middleware***

Luís Assunção (lass@isel.ipl.pt ; luis.assuncao@isel.pt)

José Simão (jsimao@cc.isel.ipl.pt ; jose.simao@isel.pt)

As partes constituintes dos Sistemas Distribuídos

- O desafio de desenhar a arquitetura de um sistema distribuído consiste em:
 - Definir quais as partes constituintes que se relacionam, tendo em conta as funcionalidades (requisitos funcionais e não funcionais)
 - Especificar quais os padrões de interação, comunicação, e dependência entre as partes
- A granularidade das partes pode ser:
 - Processos de sistema operativo
 - Objetos remotos instanciados num processo de sistema operativo
 - Serviços que podem resultar da composição de múltiplos serviços
- As partes podem ter os seguintes papéis:
 - **Servidor** - Aceita pedidos, processa-os e devolve um resultado
 - **Cliente** - Realiza pedidos e obtém os resultados
 - **Peer** - Coopera com outros pares de forma simétrica por forma a executar uma tarefa
- As partes podem ser alojadas em **dispositivos IoT**, máquinas físicas, máquinas virtuais, **containers** no contexto de infraestruturas **Locais; Fog; Cloud ou Intercloud**



Conceito de *Middleware* numa infraestrutura local

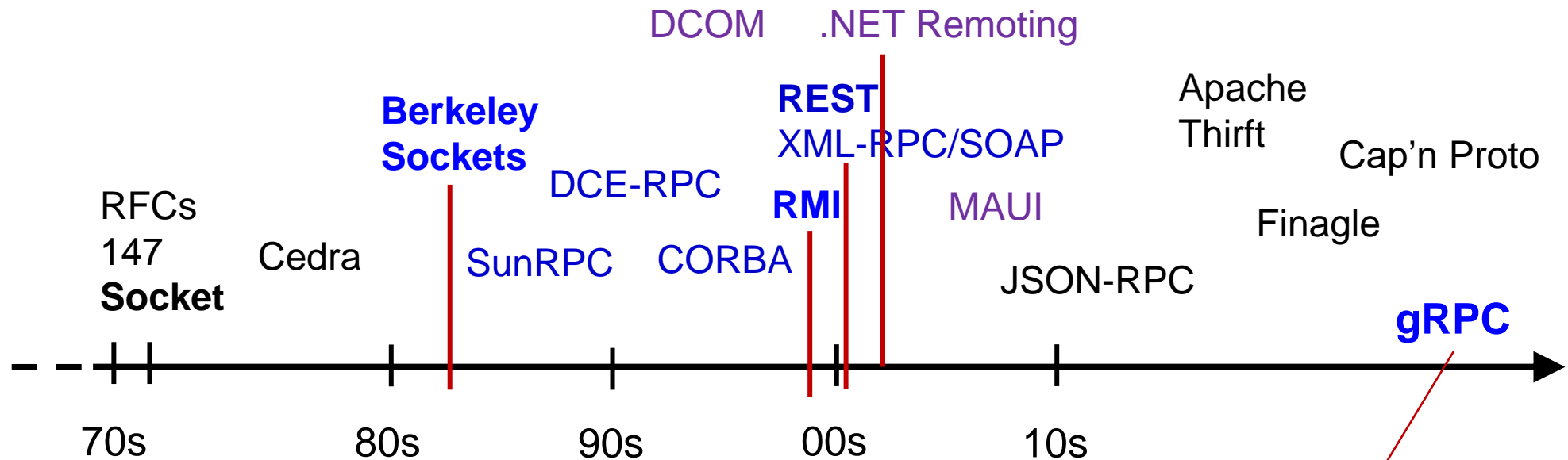
© From: *Distributed Systems, Principles and Paradigms* - Andrew Tanenbaum

- Disponibiliza um conjunto de *Application Programming Interfaces (APIs)* facilitando o desenvolvimento de aplicações de computação distribuída;
- Oferecem modelos de **Transparência** inerentes à distribuição de processamento, comunicação, armazenamento de dados, coordenação da interação entre partes, etc.

Acesso, Localização, Concorrência, Réplicas, Falhas, Migração, Desempenho, Escalabilidade

- **Acoplamento Forte (*Tight Coupling*)**: Quando as partes são altamente dependentes entre si, com dependências de implementação, ou as interações são baseadas em conexões *stateful*;
- **Acoplamento Fraco (*Lousely Coupling*)**: Quando as partes não têm dependências de implementação, dependendo só de especificação de contratos (interfaces) e idealmente as conexões devem ser *stateless*:
 - O contrato (interface) pode ser definido em linguagens específicas (ex. IDL, XML WSDL, *protocol buffer*, etc.) ou em artefactos das próprias linguagens de programação (ex: JAR em Java ou DLL em .NET)

Evolução dos *middleware* para interações remotas



DCE: Distributed Computing Environment

CORBA: Common Object Request Broker Architecture

RMI: Remote Method Invocation

MAUI: Mobile Assistance Using Infrastructure by Microsoft

Thrift: *Asynchronous* RPC by Facebook

Finagle: Fault-tolerant, protocol-agnostic RPC by Twitter

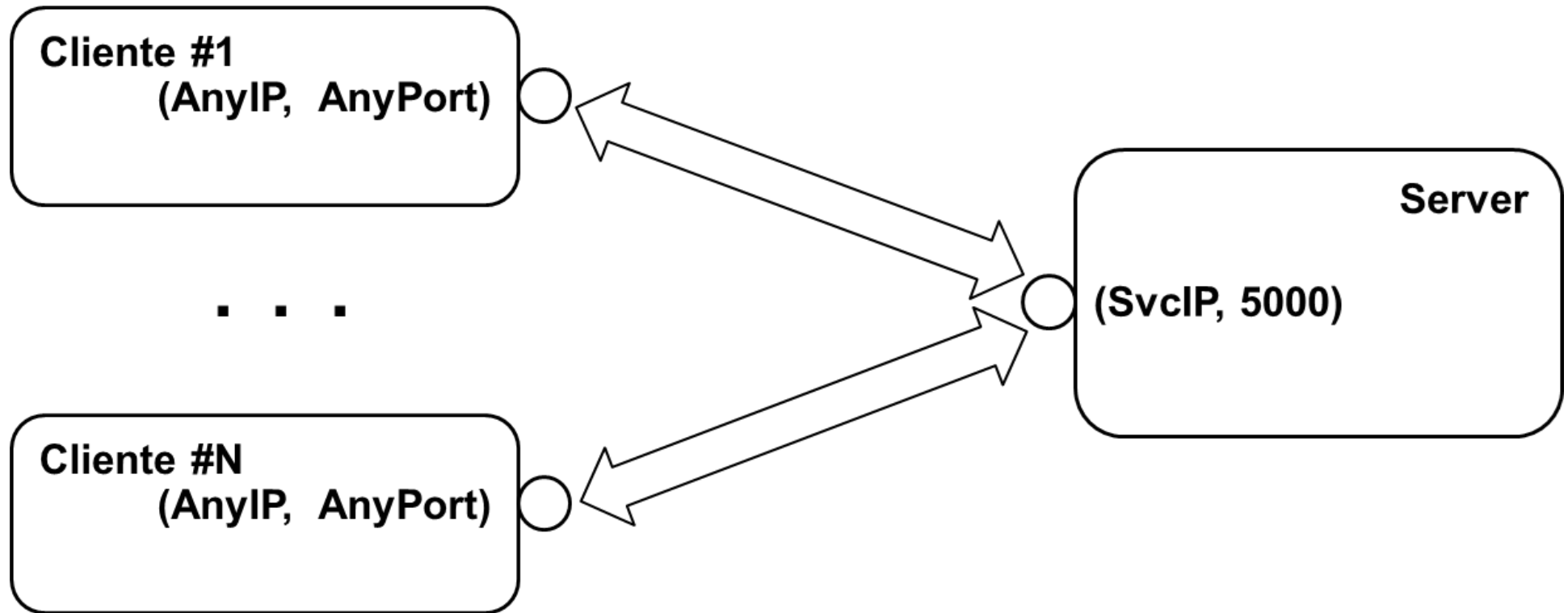
gRPC: Streaming RPC protocol by Google & Square

Cap'n Proto: Based on gRPC with improved performance

Utiliza **protocol buffers** como linguagem de definição de interfaces para descrever contratos entre os clientes e um serviço

- ❖ **Sockets TCP/IP** *[assume-se como pré-requisito]*
- ❖ **Objetos Distribuídos**
- ❖ **Remote Procedure Calls (RPC)**
- ❖ **Serviços (Web Services SOAP, REST)**
- ❖ **Micro Serviços**
- ❖ **Modelo Publish/Subscribe**

Client/Server com Sockets TCP/IP



- O TCP é um protocolo "*connection oriented*" que permite estabelecer conexões para comunicar dados entre dois pontos, em que um ponto tem o papel de servidor e o outro ponto tem o papel de cliente
- O servidor espera por conexões e o cliente estabelece a conexão para o servidor
- Para permitir a existência de múltiplas conexões entre os diferentes pontos são usadas associações (*IP Address, Port*) permitindo marcar os pacotes de dados (com o porto de origem e o porto destino) e determinar quais os programas que os enviam e os recebem
- Assim que se estabelece uma conexão entre cliente e servidor é possível comunicar dados, de forma fiável, como *Streams* em ambas as direções, até a conexão ser terminada
- A serialização dos objetos tem de ser explícita para formatos Xml, Json etc.

Sockets TCP/IP - Servidor concorrente

Client #1

connect

Client #N

connect

Accept

Session

request

reply

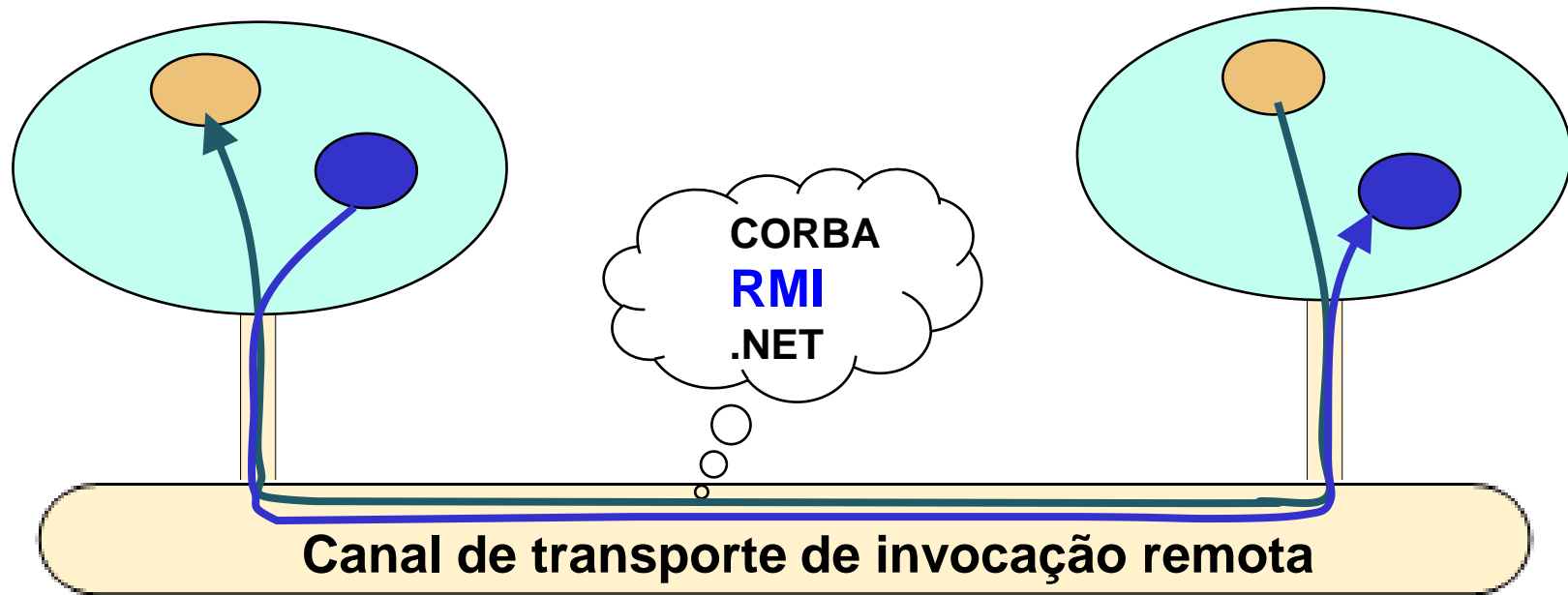
Session

request

reply

- O atendimento de pedidos concorrentes terá de ser implementada explicitamente, isto é, depende da responsabilidade do programador
- Devido à existência de *middlewares* com maiores abstrações, raramente é necessário desenvolver aplicações baseadas em *sockets*.
- No entanto, esses *middleware* usam *sockets* no mais baixo nível.

Interação com Objetos Distribuídos



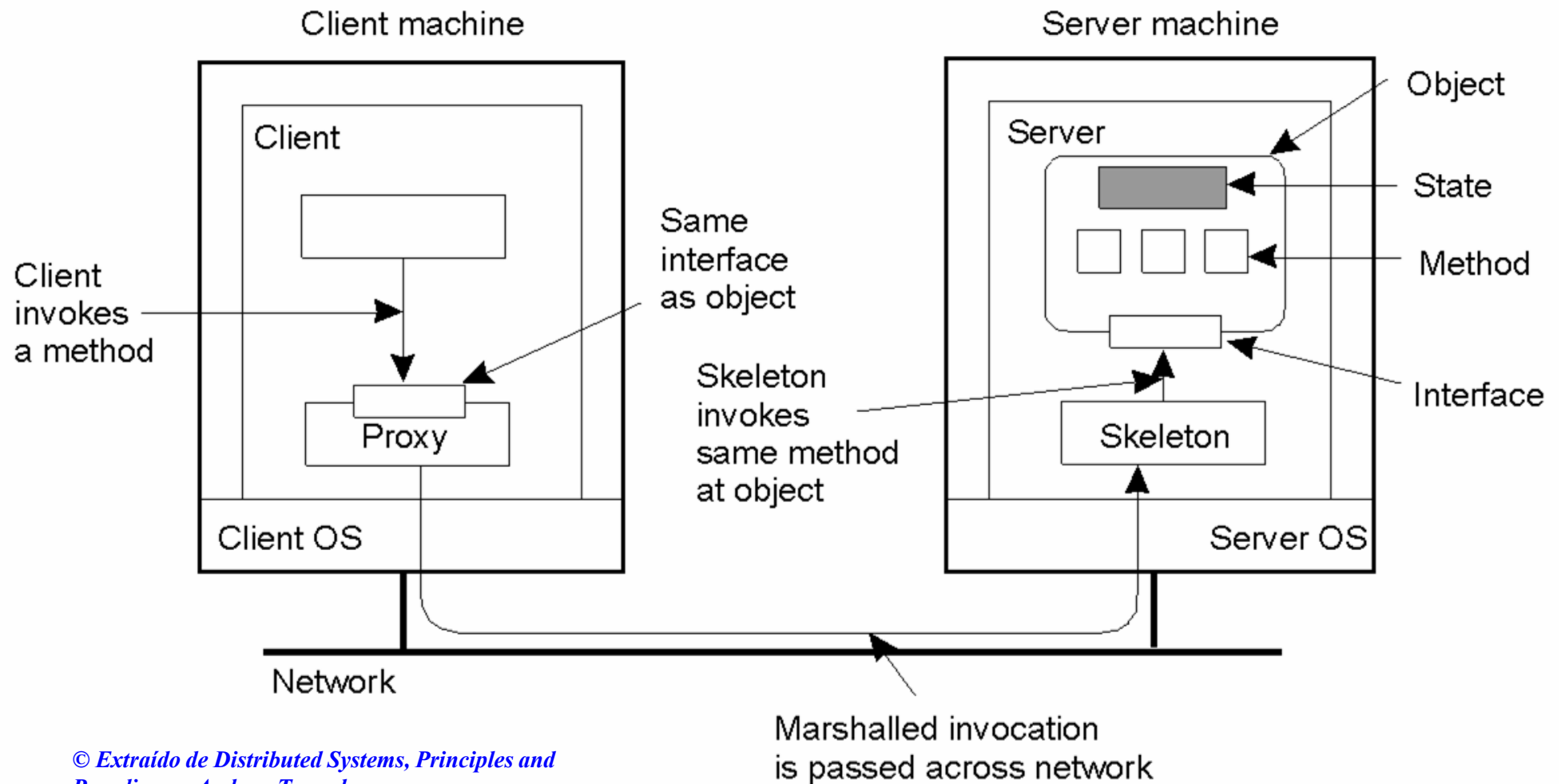
- **Sistemas compostos por instâncias de objetos**
 - ❖ Suporte à programação orientada a objetos;
 - ❖ Os objetos devem ter interfaces bem definidas;
 - ❖ Fácil interoperabilidade e transparência à distribuição

Objetos Distribuídos

Java Remote Method Invocation (RMI)

- Um objeto distribuído, ou também designado objeto remoto, é um objeto que disponibiliza uma interface acessível através de protocolos de interação entre objetos instanciados em computadores diferentes, interligados por uma infraestrutura de rede, usando mensagens num formato binário próprio do RMI.
- A semântica de chamada dos métodos remotos garante:
 - Numa chamada recebe-se sempre um resultado, sabendo-se que o target foi chamado uma única vez, ou então é recebida uma exceção.
 - Esta semântica exige o tratamento de falhas e suporte para a existência de mecanismos de *timeout*.

Invocação remota de métodos



© Extraído de *Distributed Systems, Principles and Paradigms* - Andrew Tanenbaum

Como é que o Cliente inicia a descoberta do objeto remoto?

Normalmente estes componentes são genericamente designados por *stubs*

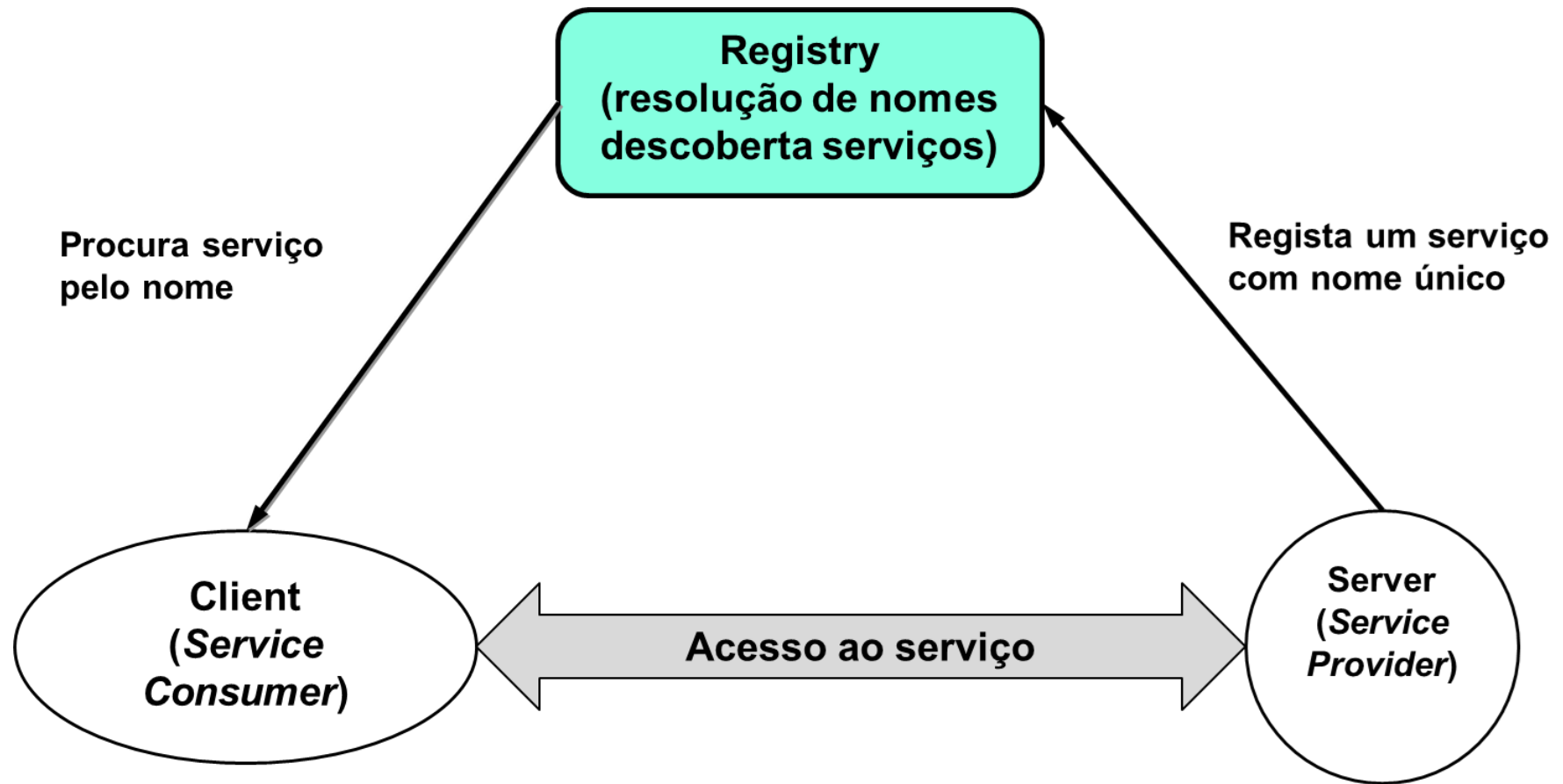
Proxy

- Torna transparente a chamada de métodos no lado do cliente, comportando-se como um objeto local em representação do objeto remoto;
- Quando recebe uma invocação redireciona-a através de uma mensagem para o objeto remoto, fazendo *marshalling* do método e respetivos parâmetros

Skeleton

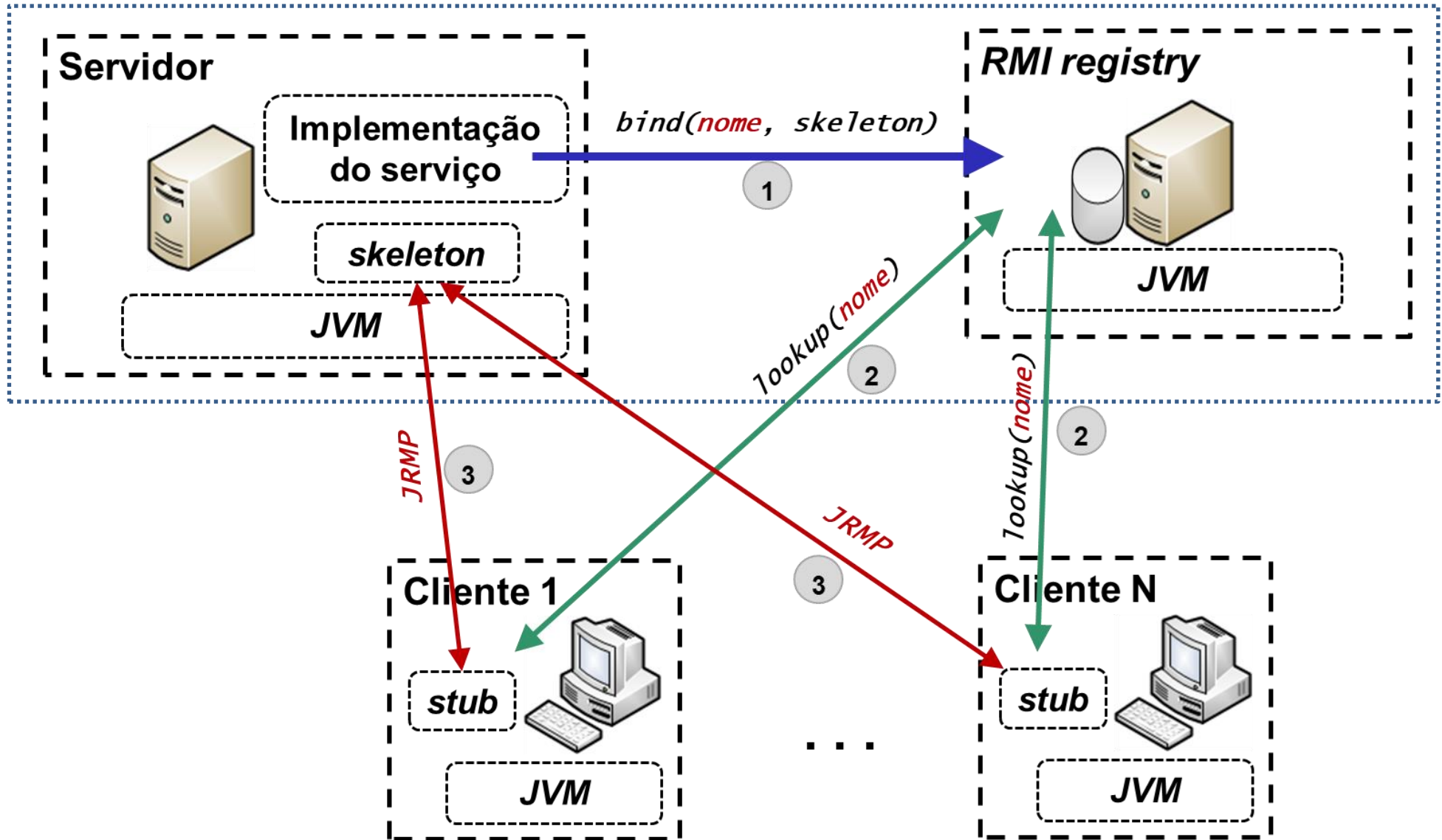
- Cada classe de um objeto remoto tem associado um *skeleton* que conhece a interface do objeto remoto e realiza as seguintes ações:
 - Faz *unmarshalling* da mensagem, enviada pelo *proxy*, com o pedido (método e argumentos);
 - Chama o método respetivo no objeto remoto;
 - Espera que a chamada termine;
 - Faz *marshalling* dos resultados, enviando uma mensagem de resposta ao *proxy* com os resultados ou eventuais exceções;

Intermediação para Registo/descoberta de serviços

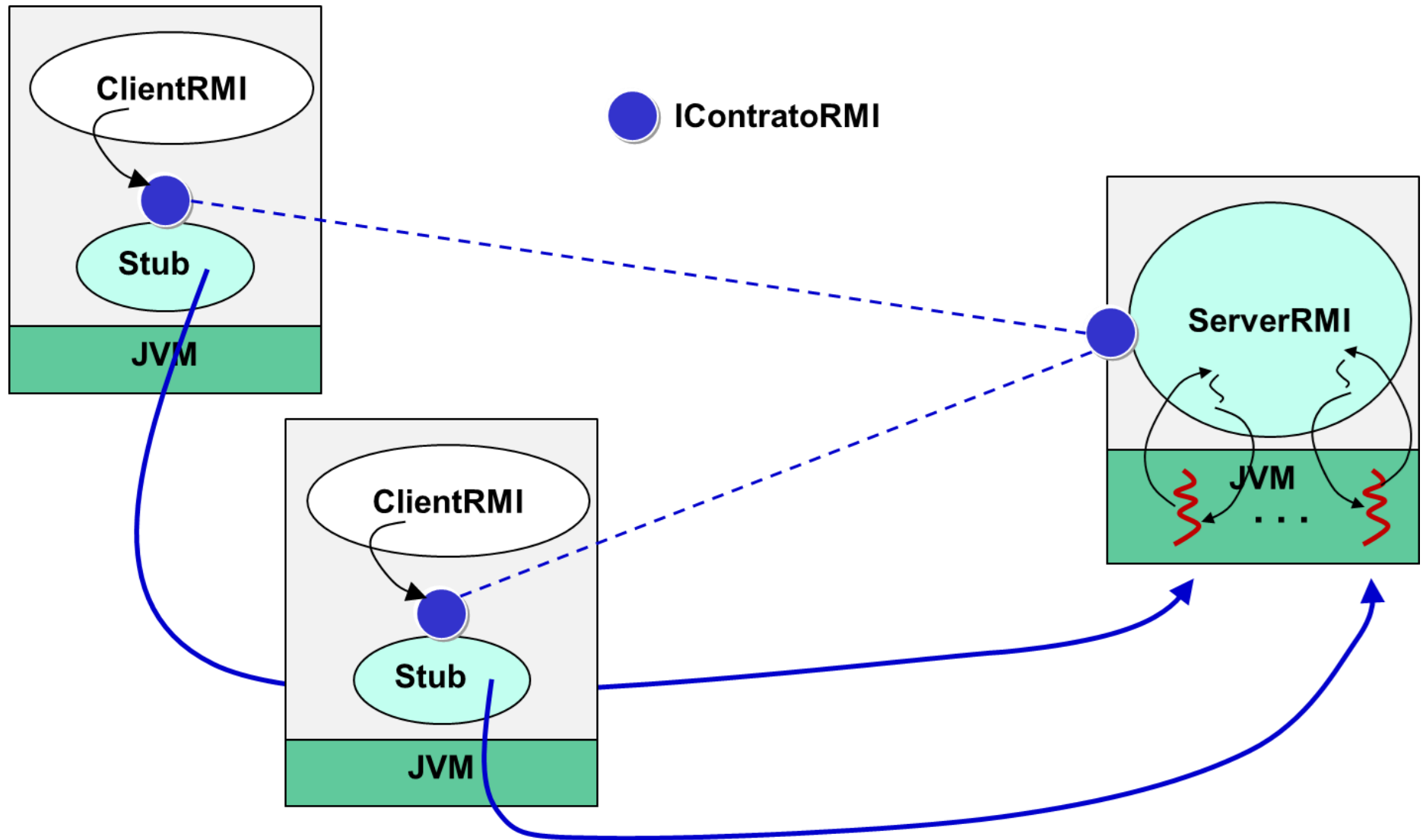


Analogia com DNS (*Domain Name Service*) para resolução de nomes e IPs

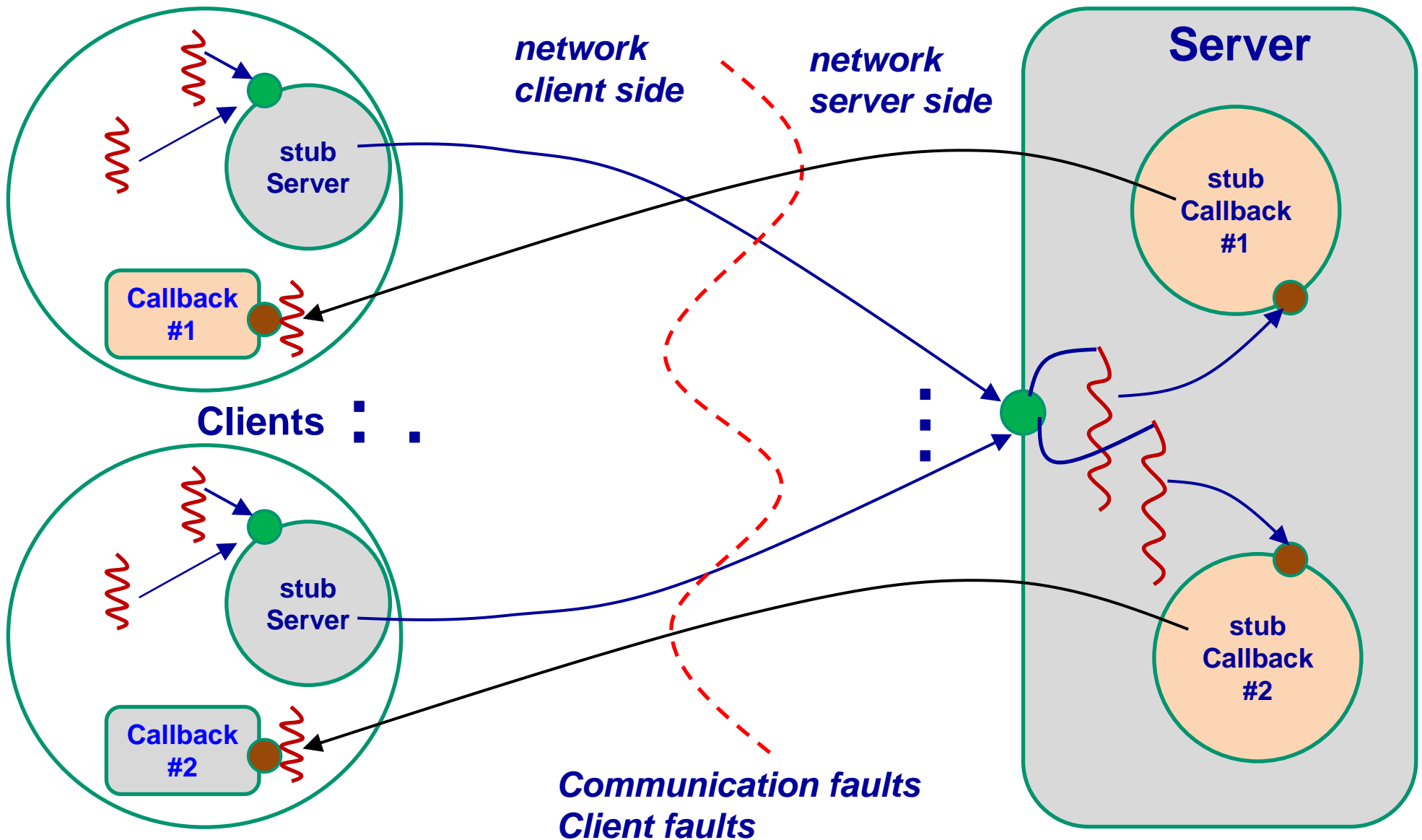
Aplicação Cliente/Servidor em Java RMI



Partilha de Contrato e concorrência implícita

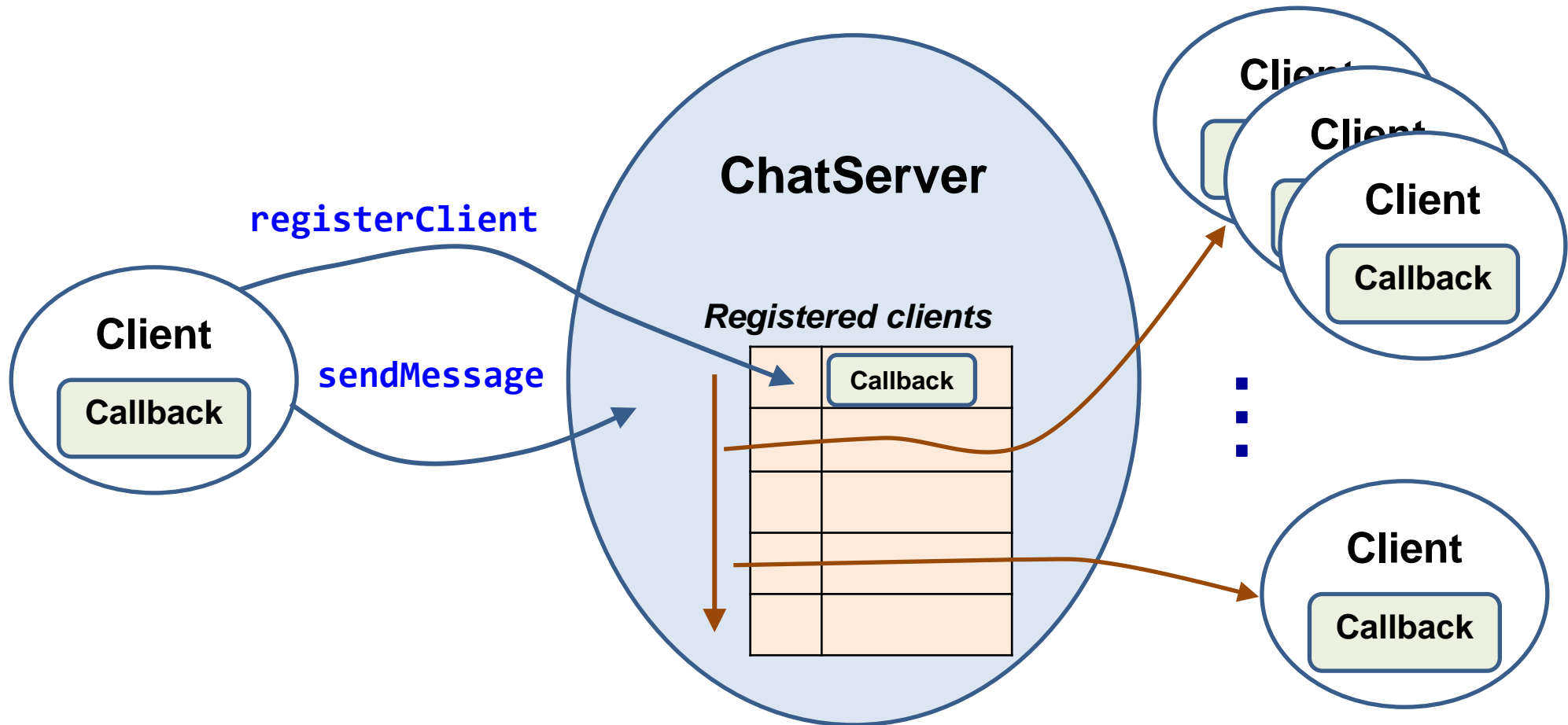


Interação por Contratos; Comunicação; Concorrência; Falhas



Exemplo RMI: *ChatServer* para troca de mensagens com múltiplos clientes

- A aplicação cliente permite registar no servidor um utilizador com um nome
- Um cliente registado pode enviar mensagens que serão recebidas por todos os clientes que estejam registados



Exemplo RMI: Contratos

```
import java.rmi.Remote;  
import java.rmi.RemoteException;
```

*Produz o artefacto
Contract.jar*

```
public interface IChatService extends Remote {  
    public void registerClient(  
        String clientName, IChatCallback callback  
    ) throws RemoteException;  
  
    public void sendMessage(  
        String senderName, String message  
    ) throws RemoteException;  
}
```

```
public interface IChatCallback extends Remote {  
    public void message(  
        String sender, String msg  
    ) throws RemoteException;  
}
```

```
public static void main(String[] args) throws Exception {
    try {
        Properties props = System.getProperties();
        props.put("java.rmi.server.hostname", serverIP);

        // create instance of server object
        svc = new ChatServer();

        // make server object available to be called in port svcPort
        IChatService stubSvc = (IChatService) UnicastRemoteObject.exportObject(svc, svcPort);

        // register server object in lookup service for clients to find it
        Registry registry = LocateRegistry.createRegistry(registerPort);
        registry.rebind("ChatServer", stubSvc);

        System.out.println("Server ready: Press any key to finish server");
        java.util.Scanner scanner = new java.util.Scanner(System.in);
        String line = scanner.nextLine();
        System.exit(0);
    } catch (RemoteException e) {
        e.printStackTrace();
    } catch (Exception ex) {
        System.err.println("Server unhandled exception: " + ex.toString());
    }
}
```

```
static String serverIP = "localhost";
static int registerPort = 7000;
static int svcPort = 7001;
static ChatServer svc = null;
```

Exemplo RMI: *ChatServer* - implementação do contrato

```
private Map<String, IChatCallback> clients=new ConcurrentHashMap<>();
```

```
@Override
```

```
public void registerClient(String clientName, IChatCallback clientCallBack) throws  
RemoteException {  
    synchronized (clients) {  
        if (!clients.containsKey(clientName))  
            clients.put(clientName, clientCallBack);  
        else throw new RemoteException("Esse nome de cliente já existe");  
    }  
}
```


```
@Override
```

```
public void sendMessage(String senderName, String msg) throws RemoteException {  
    if (!clients.containsKey(senderName))  
        throw new RemoteException("Unregister client cannot send messages");  
  
    for (String clientDestName : clients.keySet())  
        try {  
            clients.get(clientDestName).message(senderName, msg);  
        } catch (RemoteException ex) {  
            // error calling client. remove client and callback  
            System.out.println("Client "+clientDestName+" removed");  
            clients.remove(clientDestName);  
        }  
}
```

```
public static void main(String[] args) {
    try {
        Registry registry = LocateRegistry.getRegistry(serverIP, registerPort);
        IChatService svc=(IChatService)registry.lookup("ChatServer");
        System.out.print("Enter user name: ");
        Scanner input = new Scanner(System.in); String name = input.nextLine();

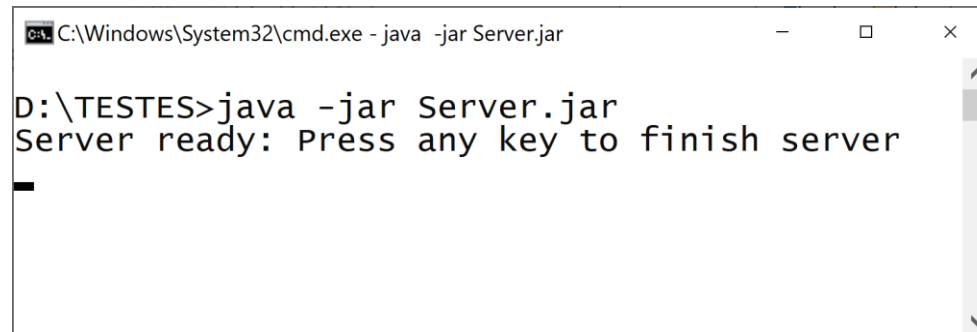
        IChatCallback callback = new MyCallback();
        Properties props = System.getProperties();
        props.put("java.rmi.server.hostname", LocalIP); // force unicast with local IP
        IChatCallback stubCallBack=(IChatCallback) UnicastRemoteObject.exportObject(callback, 0);
        svc.registerClient(name, stubCallBack);

        System.out.println("Enter lines or the word \"exit\"");
        while (true) {
            String line = input.nextLine(); if (line.equals("exit")) break;
            svc.sendMessage(name, line);
        }
    } catch (RemoteException e) {
        e.printStackTrace();
    } catch (Exception ex) {
        System.err.println("Client unhandled exception: " + ex.toString());
        ex.printStackTrace();
    }
    System.exit(-1);
}
```

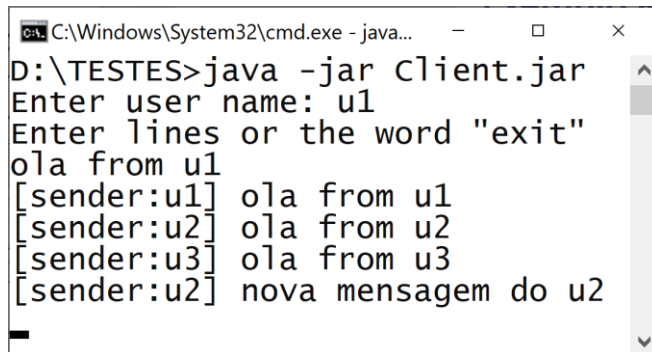


Exemplo RMI: Cliente – Implementação do contrato de *Callback*

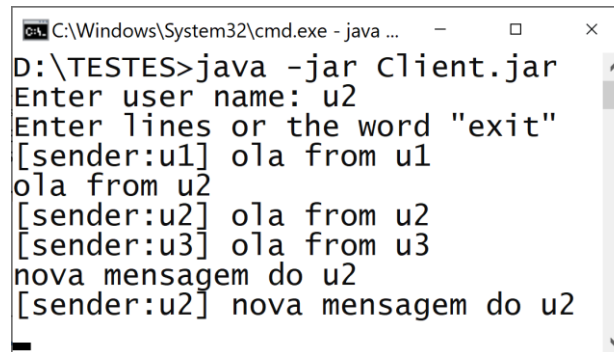
```
public class MyCallback implements IChatCallback {  
  
    @Override  
    public void message(String senderName, String msg) throws RemoteException {  
        System.out.println("[sender:" + senderName + "] " + msg);  
    }  
}
```



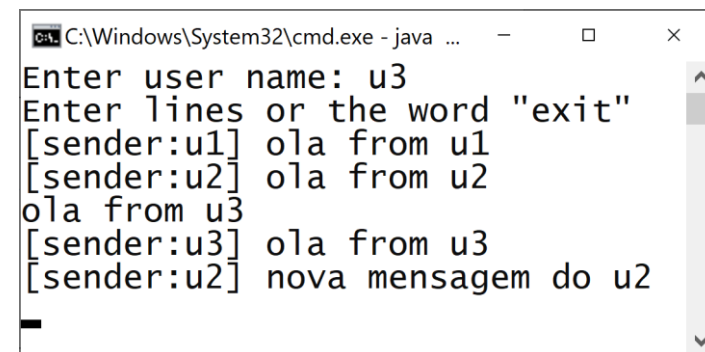
```
C:\Windows\System32\cmd.exe - java -jar Server.jar  
  
D:\TESTES>java -jar Server.jar  
Server ready: Press any key to finish server
```



```
C:\Windows\System32\cmd.exe - java...  
  
D:\TESTES>java -jar Client.jar  
Enter user name: u1  
Enter lines or the word "exit"  
ola from u1  
[sender:u1] ola from u1  
[sender:u2] ola from u2  
[sender:u3] ola from u3  
[sender:u2] nova mensagem do u2
```



```
C:\Windows\System32\cmd.exe - java ...  
  
D:\TESTES>java -jar Client.jar  
Enter user name: u2  
Enter lines or the word "exit"  
[sender:u1] ola from u1  
ola from u2  
[sender:u2] ola from u2  
[sender:u3] ola from u3  
nova mensagem do u2  
[sender:u2] nova mensagem do u2
```



```
C:\Windows\System32\cmd.exe - java ...  
  
D:\TESTES>java -jar Client.jar  
Enter user name: u3  
Enter lines or the word "exit"  
[sender:u1] ola from u1  
[sender:u2] ola from u2  
ola from u3  
[sender:u3] ola from u3  
[sender:u2] nova mensagem do u2
```