COMPUTAÇÃO NA NUVEM

ISEL – LEIRT / LEIC

Evolução dos mecanismos para chamadas remotas

(Parte 1)

José Simão jsimao@cc.isel.ipl.pt

Luís Assunção lass@isel.ipl.pt

Que técnicas são usadas para desenvolver os serviços que suportam a *Cloud*?

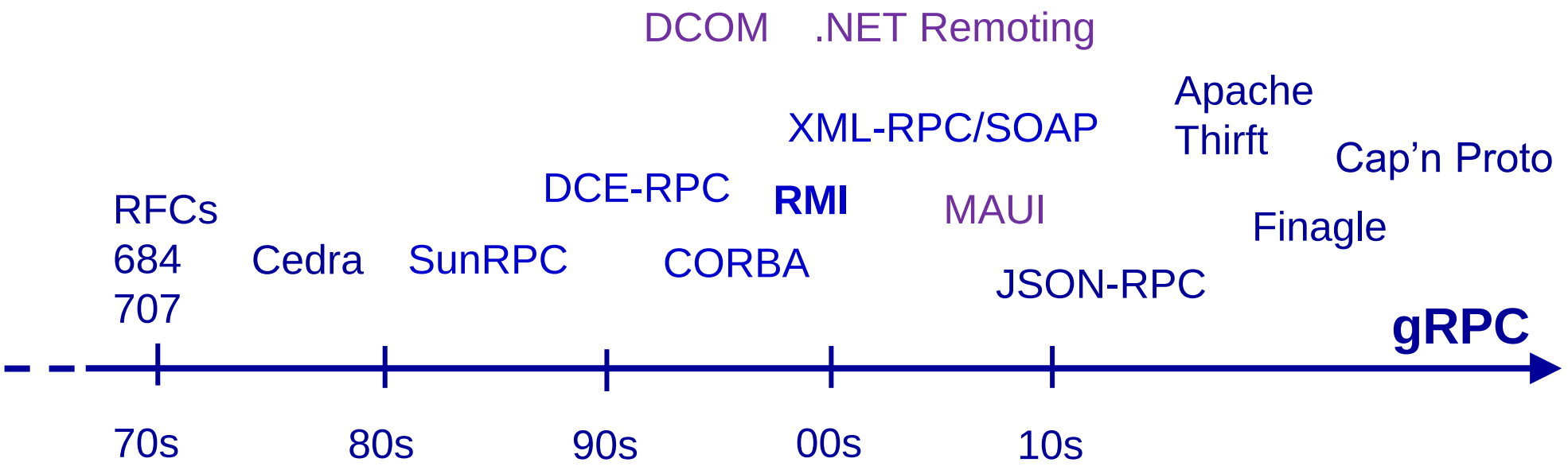
➢ Perfil procurado pela Amazon AWS em 2019:

- *Deep understanding of distributed systems and web services technology;*
- *If you have ever pondered about CAP theorem, consistent hashing, multi-master replication, merkle trees, leader election or Paxos Algorithm, gossip protocols, tiered storage, this is an opportunity to get your hands dirty with a real-world solution implementing these distributed system concepts;*
- *Software Development Engineer – In-Memory Distributed Systems Strong at applying data structures, algorithms, and object oriented design, to solve challenging problems. Experience working with REST and RPC service patterns and other client/server interaction models;*

Evolução das chamadas remotas

- *Sockets TCP/IP*
- Objetos distribuídos - *Java RMI*
- *Remote Procedure Call* - Google RPC (*gRPC*)

Evolução dos *middlewares* para chamadas remotas



DCE: Distributed Computing Environment
CORBA: Common Object Request Broker Architecture
RMI: Remote Method Invocation
MAUI: Mobile Assistance Using Infrastructure by Microsoft
Thrift: A*synchronous* 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

http://dist-prog-book.com/chapter/1/rpc.html#introduction

Sockets TCP/IP

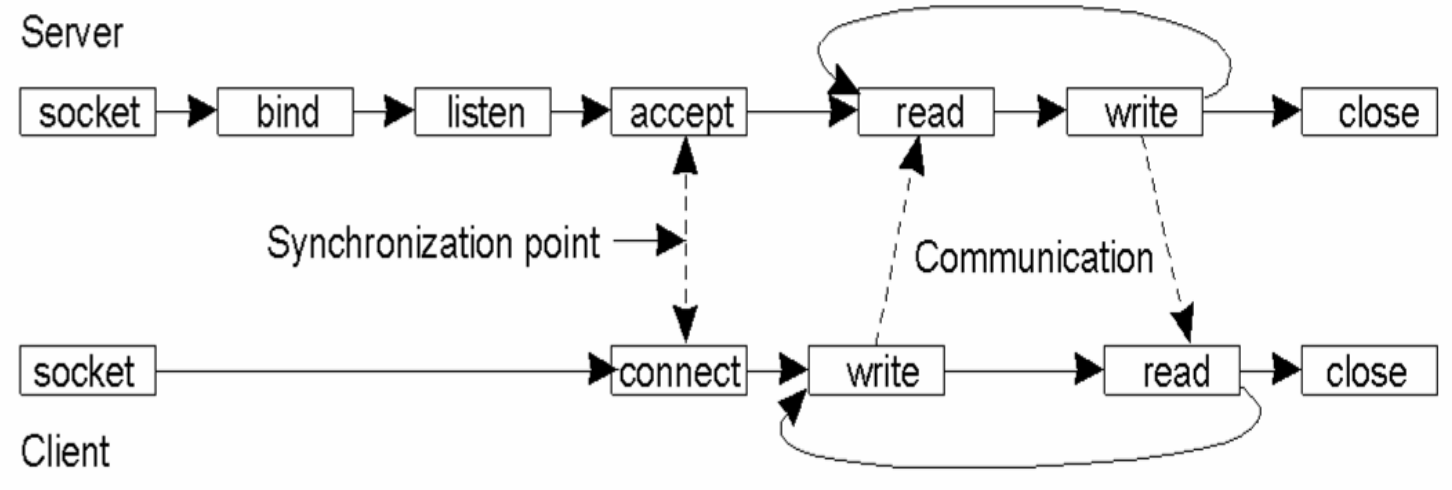
- O TCP é um protocolo "*connection oriented*" que permite estabelecer conexões para comunicar dados entre dois pontos, em que um ponto tem o papel de servidor e o outro ponto tem o papel de cliente;

- O servidor espera por conexões e o cliente estabelece a conexão para o servidor;

- 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;

- 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.

Sockets TCP/IP como definidos na década de 80



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

Primitiva	Descrição
Socket	Cria um novo ponto de comunicação (endpoint)
Bind	Associa um socket a um porto estabelecendo o par (IP, port)
Listen	Anuncia a disponibilidade para aceitar ligações
Accept	Bloqueia a *Thread caller* até chegar um pedido
Connect	Estabelece uma ligação para um par (IP, port)
Send/write	Envia dados pela ligação estabelecida
Receive/read	Recebe dados pela ligação estabelecida
Close	Termina a ligação

Sockets TCP/IP (em Java)

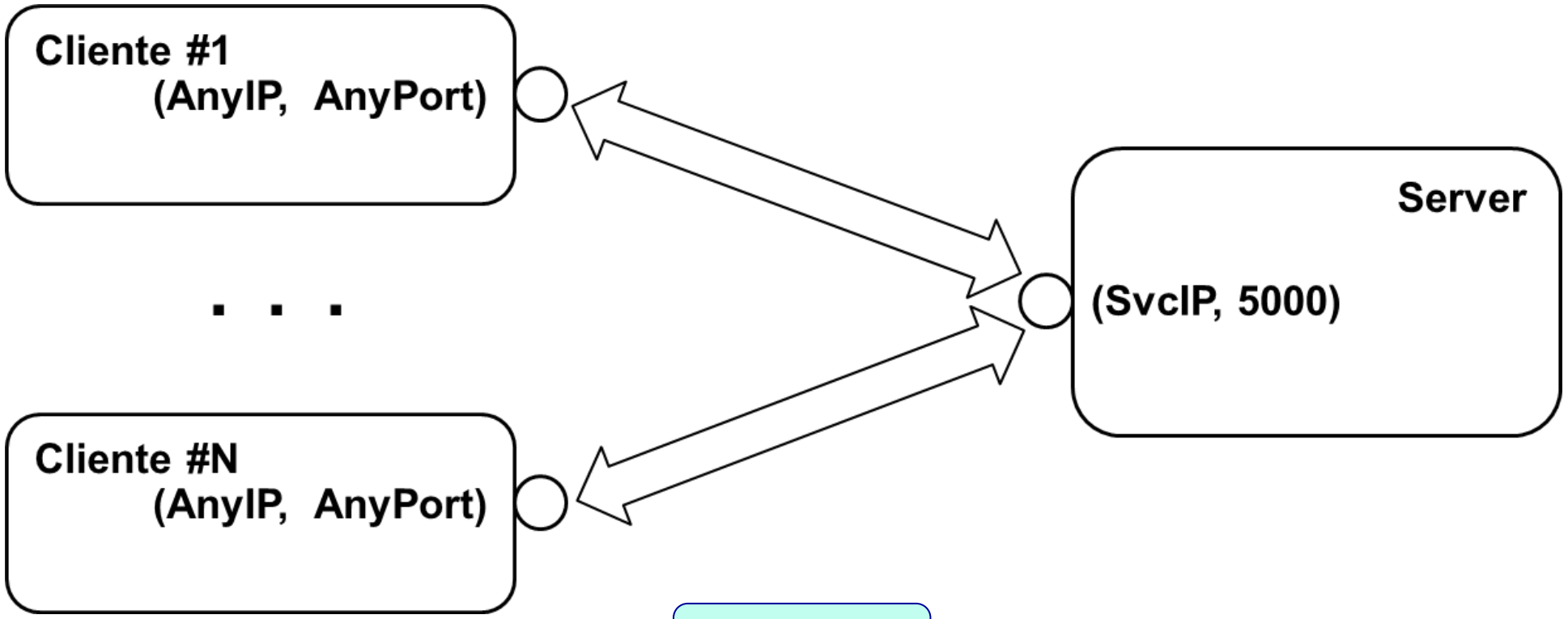
package java.net

- Classe ServerSocket (Construtor: definição do porto TCP, IP, dimensão da fila de conexões etc.)
  - bind: definição da associação (IP, porto TCP)
  - accept: Bloqueia até receber uma conexão. Retorna um objeto Socket

- Classe Socket:
  - *bind: associa o socket a um par (IP, porto)*
  - *connect:* Conecta o socket a um endpoint (IP, porto)
  - getInputStream: obtém um InputStream para ler dados do socket
  - getOutputStream: obtém um OutputStream para escrever dados no socket
  - close: Fecha o socket

- Classes úteis:
  - *java.net.InetSocketAddress: permite criar associações (IP, port);*
  - java.io.BufferedReader: ler caracteres de uma InputStream
  - java.io.InputStreamReader: uma ponte entre um byte stream e char stream
  - java.io.PrintWriter: escreve objetos num output stream de texto

Sockets TCP/IP (em Java)



Exemplo

- Cada cliente envia uma linha de texto com múltiplas palavras separadas por espaço
- O servidor processa a linha de texto devolvendo a palavra de maior dimensão

Sockets TCP/IP - Servidor

SERVIDOR

```
ServerSocket svcSocket = new ServerSocket(svcPort);
for(;;) {
  try {
    // block until receive connection
    Socket cliSocket = svcSocket.accept();
    BufferedReader inStream = new BufferedReader(
       new InputStreamReader(cliSocket.getInputStream())
    );
    PrintWriter outStream = new PrintWriter(cliSocket.getOutputStream(), true);
    String request = inStream.readLine(); // receive the request
    String response = processar(request);
    // send the reply
    outStream.println("biggest word: "+response + " size="+response.length());
    cliSocket.close();
  } catch (IOException ex) {
    System.out.println("Server crashed!"); System.exit(-1);
  }
}
```

O servidor processa os pedidos em série.
Se o processamento for demorado os clientes terão de esperar !

Sockets TCP/IP - Server: Processamento do pedido

```
public static String processar(String request) throws InterruptedException {
    String[] tmp = request.split(" ");
    String bigWord = "";
    for (String s : tmp) {
        if (s.length() > bigWord.length()) {
            bigWord = s;
        }
    }
    // simula tempo de processamento
    Thread.sleep(10*1000);
    // ...
    return bigWord;
}
```

Cada pedido demora a processar 10 segundos

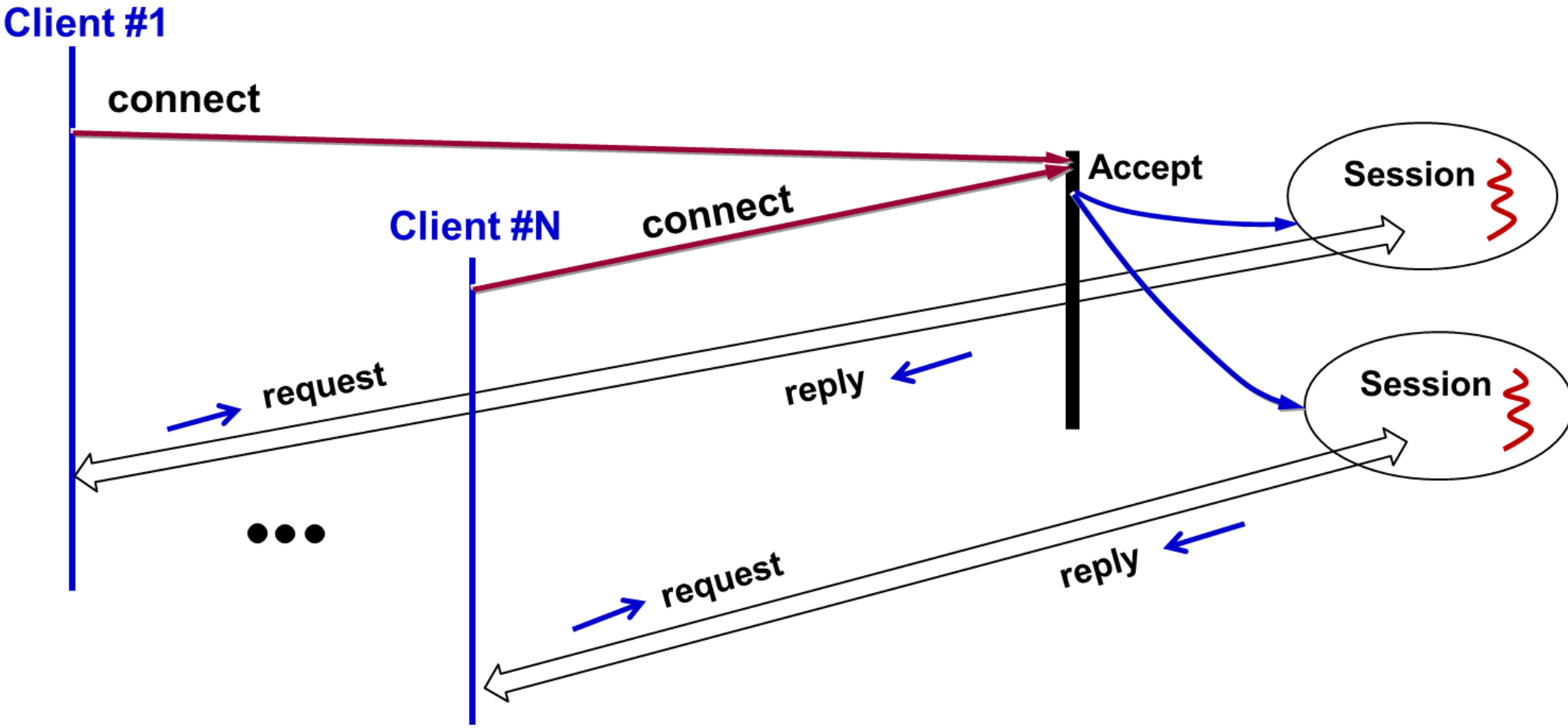
Socket TCP/IP - Cliente

CLIENTE

```
public static void main(String[] args) throws Exception {
    Socket client = new Socket(args[0], Integer.parseInt(args[1]));
                            // (IP server, port)
    // Stream to write to
    PrintWriter outSock = new PrintWriter(client.getOutputStream(), true);
    // Stream to read from
    BufferedReader inSock = new BufferedReader(
                              new InputStreamReader(client.getInputStream())
                            );

    long start = System.currentTimeMillis();
    // write command with word to search
    outSock.println("A pangram: The quick brown fox jumps over the lazy dog");
    // read response
    System.out.println(inSock.readLine());
    long end = System.currentTimeMillis();

    System.out.println("Operation completed in:" + (end-start) + " ms");
    client.close();
}
```

Qual o tempo esperado no caso de existirem múltiplos clientes ?

Sockets TCP/IP - Servidor concorrente



Server: Por cada cliente cria uma *Session concorrente*

```
public static void main(String[] args) throws Exception {
    System.out.println("Server concurrent on port 5000");
    ExecutorService executor = Executors.newFixedThreadPool(5);
    ServerSocket svcSocket = new ServerSocket(5000);
    int sessionId = 0;
    for (;;) {
        System.out.println("Accepting new connections... ");
        Socket client = svcSocket.accept();
        System.out.println("New connection with... " + client);
        Runnable worker = new Session(client, sessionId);
        executor.execute(worker);
    }
}
```

```
class Session implements Runnable {

    public Session(Socket cliSocket, int id) { ... }

    @Override
    public void run() { ... }
}
```

Server: Uma possível implementação de Session

```
static class Session implements Runnable {
    Socket cliSocket=null; int id; BufferedReader inStream=null; PrintWriter outStream=null;

    public Session(Socket cliSocket, int id) {
        this.cliSocket = cliSocket;  this.id = id;
        try {
            inStream = new BufferedReader(new InputStreamReader(cliSocket.getInputStream()));
            outStream = new PrintWriter(cliSocket.getOutputStream(), true);
        } catch (IOException e) { e.printStackTrace(); }
    }

    @Override
    public void run() {
        try {
            String line = inStream.readLine();
            String[] tmp = line.split(" ");
            String bigWord = "";
            for (String s : tmp) { if (s.length() > bigWord.length()) { bigWord = s; } }
            Thread.sleep(10*1000); // simula tempo de processamento
            outStream.println("biggest word: " + bigWord + " size = " + bigWord.length());
            cliSocket.close();
            System.out.println("Session " + id + " terminates");
        } catch (Exception ex) { System.out.println("Server session " + id + " crashed!"); }
    }
}
```

Conclusão:

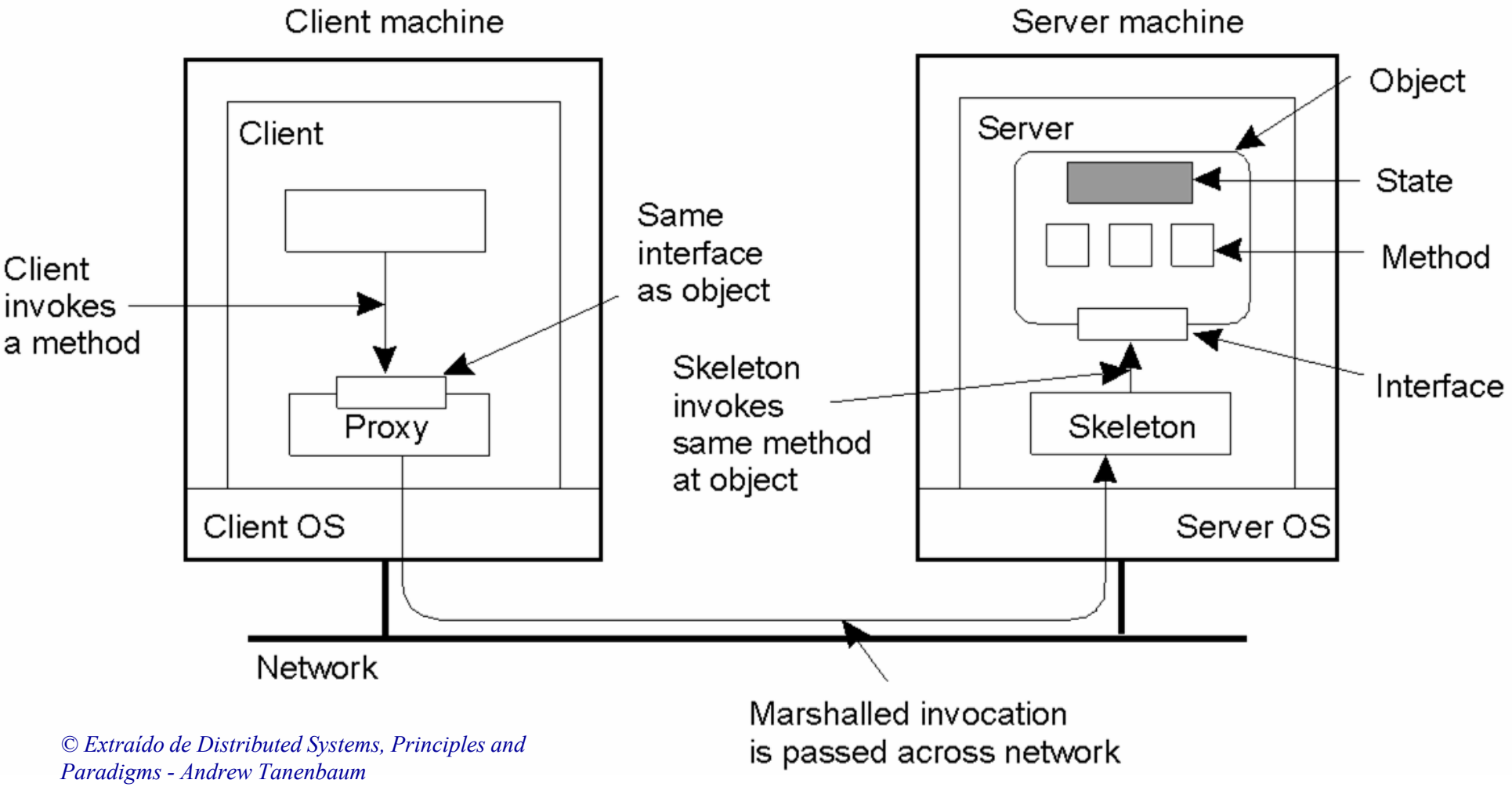
- A Concorrência no servidor tem de ser explícita e depende do programador;
- O protocolo de troca de mensagens (*request/reply*) depende de dados em *stream* e por isso implica uma serialização explícita dos objetos para formatos Xml, Json etc.
- Devido à existência de *middlewares* com maiores abstrações, raramente é necessário desenvolver aplicações baseadas em *sockets*. No entanto, esses *middleware* usam no mais baixo nível *sockets*.

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.

Invocação remota de métodos



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

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

Cliente (*Proxy*) / Servidor (*Skeleton*)

Proxy

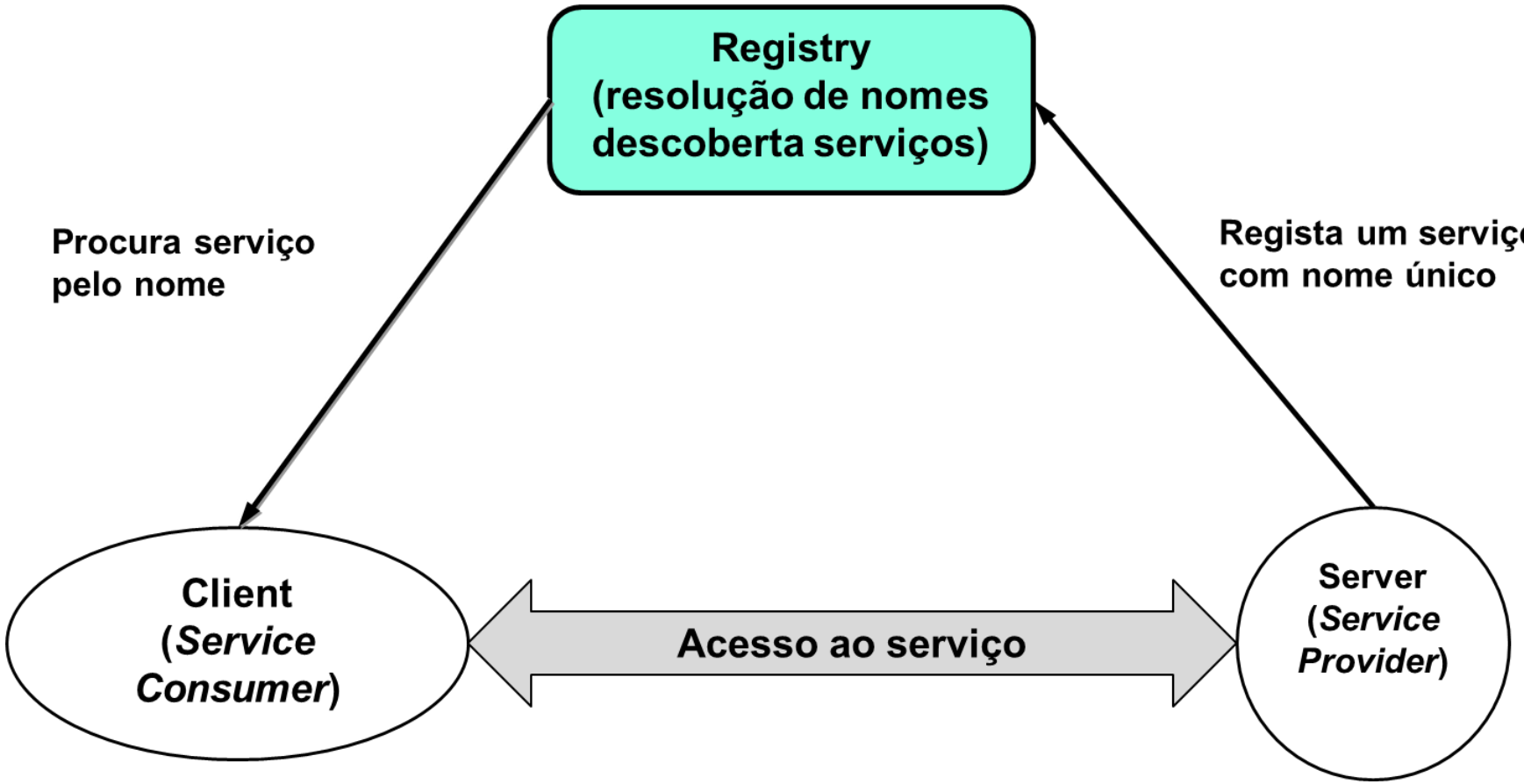
Normalmente estes componentes são genericamente designados por *stubs*

- 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

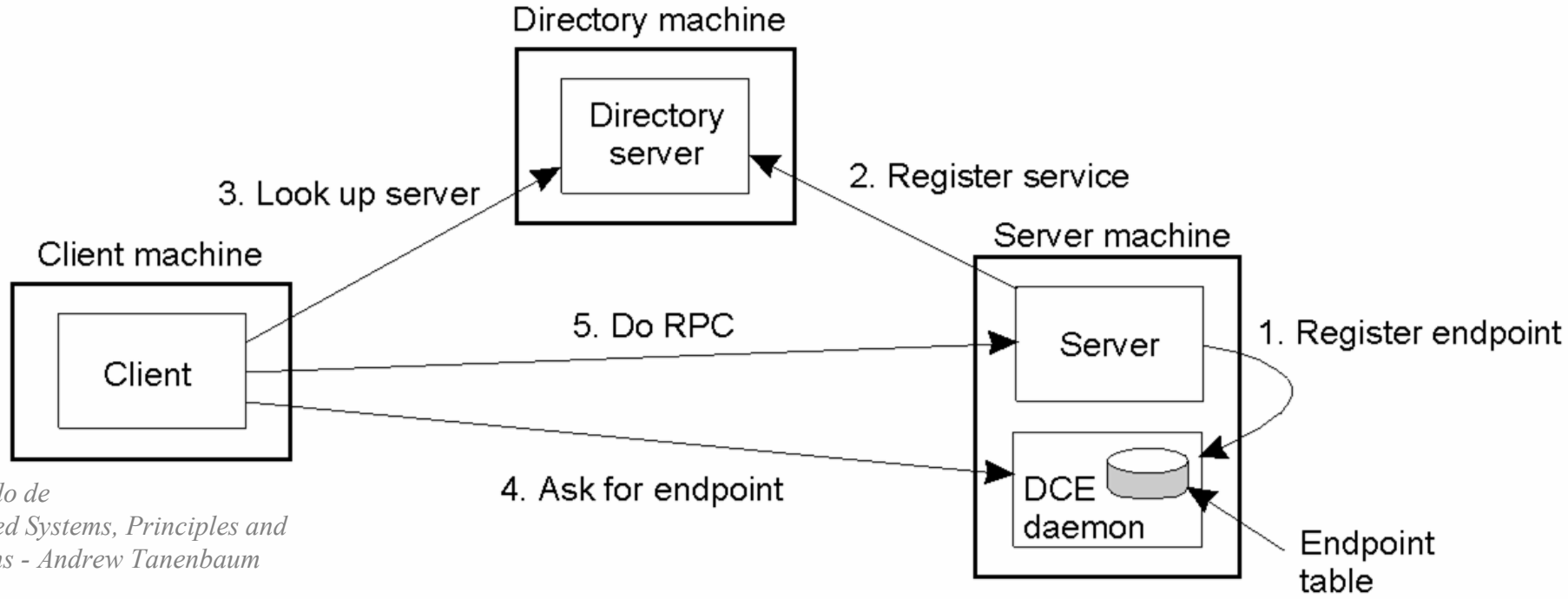
- Cada classe de um objeto remoto tem associado um *skeleton* que conhece a interface do objeto remoto e realiza as seguintes ações:
  - Faz *unmarshalling* da mensagem, enviada pelo *proxy*, com o pedido (método e argumentos);
  - Chama o método respetivo no objeto remoto;
  - Espera que a chamada termine;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

Intermediação (Serviço de Nomes/Diretoria)



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

- Na década de 80/90 é definido um standard designado por DCE (*Distributed Computing Environment*) formando um triângulo, em que um serviço intermediário (*Directory server*) regista os serviços (*Server*) e onde os clientes procuram (*Look up*) os serviços disponíveis;
- A existência de um serviço local (*DCE daemon*) permitia que um mesmo computador disponibilizasse múltiplos serviços em portos TCP diferentes

Semântica de chamadas remotas versus falhas

- ***Exactly once***: Numa chamada é garantido que o target é executado uma única vez. A chamada de métodos em objetos locais tem esta semântica;

Porém:

- Em ambiente distribuído podem ocorrer falhas que podem ser mascaradas das seguintes formas:
  - Retransmitir a mensagem de chamada até receber uma resposta ou assumir que houve uma falha e lançar exceções;
  - Detetar/filtrar mensagens de chamada duplicadas ou em falta do lado do *target* remoto;
  - Retransmitir resultados sem executar de novo o código no *target* remoto

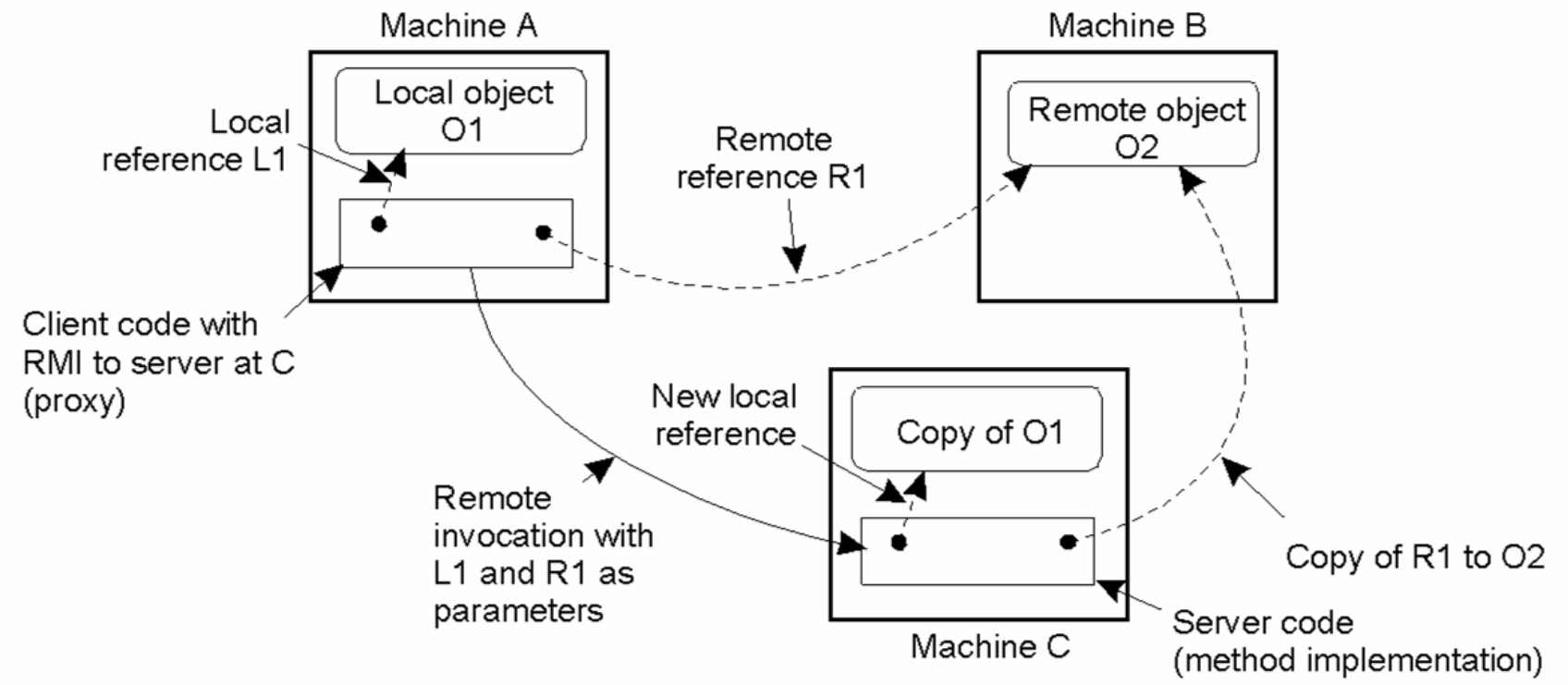
Complexo pois exigiria acoplamento forte com estado sobre a conexão

Semântica de chamadas remotas versus falhas

- ***Maybe***: Numa chamada não é possível saber se o target foi executado ou não.
  - Pode acontecer quando não são usados mecanismos de tolerância a falhas, por exemplo, perda de mensagens; falha do processo ou máquina que contém o target remoto.
- ***At-least-once***: Numa chamada sabe-se que o target foi executado pelo menos uma vez, recebendo sempre um resultado ou uma exceção.
  - No caso de existir retransmissão de mensagens o target remoto pode ser executado mais que uma vez, podendo causar resultados inconsistentes;
  - Na presença desta semântica o target deve ser idempotente, isto é, pode executar-se repetidamente causando o mesmo efeito.
- ***At-most-once***: 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*.

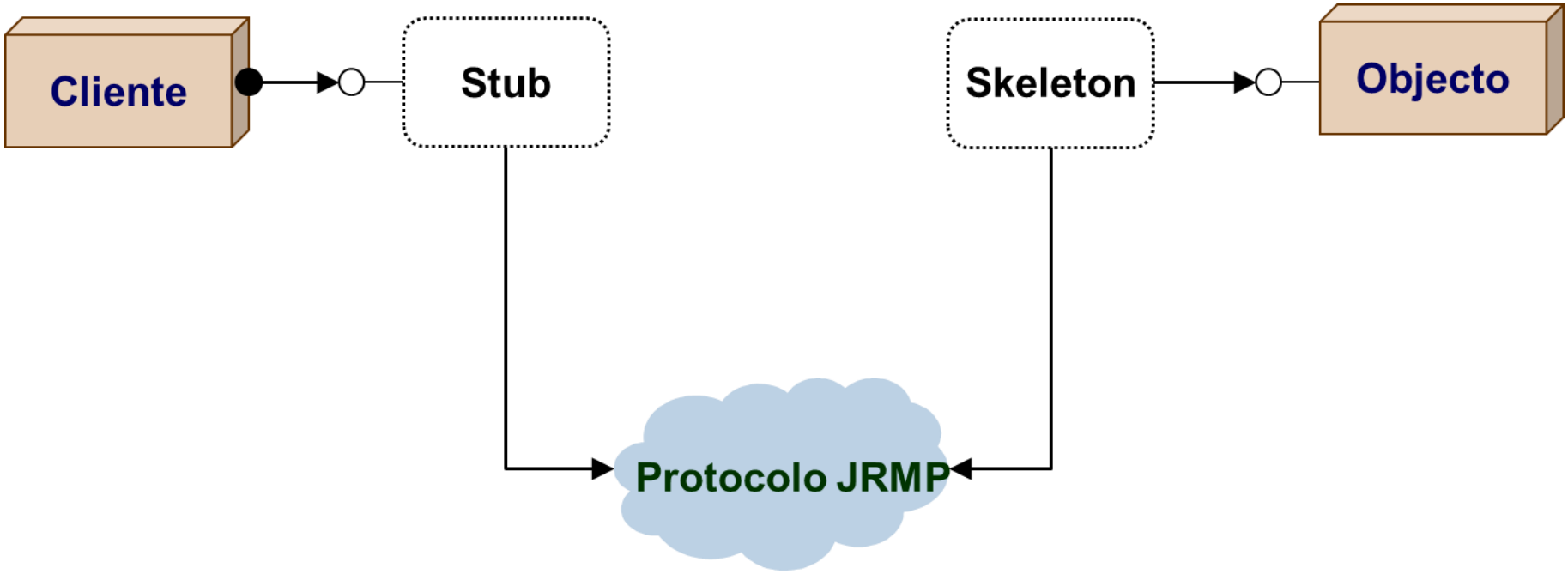
Semântica atualmente mais usada

Passagem de objetos por referência e por valor



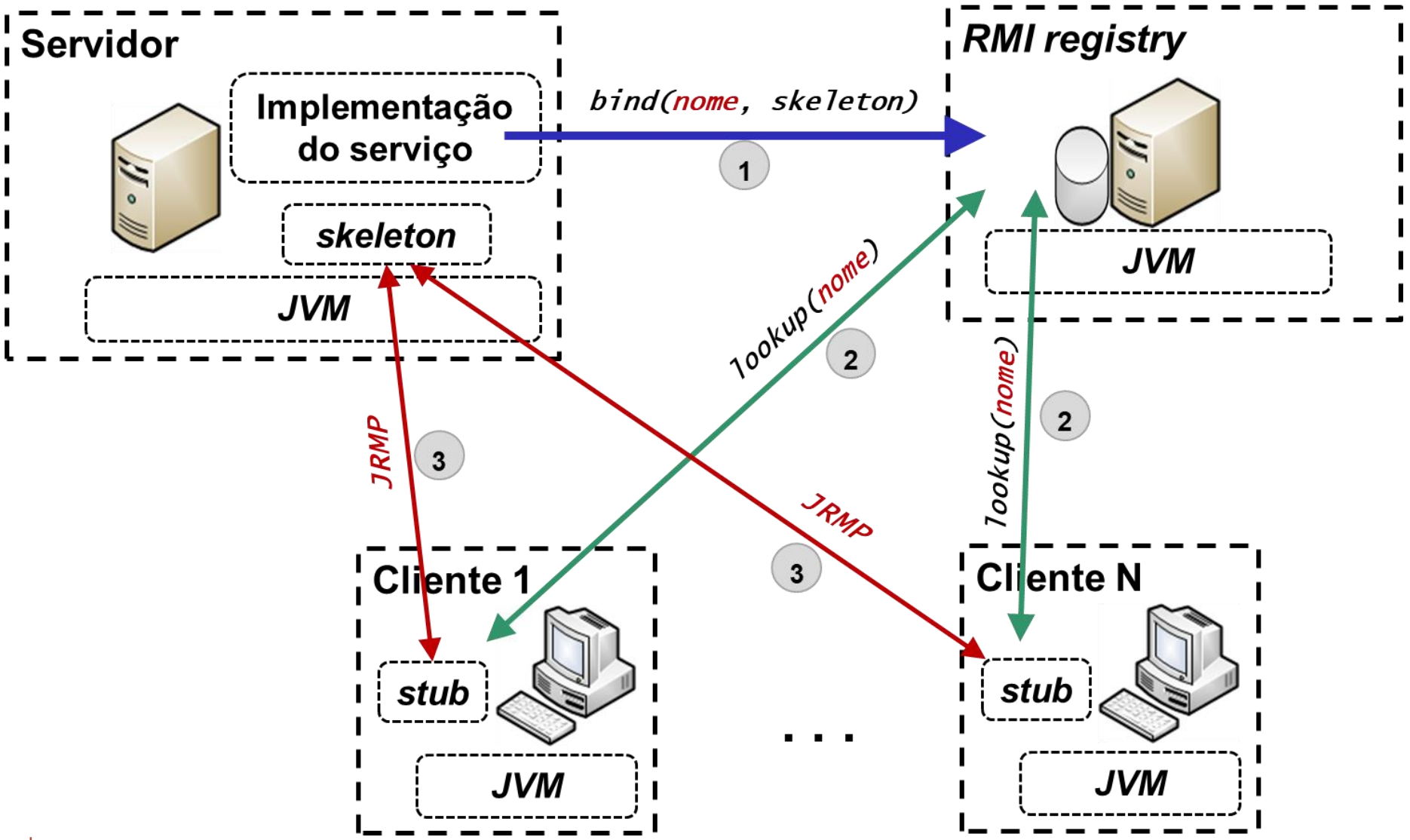
- O cliente na máquina A passa ao servidor na máquina C o objeto O1 por valor e a referência R1 para o objeto O2 que reside na máquina B.
- Salienta-se os desafios na implementação da passagem de referências para objetos (R1) entre máquinas;

Acesso a objetos remotos via RMI



JRMP – Java Remote Method Protocol

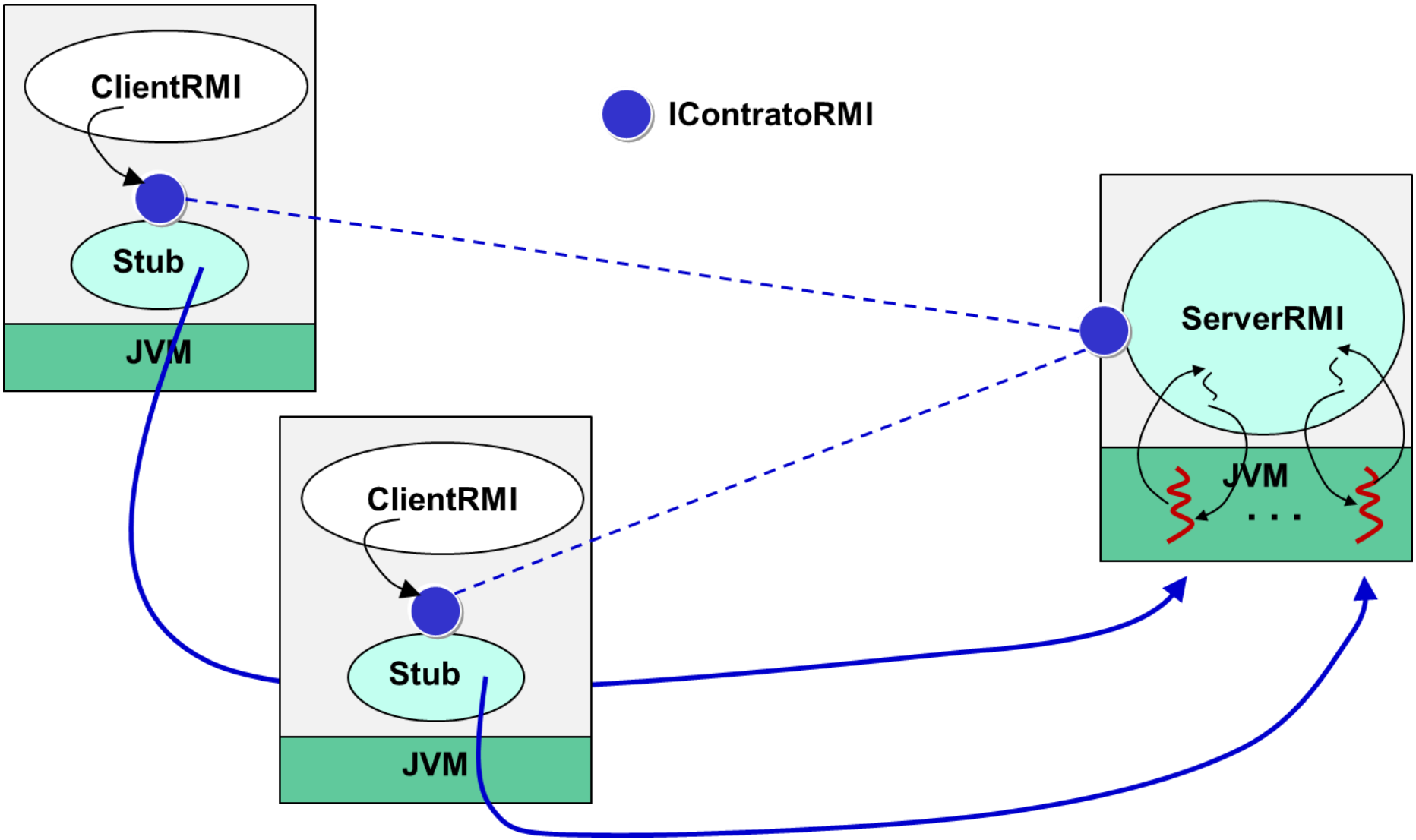
Aplicação Cliente/Servidor em Java RMI



Acoplamento entre as partes

- 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 especificado em linguagens específicas (ex. IDL, XML WSDL, Protocol buffer, etc.) ou simplesmente em artefactos das próprias linguagens de programação (ex: JAR em Java ou DLL em .NET)

Partilha de Contrato e concorrência implícita



Exemplo RMI: Especificação do Contrato

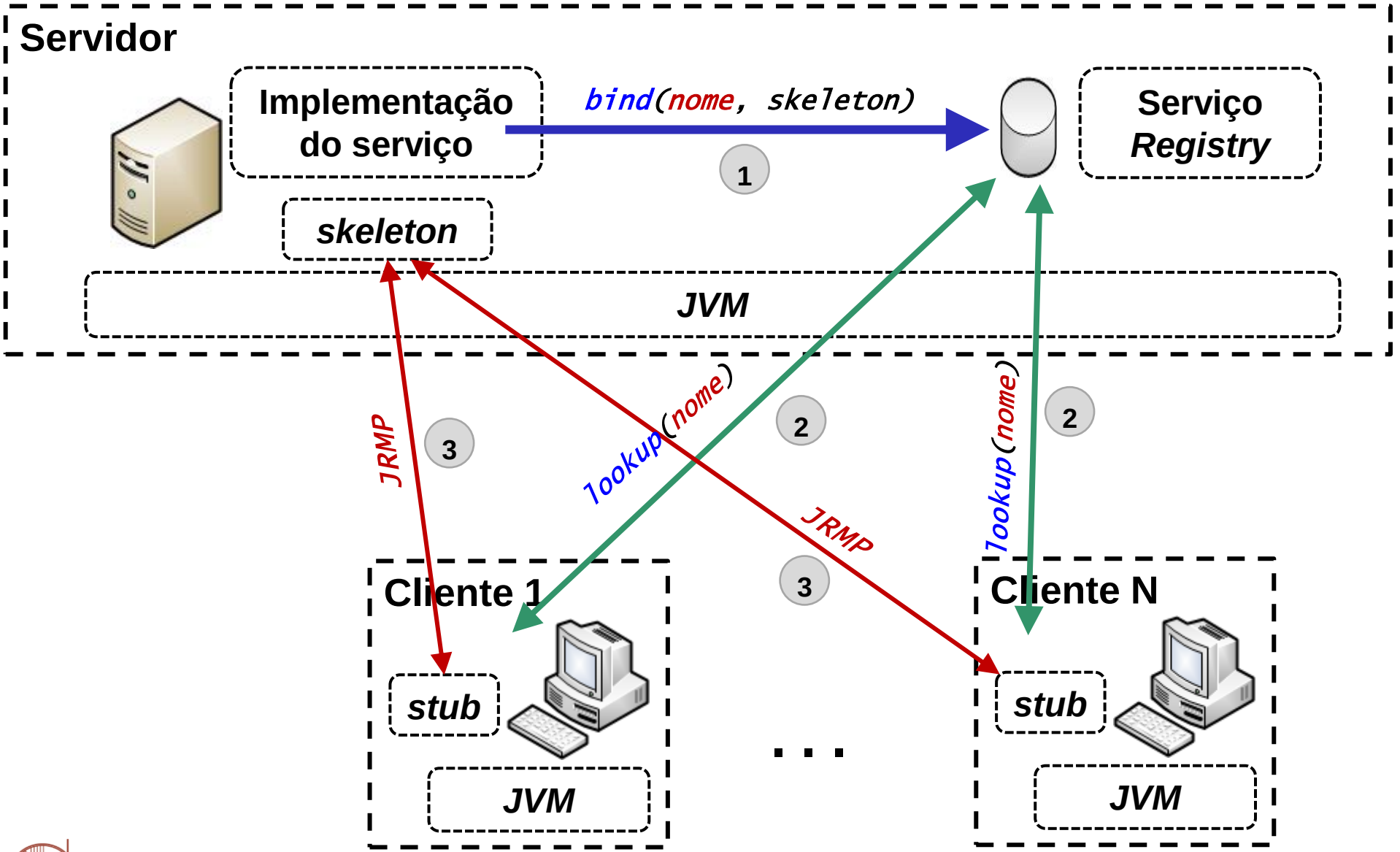
```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface IContratoRMI extends Remote {
    Reply doRemoteWork(Request req) throws RemoteException;
}
```

Geração de artefacto Contrato.jar

```
import java.io.Serializable;
import java.util.List;

public class Request implements Serializable {
   private List<String> textLines;
   public List<String> getTextLines() {
        return textLines;
   }

   public void setTextLines(List<String> linex)
   {
        this.textLines = linex;
   }
}
```

```
public class Reply implements Serializable {
    private String word;
    private int maxSize;

    public String getWord() {return word;}

    public void setWord(String word) {
       this.word = word;
    }

    public int getMaxSize() {return maxSize;}

    public void setMaxSize(int maxSize) {
       this.maxSize = maxSize;
    }
}
```

Servidor e *Registry* na mesma JVM



Exemplo RMI: Server

```
public class ServerRMI implements IContratoRMI {
  static String serverIP="localhost";
  static int registerPort = 7000;  static int svcPort = 7001;
  static ServerRMI svc = null;
  public static void main(String[] args) {
    try {
      Properties props = System.getProperties();
      props.put("java.rmi.server.hostname", serverIP);

      svc = new ServerRMI();
      IContratoRMI stubSvc=(IContratoRMI)UnicastRemoteObject.
                                          exportObject(svc, svcPort);
      Registry registry = LocateRegistry.createRegistry(registerPort);
      registry.rebind("RemoteServer", stubSvc);  //regista skeleton com nome lógico
      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());
    }
  }
```

A classe do servidor pode estender da classe base `UnicastRemoteObject` sendo as instâncias já *stubs*

Permite sobreposição do registo com o mesmo nome

Exemplo RMI: O servidor - implementação do contrato

```
    @Override
    public Reply doRemoteWork(Request request) throws RemoteException {
        int size = 0; String bigWord = "";
        for (String line : request.getTextLines()) {
            String[] tmp = line.split(" ");
            for (String s : tmp) {
                if (s.length() > size) {
                    size = s.length();  bigWord = s;
                }
            }
        }
        Reply rpy = new Reply();
        rpy.setWord(bigWord); rpy.setMaxSize(size);
        return rpy;
    }

}  // end class ServerRMI
```

Exemplo RMI: Cliente - Utilização do Contrato

Depende do artefacto Contrato.jar

```
public class ClientRMI {
   static String serverIP="localhost";
   static int registerPort = 7000;

  public static void main(String[] args) {
    try {
       Registry registry = LocateRegistry.getRegistry(serverIP,registerPort);
       IContratoRMI svc=(IContratoRMI)registry.lookup("RemoteServer");

       List<String> lines = LerNlinhas();
       Request req=new Request();
       req.setTextLines(lines);

       Reply rpy=svc.doRemoteWork(req);

       System.out.println("MaxSize "+rpy.getMaxSize()+" word:"+rpy.getWord());
     } catch (RemoteException e) {
            e.printStackTrace();
     } catch (Exception ex) {
            System.err.println("Client unhandled exception: " + ex.toString());
     }
  }
```

Interação por Contratos; Comunicação; Concorrência; Falhas