UNIVERSIDADE DE ÉVORA

Sistemas Distribuídos

Objetos Distribuídos e Invocação Remota

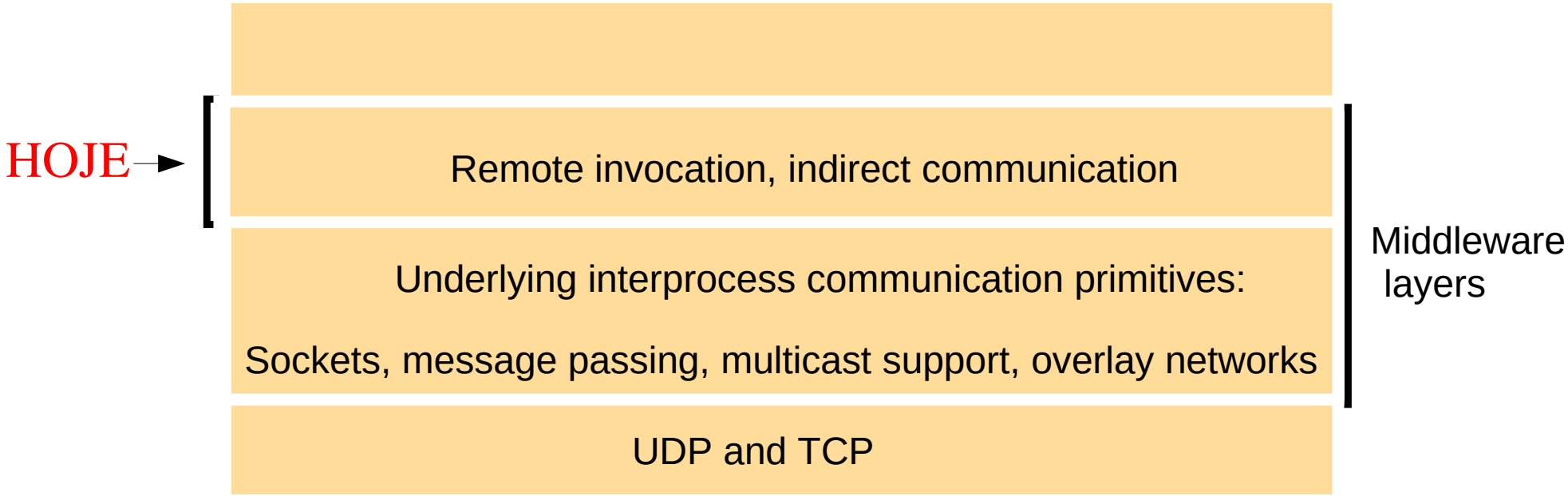
Introdução

- modelos de programação para aplicações distribuídas
  - compostas por vários programas que cooperam e são executados em distintos **processos** simultâneos

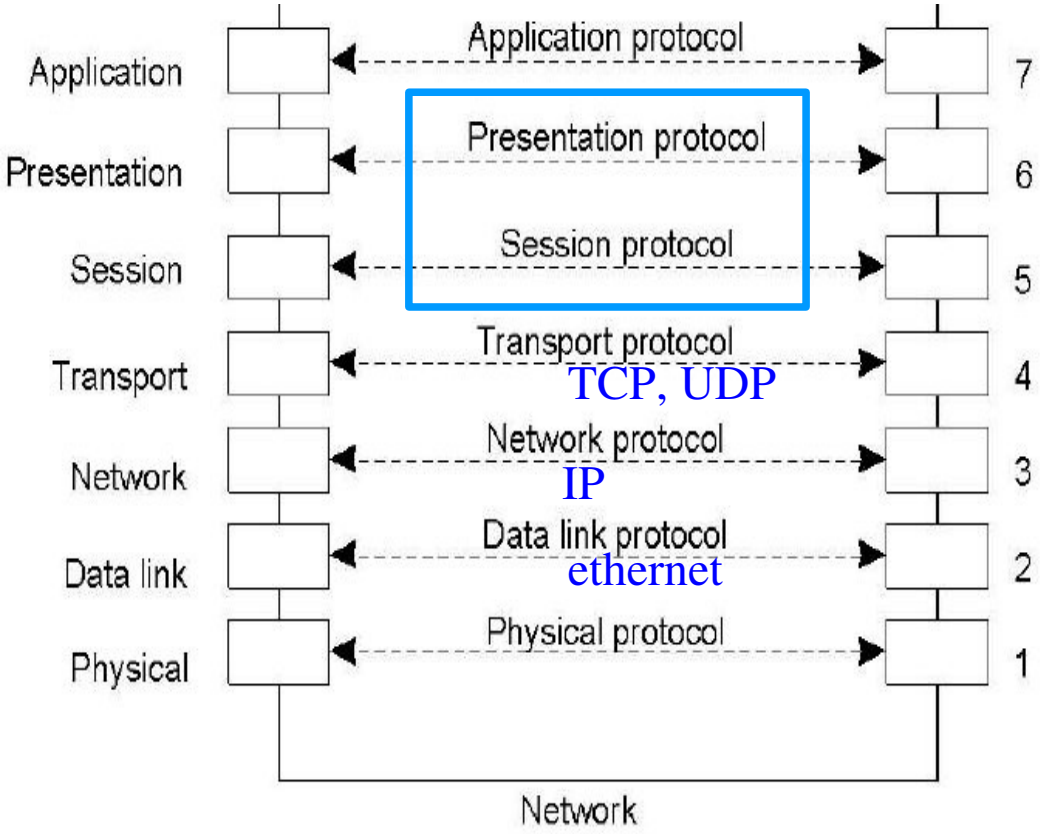
- necessidade de um processo executar uma operação noutro processo

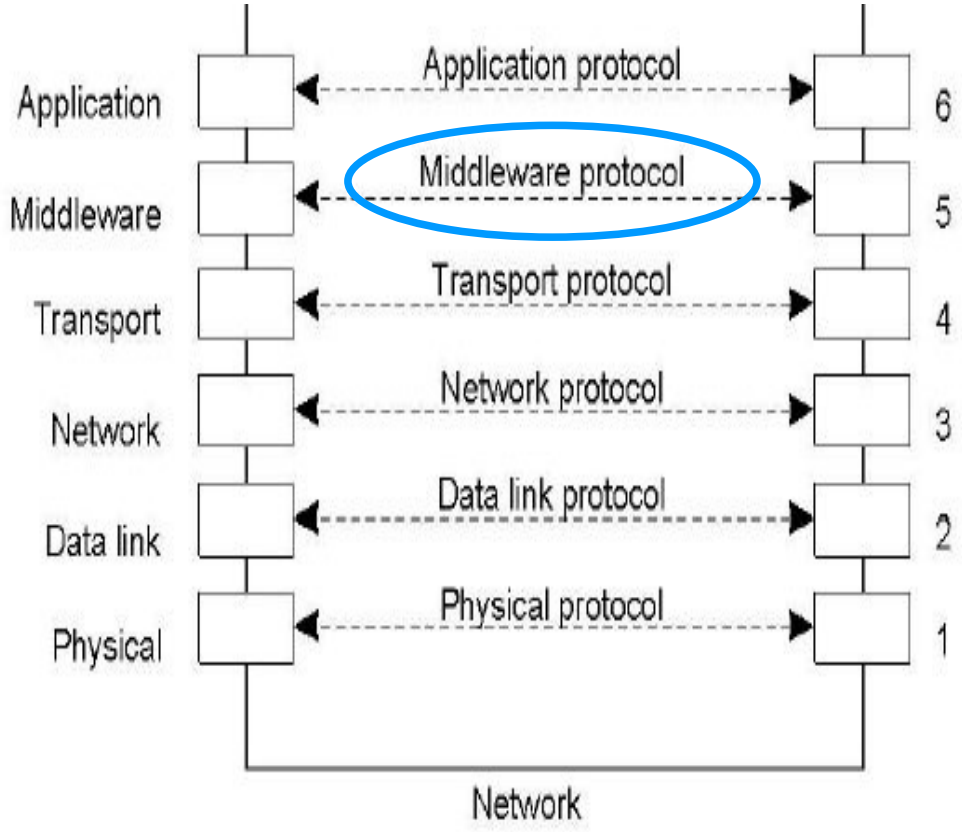
  MODELOS:
  - *Remote Procedure Call* (RPC)
    - programa cliente executa um procedimento do lado do servidor
  - *Remote Method Invocation* (RMI) *(JAVA RMI, CORBA)*
    - OO
    - um objeto pode invocar métodos de outro objeto num processo diferente
  - Modelo baseado em eventos
    - OO
    - objetos recebem notificações de eventos que acontecem num outro objeto em que estão interessados

Camada superior do *Middleware*



Modelo OSI e o Middleware





Abstração Middleware

- Transparência face à localização
  - o processo cliente não sabe o endereço do servidor
- Independência dos protocolos de comunicação
  - o Request-Reply pode ser implementado sobre TCP ou UDP
- Independência do Hardware
  - normas para Representação Externa de Dados escondem diferenças
- Independência dos SO
  - as abstrações de nível superior fornecidas pelo Middleware não dependem do Sistema Operativo
- Utilização de diferentes linguagens de programação (no caso de CORBA)
  - CORBA: cliente numa linguagem pode invocar métodos em servidores escritos noutra linguagem – graças à CORBA IDL

Interfaces

- Interface
  - forma de controlo sobre a comunicação entre módulos ou processos
  - especifica um formato para as mensagens

- A interface de um módulo especifica os métodos e variáveis acessíveis para outros módulos (e o restante em cada módulo fica escondido do exterior)

- Tipos de Interface:
  - em RPC: *Service Interface (em modelos cliente-servidor)*
    - descrição dos procedimentos disponíveis e respectivos argumentos (input/output)
    - Não se podem passar apontadores como argumentos
  - em RMI: *Remote Interface*
    - *métodos de um objeto disponíveis para Inv. Remota*
    - *podem passar-se referências para objetos remotos (diferente de pointers)*

IDL

- *Interface Definition Language* (IDL)
  - linguagem de programação adequada para definir interfaces

- Java RMI
  - todas as aplicações programadas na mesma linguagem: **Java**

- Sistemas baseados em várias linguagens de programação

(ex: CORBA)

  - requerem uma IDL que forneça a notação para as interfaces que poderão ser usadas pelas diferentes aplicações

Exemplo de CORBA IDL

Em CORBA IDL cada argumento é marcado como “de entrada”, “de saída” ou “de entrada e saída”

```
// In file Person.idl
struct Person {
        string name;
        string place;
        long year;
} ;
interface PersonList {
        readonly attribute string listname;
        void addPerson(in Person p) ;
        void getPerson(in string name, out Person p);
        long number();
};
```

Object Model (1)

- Programa OO
  - conjunto de objetos, cada um contendo dados e métodos
  - os objetos comunicam via invocação de métodos, passando argumentos e recebendo o resultado
  - algumas linguagens (Java,C++) permitem acesso direto a algumas variáveis (num SD os dados devem ser acedidos apenas pelos métodos)
- Os objetos têm uma referência
- Interfaces
  - declaração de métodos (argumentos, retorno e excepções) sem especificar a implementação
- Ações: iniciadas pelas invocações de métodos (podem alterar estado)
- Excepções: situação anómala
- *Garbage Collection*: libertar espaço ocupado por objetos quando eles já não são necessários

Objetos Distribuídos

- Exemplo: arquitetura cliente-servidor
  - cliente envia um pedido ao servidor com o método que deseja invocar sobre um objeto pertencente ao servidor. O resultado é-lhe devolvido.

- Estado do Objeto: definido pelos valores das suas variáveis de instância

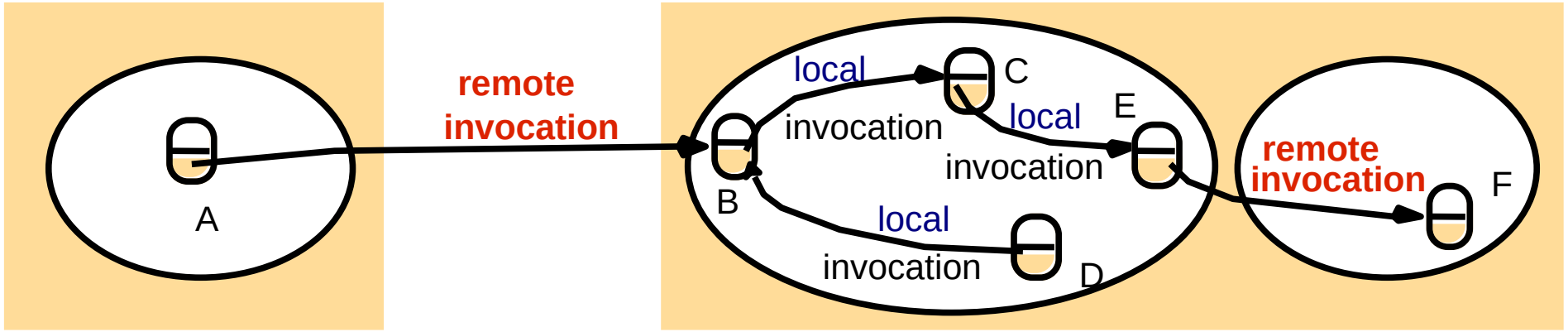
- quando o objeto pertence a um processo diferente, o seu estado só pode ser consultado/alterado através de métodos desse objeto
  - vantagem: facilita o mecanismo de proteção contra acessos incorretos (simultâneos/perigosos)

Object Model Distribuído

- processo
  - tem um conjunto de objetos, alguns dos quais recebem invocações locais e remotas, outros apenas recebem invocações locais
- Objeto Remoto
  - aquele que pode receber invocações remotas
- Referência Remota do Objeto
  - necessária para a invocação remota sobre o OR
  - identificação única no SD para aquele objeto
  - pode ser passada como argumento (Java)
- Interface Remota
  - especificação dos métodos que podem ser invocados remotamente

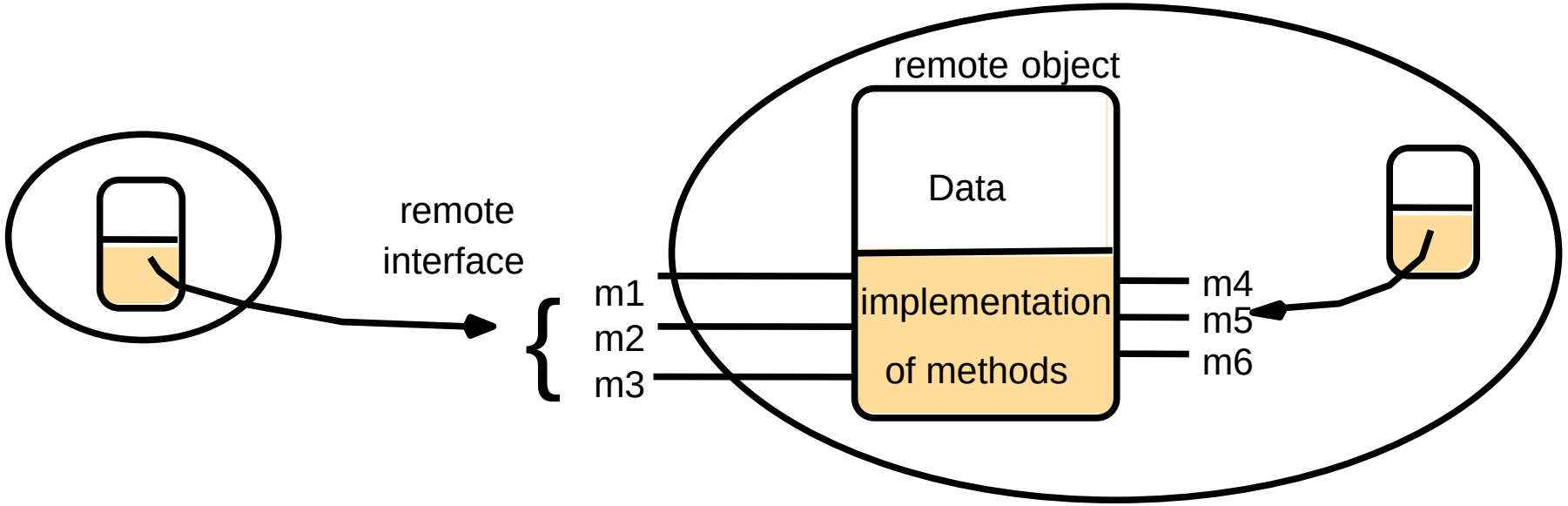
Invocação: local vs remota

- Invocação é Remota quando incide sobre um objeto de **outro** processo (que pode ou não estar na mesma máquina)



Object Model Distribuído

- Interface Remota



objetos do mesmo processo: podem invocar m1, ..., m4..., m6
objetos noutro processo: podem apenas invocar os métodos da interface remota m1,..., m3

Object Model Distribuído

- Ações
  - iniciadas pela invocação* de métodos
    * - que agora pode ser remota e desencadear outras invocações remotas em cadeia
- *Garbage Collection*
  - a nível distribuído, é necessário considerar as referências remotas para o objeto
- Exceções
  - podem surgir da mesma forma que na invocação local
  - Java: `RemoteException`

Detalhes na conceção da abstração RMI

- Semântica na invocação em RMI

- garantias de *doOperation* no protocolo RR:
  - reenvio do pedido
    - reenviar a mensagem com o pedido para o servidor até a reposta chegar ou se detectar que o servidor está com problemas
  - filtragem de duplicados
    - decidir se o duplicado deve ser processado para reenvio ou ignorado
  - retransmissão de resultados
    - através de um histórico de resultados para evitar uma nova execução da operação

Semântica da Invocação

medidas			***semântica de invocação***	
reenvio do pedido	*filtragem de duplicados*	*executar de novo ou retransmitir resultado do histórico*		
Não	Não aplicável	Não aplicável	*Maybe*	
Sim	Não	executar de novo	*At-least-once*	Sun RPC
Sim	Sim	retransmitir do histórico	*At-most-once*	Java RMI CORBA

- Possíveis Falhas:
  - *Maybe*: omissão, crash
  - *At-least-once*: crash, arbitrárias (retransmissão do pedido, reexecução)
  - *At-most-once*: falhas evitam-se com os mecanismos de tolerância a falhas

Implementação de RMI: **componentes**

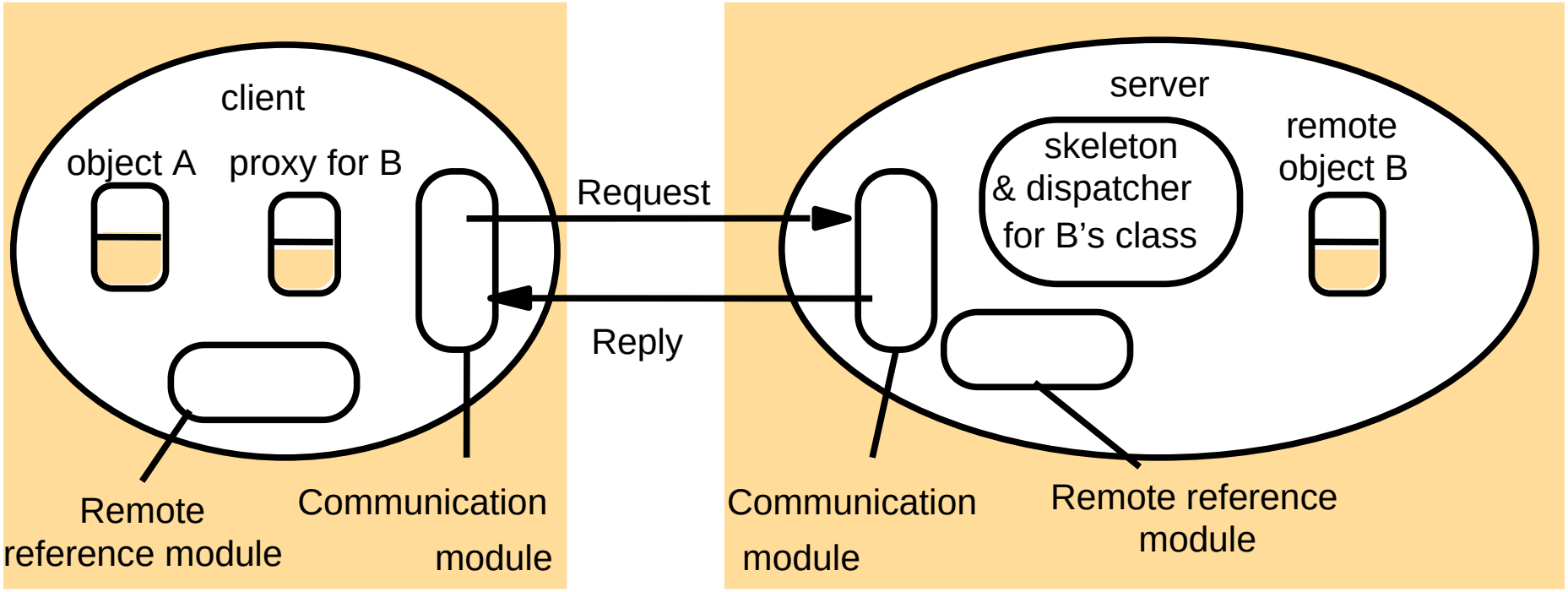
- módulo de comunicação
  - protocolo RR, semântica da invocação, do lado do servidor escolhe o *dispatcher*
- módulo de referências remotas
  - tradução entre referências remotas e locais, criação de refs. remotas
- Software RMI
  - **Proxy**
    - torna a invocação remota transparente para o cliente
    - *marshalling* de argumentos, *unmarshalling* de resultado da invocação
    - existe um proxy para cada objeto remoto que um processo referencie
    - implementa os métodos da interface remota do objeto, mas cada método faz *marshall* da referência do objeto, *methodId*, e argumentos, aguardando a resposta para o *unmarshall*

Implementação de RMI: componentes

- Software RMI - *continuação*
  - **Dispatcher**
    - um para cada classe de objeto remoto, no servidor
    - recebe a mensagem e pelo *methodId* seleciona o método apropriado no Skeleton
  - **Skeleton**
    - um por cada classe que representa um objeto remoto, no servidor
    - implementa os métodos na interface remota, mas efetuando *unmarshall* a argumentos no pedido, invocando o método no objeto remoto (localmente) e devolvendo o *marshall* do resultado e eventual excepção na reposta ao proxy

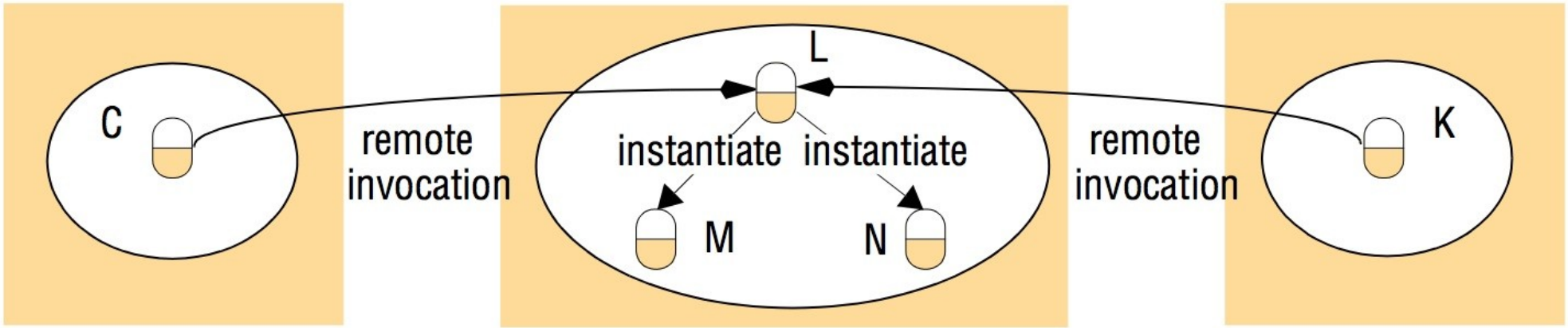
- Orbix CORBA e Java RMI: proxy, dispatcher e skeleton são gerados automaticamente

RMI: Proxy e Skeleton



Objetos Remotos: podem ser criados *on-demand*

- novas instâncias ou ativações



Objeto Remoto pode estar em estado:

- Ativo: disponível para invocação a qualquer momento
  - permanece junto ao processo a que pertence
- Passivo: não disponível, mas pode ativar-se
  - inclui os seus métodos e uma representação serializada/*marshalled* do seu estado

Servidor, Cliente, Binder: quem tem e faz o quê?

- Servidor
  - classes para dispatcher e skeleton
  - classes com implementação para todos os objetos remotos *Servant ou Impl*
  - zona de inicialização:
    - criar pelo menos um objeto remoto e inicializá-lo
    - registar o objeto no *binder*
  - para evitar demoras, cada invocação remota é tratada numa nova Thread
- Cliente
  - classes dos proxies dos objetos remotos usados
  - lookup no *binder* para obter a referência remota do objeto
- Binder
  - serviço que guarda pares (nome, referência remota do objeto)
  - “servidor de nomes”

Garbage Collection Distribuído

- Se um objeto tem uma referência local ou remota no conjunto de objetos distribuídos, então deve continuar a existir.

- Cooperação com o GC local:
  - cada servidor mantém uma lista com o conjunto de processos com referências para os seus objetos
  - quando um cliente cria um Proxy para um objeto, é adicionado ao conjunto de processos com referências para aquele objeto
  - quando o GC do cliente deteta que o Proxy do objeto já não necessário/referido, envia uma mensagem ao servidor (removeRef(O)) e elimina o proxy. O servidor remove o processo da lista.
  - Quando a lista estiver vazia, o GC do servidor recupera o espaço do objeto, excepto se existirem referências locais.

Garbage Collection Distribuído

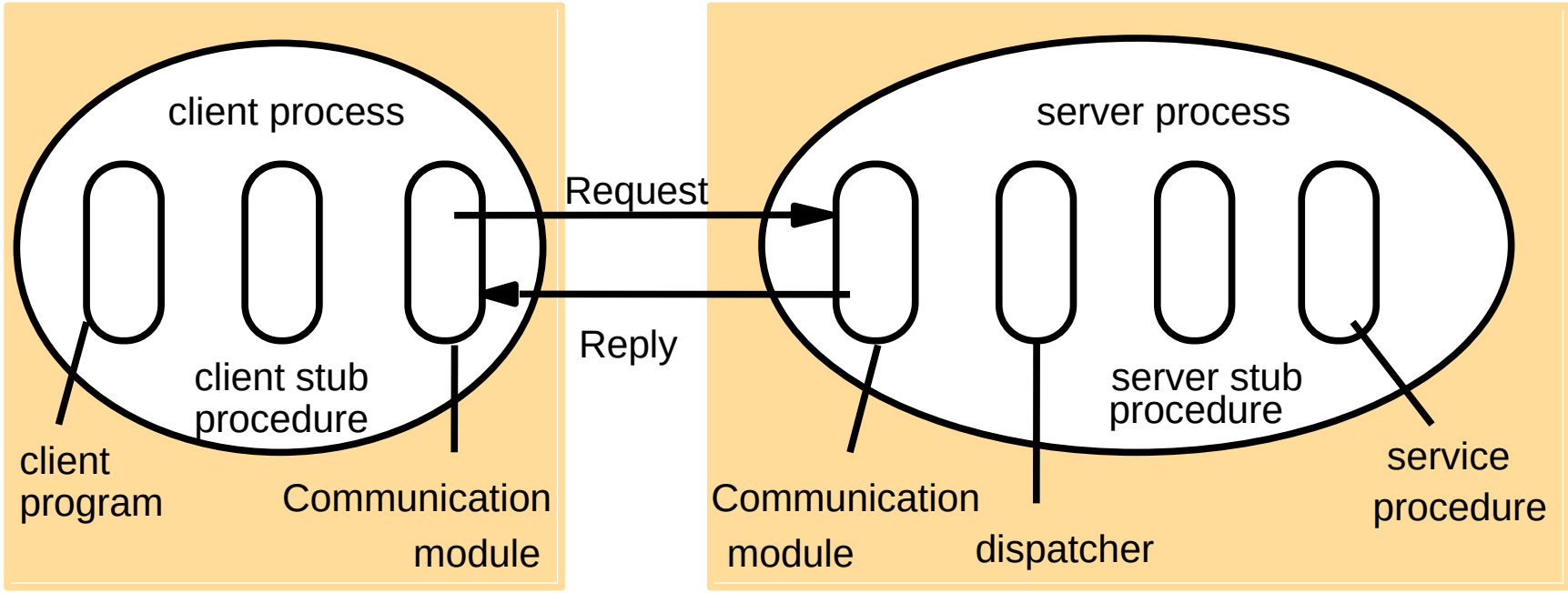
- protocolo RR
- semântica de invocação *At-most-once*

- **Java Distributed GC**
  - tolera falhas no cliente
    - o servidor (com objetos remotos) atribui um intervalo de tempo (*lease*) ao cliente
    - a contagem é válida até o tempo expirar ou o cliente pedir *removeRef()*
  - o cliente é responsável por renovar periodicamente o seu *lease*
    - e assim ser contabilizado na lista de processos com referências para os objetos remotos

Abstração RPC

- *service interface*
- semânticas de invocação usualmente implementadas:
  - *at-least-once*
  - *at-most-once*
- sobre o protocolo RR
- Software RPC
  - semelhante ao anterior, mas sem Módulo de Referências Remotas, uma vez que não há a noção de objetos

RPC: stub de cliente e de servidor, dispatcher



Abstração RPC

- Stub de Cliente
  - como o Proxy: tem métodos locais que fazem marshalling, (unmarshalling) no pedido (resposta) para o servidor
- Server Stub
  - como o Skeleton (unmarshalling, marshalling)
- Dispatcher
  - recebe o pedido do cliente e escolhe o stub adequado

- Existe um *Stub Procedure* por procedimento na *Service Interface*

- Stubs e Dispatcher
  - gerados automaticamente a partir da interface do serviço

Abstração RPC

- Sun RPC (aka ONC RPC)
  - sobre TCP ou UDP
  - *interface language*: XDR (e**x**ternal **d**ata **r**epresentation)

- SunRPC / XDR
  - não permite dar o nome à interface
  - números de programa e versão
  - o procedimento a invocar é identificado por um nº, no pedido
  - é permitido um único parâmetro de input *(vários: uma estrutura)*
  - o output do procedimento sai num único resultado/valor

Sun RPC

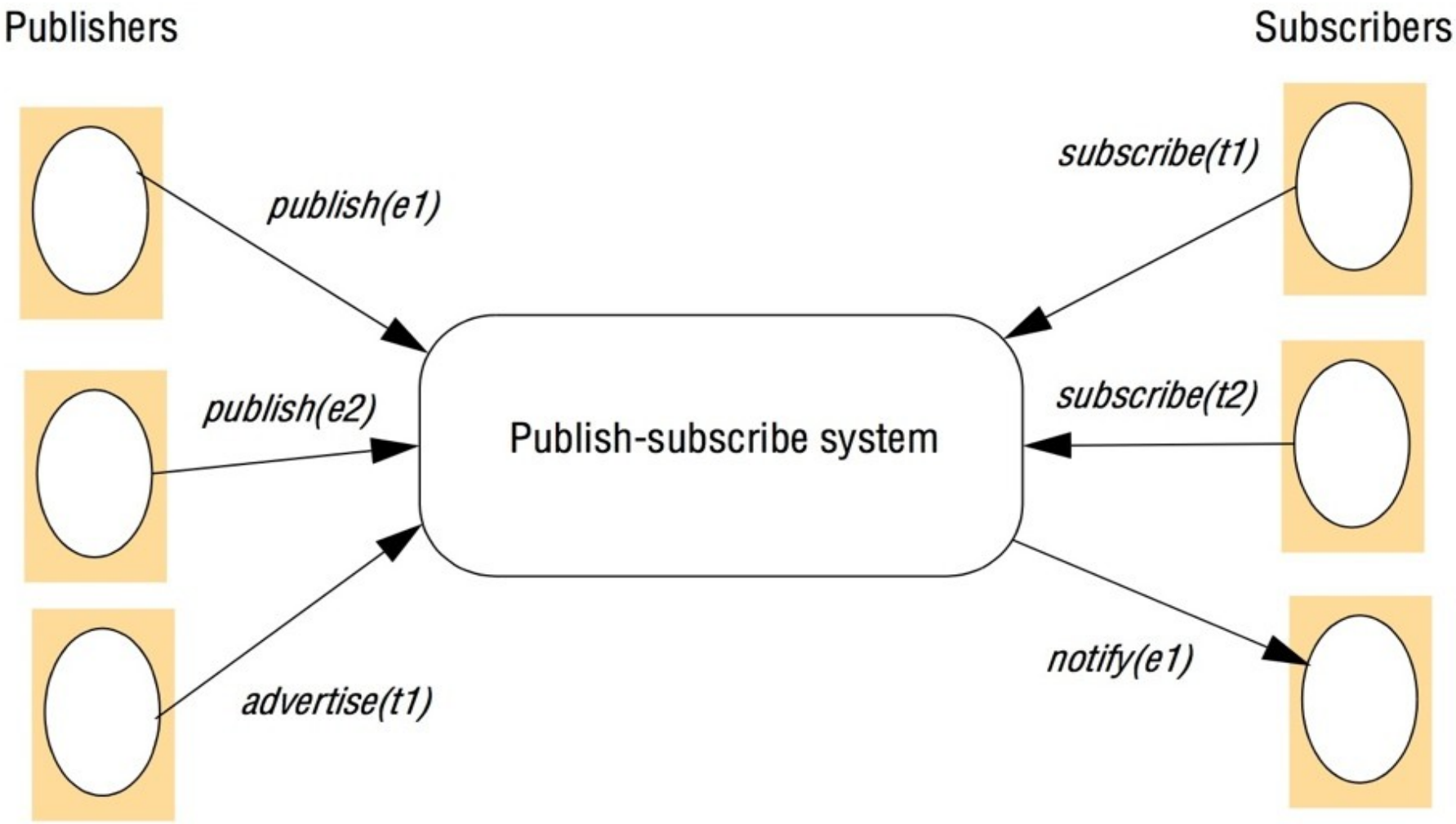
- XDR
  - permite definir constantes, typedefs*, estruturas*, enumerações*, unions e programas. (* com a sintaxe da linguagem C)
- Binding: *port mapper*
  - ao iniciar, o servidor regista os nº de programa, versão e porto no port mapper
- Autenticação
  - mensagens Sun RPC incluem campos adicionais dedicados à autenticação de cliente e servidor

Eventos e Notificações

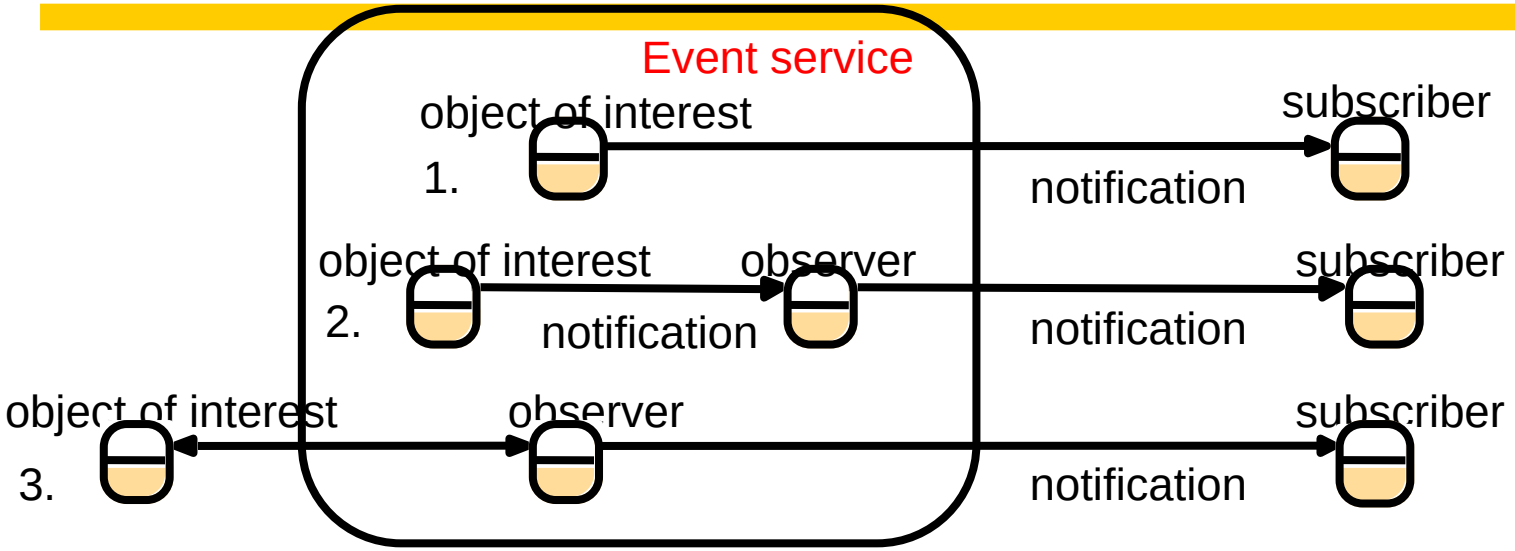
- Evento: provoca alteração do estado de um objeto
- Notificações: objetos que representam eventos

- Características de SD baseados em Eventos
  - usam paradigma *publish-subscribe*
    - os eventos são publicados: ficam disponíveis para observação por outros objetos
    - os objetos que querem receber notificações subscrevem o tipo de eventos que lhes interessa
  - heterogéneos
    - os componentes têm conhecimento dos outros componentes (a quem subscrevem ou para quem notificam)
  - assíncronos
    - notificações enviadas de modo assíncrono

Paradigma *Publish-Subscribe*



Arquitetura para um SD com notificação de eventos



Subscriber: objeto interessado em certo tipo de eventos de outro objeto

Event service: regista os eventos publicados e os interesses dos subscritores

Object of Interest: objeto que sofre alteração de estado em resultado de operações/invocações

Observer: intermediário entre o *Object of Interest* e o *Subscriber*, tem as funções:

- Forwarding: pode haver muitos *subscribers,* libertando o OoI
- Filtragem de Notificações
- Padrões de eventos: notificação especial na ocorrência de um padrão de eventos
- Notification Mailbox: guarda as notificações até o subscriber estar pronto para as receber

Notificação de Eventos em Java

- Jini
  - permite a um *subscriber* de uma JVM subscrever e receber notificações de eventos num *Object of Interest* localizado numa JVM diferente (eventualmente noutra máquina)
  - utiliza Java RMI no envio de notificações

Notificação de Eventos em Java: Jini

- *EventGenerator*: interface com o método *register().* É implementada pelos objetos cujos eventos geram notificações.
- *RemoteEvent*: classe que descreve o tipo de evento, o gerador do evento, nº de sequência para aquele tipo de evento e ainda uma representação serializada do evento
- *RemoteEventListener*: interface com o método *notify()*. Implementada pelos *subscribers* e *third-party agents* para recebem as notificações (pela invocação remota deste método)
- *Third-party agents*: podem existir entre o Objecto de Interesse e o *subscriber.* Equivalentes aos observers.

Java RMI

- permite a invocação remota com uma sintaxe idêntica à invocação local
- o cliente tem de *tratar* as `RemoteExceptions`

- Interface Remota: definida em Java
- cuidado: prever o comportamento do objeto remoto com a concorrência

- Exemplo: servidor guarda registo de um quadro com figuras
  - objetos remotos: Shape, ShapeList – *ver livro*

Java RMI: interfaces remotas *Shape* e *ShapeList*

```
import java.rmi.*;
import java.util.Vector;

public interface Shape extends Remote {
    int getVersion() throws RemoteException;
    GraphicalObject  getAllState() throws RemoteException;          1
}

public interface ShapeList extends Remote {
    Shape newShape(GraphicalObject g) throws RemoteException; 2
    Vector allShapes() throws RemoteException;
    int getVersion() throws RemoteException;
}
```

Java RMI

- Interface Remota
  - interface remota: estender a interface *java.rmi.Remote*
  - declaração de métodos: *throws RemoteException (e eventualmente outras específicas da aplicação)*

- Parâmetros e Resultado
  - argumentos do método da interface: parâmetros de input
  - retorno do método: único parâmetro de output
  - as classes dos argumentos e retorno do método devem ser serializáveis
    - (todos os tipos primitivos e objetos remotos são serializáveis)
  - passagem de parâmetros *(ver linhas 1 e 2)*
    - objetos remotos: por referência (remota)
    - objetos não remotos: passagem por valor

Java RMI

- Download de Classes
  - em Java as classes podem passar de uma JVM para outra
  - se o cliente não tem a classe do tipo de retorno no método, o código é obtido do seu interlocutor automaticamente
    - Vantagem: transparência
    - Segurança: precisamos de regras para aceitar código – RMISecurityManager

- RMI registry
  - binder, serviço de nomes
  - mapeia nomes* e referências remotas
    - `* rmi//hostname:port/objectName`
  - ver `java.rmi.Naming`

Java RMI Registry: classe Naming

void rebind (String name, Remote obj)

método usado por um servidor para adicionar uma entrada, fazendo o registo de um par (nome,referência para o objeto remoto).

void bind (String name, Remote obj)

método usado por um servidor para adicionar uma entrada, fazendo o registo de um par (nome,referência para o objeto remoto).
Se o nome já existir é lançada uma exceção.

void unbind (String name, Remote obj)

método para eliminar uma entrada

Remote lookup(String name)

método invocado por clientes para obter uma referência para um objeto remoto a partir do seu nome

String [] list()

devolve um array de Strings com os nomes registados neste serviço

Java RMI: um servidor

```
import java.rmi.*;

public class ShapeListServer{
     public static void main(String args[]){
          System.setSecurityManager(new RMISecurityManager());
           try{
               ShapeList aShapeList = new ShapeListServant();                    1
               Naming.rebind("Shape List", aShapeList );                         2
               System.out.println("ShapeList server ready");
               }catch(Exception e) {
               System.out.println("ShapeList server main " + e.getMessage());}
     }
}
```

Java RMI: Servant *(ServiceImpl – classe do objecto remoto)*

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import java.util.Vector;

public class ShapeListServant extends UnicastRemoteObject implements ShapeList {
      private Vector theList;                    // contains the list of Shapes        1
      private int version;
     public ShapeListServant() throws RemoteException{...}
     public Shape newShape(GraphicalObject g) throws RemoteException {        2
          version++;
               Shape s = new ShapeServant( g, version);                        3
               theList.addElement(s);
               return s;
     }
     public  Vector allShapes()throws RemoteException{...}
     public int getVersion() throws RemoteException { ... }
}
```

implementação dos métodos sem os detalhes de comunicação.
Só funcionalidade.

Java RMI: um cliente

```
import java.rmi.*;
import java.rmi.server.*;
import java.util.Vector;

public class ShapeListClient{
   public static void main(String args[]){
     System.setSecurityManager(new RMISecurityManager());
     ShapeList aShapeList = null;
     try{
          aShapeList  = (ShapeList) Naming.lookup("//bruno.ShapeList") ;          1
          Vector sList = aShapeList.allShapes();                                  2
     } catch(RemoteException e) {System.out.println(e.getMessage());
     }catch(Exception e) {System.out.println("Client: " + e.getMessage());}
   }
}
```

Callbacks

- há situações em que é o servidor que toma a iniciativa de contactar o cliente devido a um evento
  - evita consultas desnecessárias por parte do cliente
  - implementável via RMI
    - cliente cria um objeto* remoto com o método que o servidor há-de invocar

      * designado *callback object*
    - cada cliente interessado informa o servidor do seu *callback object*
    - a cada evento, o servidor invoca o método junto daqueles clientes

Java IDL

- conectividade e interoperabilidade com ambiente CORBA
- permite desenvolver uma aplicação em Java, com uma interface definida em CORBA IDL, e inseri-la num ambiente CORBA
  - (com outras aplicações possivelmente noutras linguagens)

CORBA *(common object request broker arquitecture)*

- permitir que objetos distribuídos e implementados em diferentes linguagens de programação possam comunicar

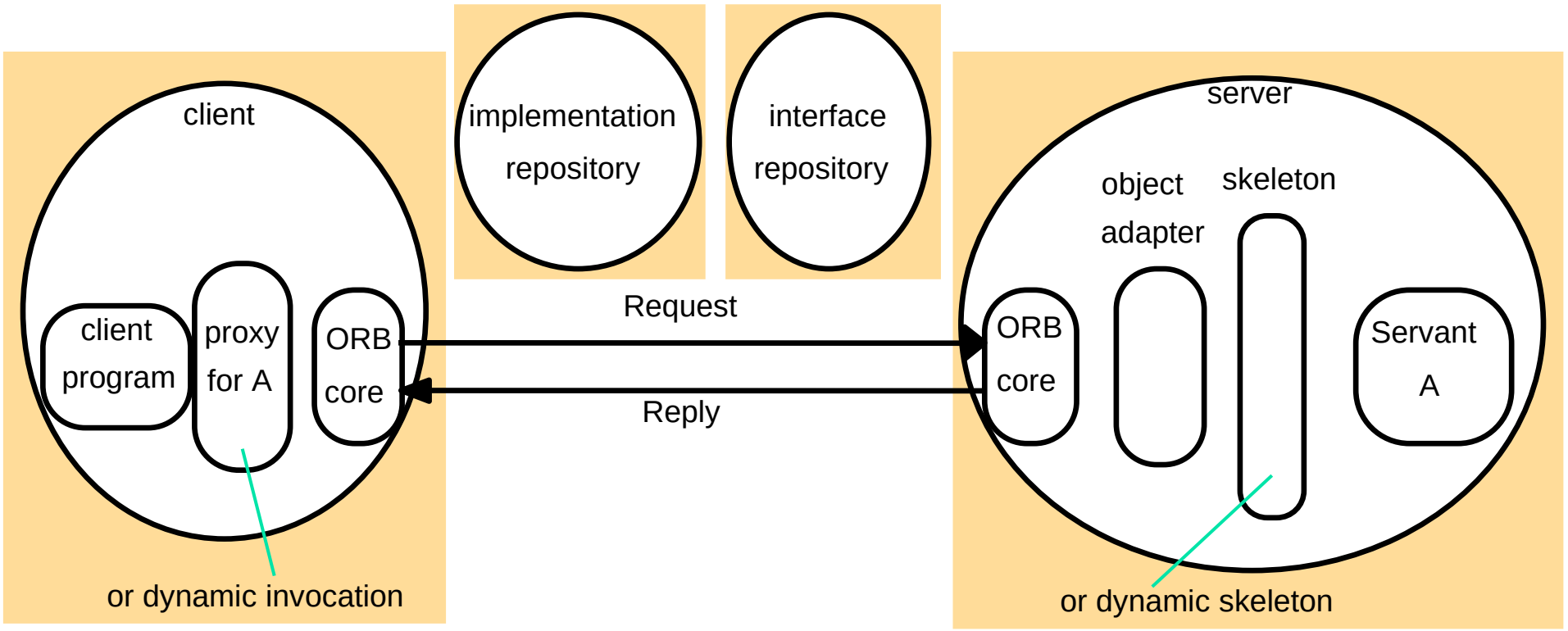
- assenta na metáfora do *object request broker*
  - ajuda um cliente a invocar um método num objeto
    - localizar o objeto
    - ativar o objeto, se necessário
    - comunicar o pedido ao objeto, que por sua vez presta o serviço
    - devolver a resposta ao cliente

CORBA

- Componentes principais:
  - IDL
  - arquitetura
  - formato para representação externa de dados CDR e formato das mensagens a trocar no protocolo RR, definidos na norma GIOP (*General InterORB Protocol*)
  - referência de objetos remotos, definida pela norma IIOP
    (Internet Inter-Orb Protocol)
- Passagem de parâmetros:
  - tipo definido pela IDL: é devolvida uma referência remota
  - tipos primitivos e outros tipos compostos: copiados e passados por valor
- Semântica de invocação
  - *at-most-once*

Arquitetura CORBA

- Semelhante à arquitetura Java RMI... com alguns componentes novos



Arquitetura CORBA

- Object Adapter: faz a ponte entre os objetos CORBA na interface IDL e os objetos do servant com a interface da linguagem de programação
- ORB Core: semelhante ao módulo de comunicação de Java RMI
- Implementation repository: mapeia nomes de *object adapters* em pathnames de ficheiros com a implementação dos objetos, bem como o hostname e porto do servidor em que se encontram
- Interface repository: disponibiliza aos clientes a informação sobre interfaces IDL registadas
- skeleton: gerado pelo compilador de IDL, na linguagem do servidor. Faz *unmarshall* e *marshall* de argumentos e resultado ou exceção, respetivamente
- client stub/proxy: gerado na linguagem do cliente, pelo compilador de IDL. Faz *marshall* e *unmarshall* de argumentos e resultado, respetivamente