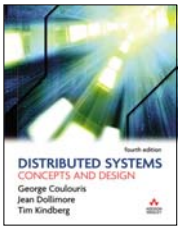
Material baseado no livro *Distributed Systems: Concepts and Design*, 4th Edition, Addison-Wesley, 2005.






# 5 – Objetos Distribuídos

**Agenda:**

- **Fundamentos de objetos distribuídos**
- **Comunicação entre objetos distribuídos**
- **Notificação distribuída de eventos**
- **Exemplo utilizando Java RMI**

## Fundamentos de objetos distribuídos

- Modelos de programação
- Middleware
- Interfaces e IDLs

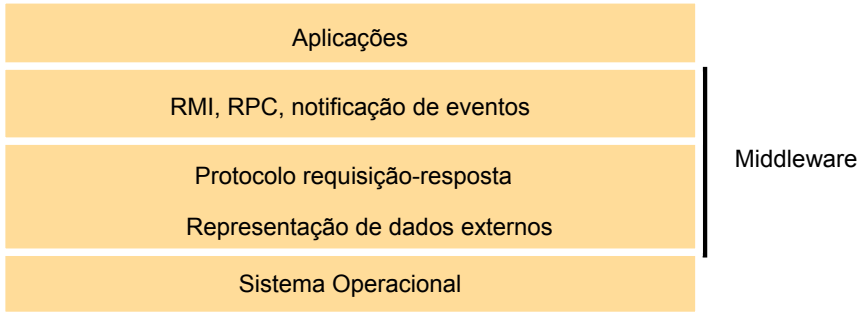
## Modelos de programação

- Chamada procedimentos remotos (*RPC*)
  - Extensão da programação procedimental convencional
  - Permite que programas clientes invoquem procedimentos em programas servidores de outros processos, possivelmente localizados em máquinas remotas
- Invocação métodos remotos (*RMI*)
  - Extensão da programação orientada a objetos
  - Permite que objetos clientes invoquem métodos em objetos servidores de outros processos, possivelmente localizados em máquinas remotas
- Notificação distribuída de eventos (*publish/subscribe*)
  - Extensão da programação orientada a eventos
  - Permite que objetos recebam notificações de eventos de seu interesse ocorridos em outros objetos, possivelmente localizados em máquinas remotas




## Middleware

- Camada de software que oferece um modelo de programação de alto nível (acima dos recursos básicos disponíveis no S.O.) para a implementação de processos e troca de mensagens entre eles



## Propriedades de middleware

- Transparência de localização
  - RPC: programa cliente não distingue se procedimento chamado é local ou remoto
  - RMI: objeto cliente não distingue se objeto invocado é local ou remoto, nem sabe a sua localização
  - Notificação de eventos: objetos que geram eventos e objetos que recebem notificações sobre esses eventos não sabem da localização uns dos outros
- Protocolo de comunicação
  - Independente dos protocolos da rede (ex.: protocolo requisição-resposta implementado sobre UDP ou TCP)
- Heterogeneidade
  - Tolerância a diferenças em termos de hardware, S.O. e linguagem/ambiente de programação




## Interfaces

- Especificam os recursos (procedimentos, tipos, variáveis, etc) definidos pelos módulos de um programa que podem ser acessados por outros módulos
  - Cada módulo é implementado de modo a esconder seus recursos internos, com exceção daqueles disponibilizados através de sua interface pública
  - Se a interface não for alterada, a implementação de um módulo pode variar independentemente do seu uso pelos outros módulos
- Restrições para interfaces em sistemas distribuídos:
  - Impossibilidade de acesso direto ao estado (variáveis de instância) de módulos remotos
    - Acesso indireto via operações *get/set*
  - Mecanismo de passagem de parâmetros limitado a variáveis de E/S
    - Impossível passar ponteiros como parâmetros! Por quê?

## Modelos de interface

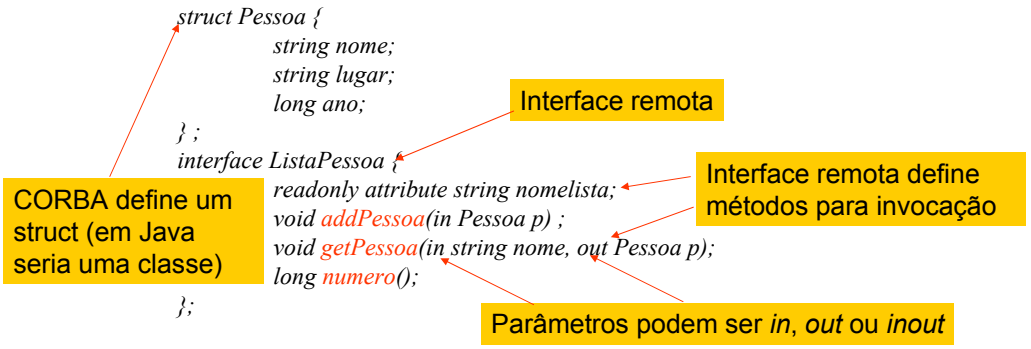
- Interface de serviço (RPC)
  - Especifica o conjunto de procedimentos oferecidos por um servidor, definindo os tipos dos argumentos de entrada e saída de cada procedimento
  - Não permite a passagem de procedimentos como argumento de outros procedimentos
- Interface remota (RMI)
  - Especifica o conjunto de métodos de um objeto que estão disponíveis para invocação por outros objetos em outros processos, definindo os argumentos de entrada e saída e as exceções de cada método
  - Permite a passagem de objetos locais, e de referências para objetos remotos, tanto como argumento quanto como resultado dos métodos de outros objetos




## IDLs: Linguagens para a definição de interfaces

- Notação específica para a descrição de interfaces
  - Define os métodos de um serviço, e os parâmetros de entrada e saída, as exceções, e os resultados desses métodos
  - Permite mapear os elementos da interface para os elementos correspondentes na linguagem de programação escolhida, tanto no lado do cliente como no lado do servidor
  - Desnecessária se a mesma linguagem é utilizada nos dois lados
- Exemplos:
  - CORBA IDL
  - DCOM IDL
  - WSDL
  - Java IDL (?)

# Exemplo em CORBA IDL







# Agenda

- Fundamentos de objetos distribuídos
- Comunicação entre objetos distribuídos
- Notificação distribuída de eventos
- Exemplo de aplicação utilizando Java RMI

## Comunicação entre objetos distribuídos

- Modelo de objetos
- Objetos distribuídos
- Questões de projeto
  - Semântica de invocação
  - Transparência
- Implementação
- Coleta de lixo distribuída




## Modelo de objetos

- Define as características dos objetos e suas formas de interação
- Características de interesse:
  - Referências
    - Mecanismo através do qual um objeto (alvo) pode ser acessado por outros objetos
    - Podem ser atribuídas a variáveis, passadas como argumentos, ou devolvidas como resultado de um método
  - Interfaces
    - Definição das assinaturas de um conjunto de métodos (tipos dos argumentos, valores de retorno, exceções) sem especificar sua implementação
    - Objeto oferece um interface se sua classe implementa os métodos definidos pela mesma
    - Podem ser usadas para declarar o tipo de objetos locais ou os argumentos e o valor de retorno de um método

## Modelo de objetos

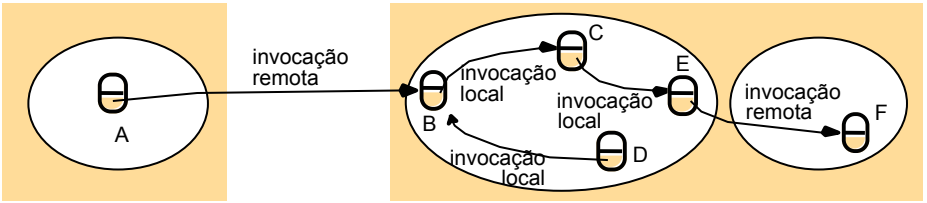
- Características de interesse (cont.):
  - Ações
    - Iniciada quando um objeto invoca um método em outro objeto
    - Objeto alvo executa o método invocado e retorna o controle para o objeto que o invocou, possivelmente devolvendo um valor de retorno
    - Execução do método pode alterar o estado do objeto alvo, instanciar um novo objeto, e ainda desencadear a invocação de outros métodos, possivelmente de outros objetos
  - Exceções
    - Problemas que podem acontecer durante a execução de um método
    - Para cada método é especificada uma lista de exceções, que devem ser tratadas explicitamente pelos usuários do método (cláusulas *throws*, *try* e *catch* em Java)
  - Coleta de lixo
    - Mecanismo para liberar o espaço de memória ocupado pelos objetos quando estes não são mais necessários (potencial fonte de erros se implementado pelo programador da aplicação; transparente em Java)




## Objetos distribuídos

- Objetos de um programa fisicamente distribuídos em múltiplos processos, possivelmente executando em diferentes computadores
- Organização baseada em diferentes modelos de arquitetura:
  - Cliente-servidor
    - Objetos são gerenciados por um servidor, que invoca localmente os métodos requisitados pelos clientes, devolvendo-lhes os resultados obtidos
    - Objetos requisitados podem tornar-se clientes de objetos mantidos em outros servidores
  - Replicação e migração
    - Objetos podem ser replicados ou migrar para outros computadores visando obter maior desempenho e disponibilidade
- Encapsulamento do estado dos objetos
  - Independência do formato de representação dos dados
  - Uso de mecanismos de proteção contra acessos concorrentes indevidos

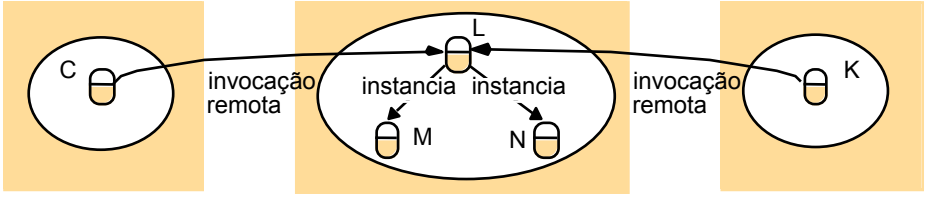
# Modelo de objetos distribuídos




- Cada processo contém um ou mais objetos, alguns dos quais podem ser invocados remotamente (B e F) e outros apenas localmente (C, D e E)
- Objetos precisam obter *referências remotas* para os objetos remotos localizados em outros processos, para poderem invocar os métodos declarados em suas *interfaces remotas*
- A invocação de um método remoto pode gerar *exceções remotas* (em geral, causadas por falhas de comunicação), além daquelas que podem surgir durante a execução do próprio método

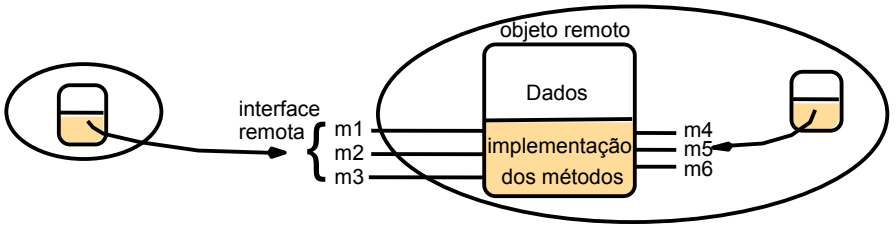



# Referências remotas




- Uma referência remota é um identificador que pode ser usado em todo o sistema distribuído para referenciar um único objeto remoto particular (representação interna discutida mais adiante)
  - Especifica o objeto remoto a ser invocado remotamente pela aplicação através da camada de middleware
  - Pode ser passada como argumento ou parâmetro de retorno de invocações a métodos remotos

## Interfaces remotas




- A classe de um objeto remoto implementa os métodos definidos na sua *interface remota* (extensão da interface *Remote,* em Java RMI)
- Objetos em outros processos apenas podem invocar os métodos de um objeto remoto que estejam definidos na sua interface remota
- Objetos co-localizados junto com o objeto remoto podem invocar os métodos de sua interface remota bem como outros métodos disponibilizados localmente pelo objeto




## Semântica de invocação

- Invocações locais são executadas exatamente uma vez
- Invocações remotas não conseguem obter essa mesma semântica. Por quê?
  - Solução: estender o protocolo requisição-resposta com medidas de tolerância a falhas

| *Medidas de tolerância a falhas* | | | *Semântica de invocação* |
|---|---|---|---|
| *Retransmitir requisição* | *Filtrar duplicações* | *Re-executar procedimento ou retransmitir resposta* | |
| Não | N.A. | N.A. | *Talvez* |
| Sim | Não | Re-executar procedimento | *No-mínimo-uma-vez* |
| Sim | Sim | Retransmitir resposta | *No-máximo-uma-vez* |

## Semântica de invocação: modelo de falha

- *Talvez* – se não há resposta, o cliente não consegue saber se o método foi executado ou não
  - Pode sofrer falhas de omissão se a mensagem de invocação ou de resultado for perdida
- *No-mínimo-uma-vez* – o cliente obtém um resultado (o método é executado pelo menos uma vez) ou uma exceção (nenhum resultado é obtido)
  - Pode sofrer falhas arbitrárias se a mensagem de invocação for retransmitida (exceto para operações idempotentes)
- *No-máximo-uma-vez* – o cliente obtém um resultado (o método é executado exatamente uma vez) ou uma exceção (o método não é executado nenhuma vez)
- Todos os três casos podem sofrer falhas de pane se o servidor que contém o objeto remoto falhar!




## Transparência

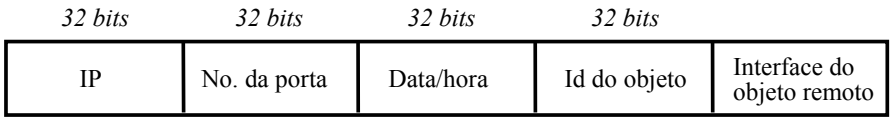
- Objetivo principal: fazer das invocações remotas tão "parecidas" quanto possível com as invocações locais
  - RPC implementa transparência quanto a empacotamento / desempacotamento de parâmetros e retransmissão de mensagens
  - RMI estende esse modelo implementando transparência quanto a localização e invocação de objetos remotos
- Na prática, transparência total é inviável ou mesmo indesejável
  - Latência para invocações remotas é consideravelmente maior do que para invocações locais (programador deve decidir quando usar)
  - Impossível distinguir entre falha da rede e falha do servidor remoto (tratamento da falha pode variar de aplicação para aplicação)
- Consenso atual:
  - Transparência somente do ponto de vista da sintaxe de invocação
  - Diferenças entre objetos locais e objetos remotos (semântica de invocação, exceções, etc) expressas explicitamente nas suas interfaces

## Implementação

- Tópicos discutidos:
  - Representação interna para referências remotas
  - Modelo de arquitetura
  - Criação, ativação e persistência de objetos remotos
  - Serviço de localização
  - Coleta de lixo distribuída




## Representação interna para referências remotas



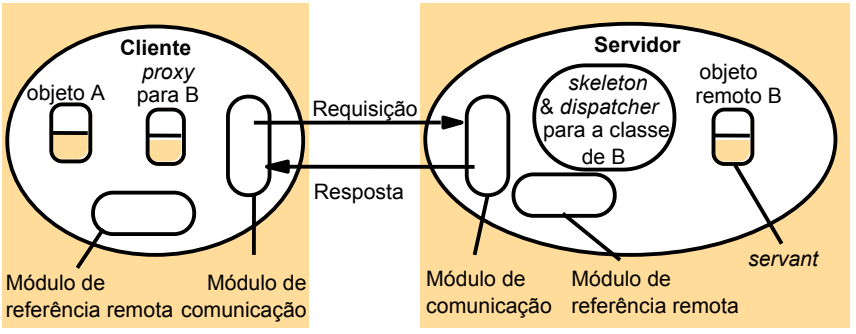

- Uma referência para um objeto remoto deve ser única com relação ao espaço e ao tempo, e não deve ser reusada após o objeto ter sido removido. Por quê?
  - Os primeiros dois campos indicam a localização do objeto (a menos que o mecanismo de RMI permita migração ou re-ativação de objetos para outros processos)
  - O quarto campo identifica o objeto dentro do processo servidor
  - O campo de interface indica os métodos que podem ser invocados no objeto
- O módulo de referência remota cria uma referência remota para um objeto local quando uma de suas referências é usada como argumento ou resultado de invocação envolvendo outro processo

## Modelo de arquitetura

- Principais componentes:
  - Módulo de comunicação
  - Módulo de referência remota
  - Camada RMI
  - *Proxy*
  - *Dispatcher*
  - *Skeleton*




## Modelo de arquitetura



## Componentes de arquitetura

- Módulo de comunicação
  - Implementa o protocolo requisição-reposta entre o cliente e o servidor de objetos, de acordo com a semântica de invocação especificada
- Módulo de referência remota
  - Mapeia referências locais para referências remotas e cria referências remotas para objetos locais
  - Utiliza uma *tabela de objetos remotos* para registrar a correspondência entre referências locais e referências remotas
- Camada RMI
  - Localizada entre os objetos da aplicação e os módulos de comunicação e de referência remota
  - Inclui um componente no lado do cliente (*Proxy*) e dois no lado do servidor (*Dispatcher* e *Skeleton*)




## Componentes da camada RMI

- *Proxy*
  - Torna a invocação remota transparente para o cliente
  - Empacota e envia os parâmetros de invocação para o servidor e desempacota o resultado recebido
  - Implementa a mesma interface do objeto remoto
- *Dispatcher*
  - Junto com o *skeleton*, é criado para cada classe que implementa um objeto remoto no servidor
  - Recebe as mensagens de requisição do módulo de comunicação e as repassa para os métodos apropriados no *skeleton*
- *Skeleton*
  - Desempacota os parâmetros recebidos na mensagem de requisição e invoca o método correspondente no objeto remoto (*servant*); empacota e envia o resultado ou exceção para o *proxy* de origem no cliente
  - Também implementa a mesma interface do objeto remoto

## Componentes da camada RMI

- Estratégias de implementação:
  - Geração automática do código dos componentes por um *compilador de interfaces*
    - Indicada quando a interface do objeto remoto é conhecida *a priori*
    - Melhor desempenho
  - Utilização de componentes genéricos através de uma interface de *invocação dinâmica*
    - Indicada quanto há a necessidade de invocar objetos cujas interfaces não são conhecidas em tempo de implementação (ex: *browser* de objetos)
    - Desempenho inferior aos componentes gerados automaticamente
  - Carregamento automático do código dos componentes a partir de um servidor remoto
    - Alternativa à invocação dinâmica (desempenho?)




## Criação de objetos remotos

- No servidor
  - Alguns objetos são criados durante o início da execução do servidor, podendo também ser registrados junto a um *serviço de localização*; outros objetos são criados sob demanda, em resposta aos pedidos dos próprios clientes
  - Código do servidor deve conter as classes do *dispatcher, skeleton*, e de todos os objetos remotos que ele gerencia
- No cliente
  - Utilização de um serviço de nomes para localizar referências remotas para objetos de interesse, ou criação de novos objetos remotos através dos serviços de uma *fábrica de objetos*
    - A fábrica de objetos esconde dos clientes os detalhes de criação de objetos de uma determinada classe
  - Aplicação deve ter acesso ao código do *proxy* dos objetos remotos que irá invocar ou utilizar um mecanismo de invocação dinâmica

## Ativação de objetos remotos

- Motivação: manter todos os objetos ativos (ou seja, executando continuamente) pode ser impraticável
  - Nem todos os objetos são invocados o tempo todo
  - Objetos não invocados mas em execução representam um desperdício de recursos que deve ser evitado
- Solução: objetos devem ser ativados apenas quando requisitados pelos clientes
  - Um objeto é considerado *ativo* quando está disponível para invocação, e *passivo* quando não está ativo mas pode ser ativado sob demanda
  - Um objeto passivo é composto pela sua classe e pelo seu estado armazenado de forma persistente
  - O processo de ativação de um objeto passivo consiste em criar um novo objeto a partir de sua classe e inicializar suas variáveis de instância com o valor do seu estado previamente armazenado




## Persistência de objetos

- Um objeto é considerado *persistente* se ele "sobrevive" às ativações do processo que o gerencia
- Gerenciada por um mecanismo de persistência (ex.: *CORBA Persistent Service*, *Java Data Objects* (JDO), *Java Persistence API* (JPA), *Hybernate*, etc)
  - Suporta o armazenamento de um grande número de objetos persistentes em disco ou BD nativo até que estes sejam requisitados
  - Ativação feita de modo transparente para os clientes
  - Objetos ativos que não são mais necessários podem ser apassivados para abrir espaço para a ativação de novos objetos
  - Tempo e local da ativação/apassivação definidos de acordo com uma política específica
    - Sob demanda, ao final de transações, em *caches* locais, etc.

## Serviço de localização

- Referência remota por pode não ser suficiente para identificar a localização física de um objeto remoto
  - IP + porta contidos na referência válidos apenas se objeto permanecer no mesmo processo durante todo o seu ciclo de vida
  - Invocações precisam tanto da referência remota para o objeto alvo como de um endereço de rede para o qual enviar as invocações
- Um *serviço de localização* ajuda os clientes a obterem referências remotas para objetos remotos de seu interesse
  - Contém uma tabela que mapeia referências remotas para a localização "mais provável" de seus objetos (objetos podem ter migrado de processo ou até de máquina desde a última atualização da tabela)
  - Pode ser estendido com um esquema de busca por difusão ou através de ponteiros de encaminhamento. Como?




## Coleta de lixo distribuída

- Motivação: garantir que um objeto distribuído continue existindo enquanto houver referências (locais ou remotas) para ele em algum ponto da rede, e que ele seja devidamente "coletado" quando essas referências deixarem de existir
- Algoritmo utilizado em Java/RMI:
  - Baseado na contagem de referências
  - Referências remotas implementadas como *proxy* para o objeto remoto nos processos clientes
  - O servidor responsável pelo objeto remoto deve ser informado toda vez que uma referência (*proxy*) para o objeto é criada ou removida em algum cliente
  - Trabalha em cooperação com o coletor de lixo local de cada cliente

## Coleta de lixo distribuída

- Algoritmo em detalhes:
  - Cada servidor mantém uma tabela com informações sobre o conjunto de processos clientes que possuem referências remotas para cada um dos seus objetos remotos
  - Quando um processo cliente cria ou recebe uma referência para algum objeto remoto, ele invoca o método *addRef* no servidor do objeto, que então inclui o ID do processo cliente na tabela de referências remotas para aquele objeto
  - Quando o coletor de lixo local de um processo cliente detecta que o *proxy* para um objeto remoto não mais está sendo referenciado, ele invoca o método *removeRef* no servidor do objeto, que então remove o ID do processo cliente da tabela de referências remotas para aquele objeto




## Coleta de lixo distribuída

- Algoritmo em detalhes (cont.):
  - Quando não houver mais processos na tabela de referência remotas para um determinado objeto do servidor, nem houver referências locais para ele, o mesmo deverá ser coletado pelo coletor de lixo local

## Coleta de lixo distribuída

- Propriedades do algoritmo:
  - Implementado através da colaboração entre os módulos de referência remota do servidor e dos processos clientes, utilizando um protocolo requisição-resposta com semântica de invocação *no-máximo-uma-vez*
  - Não necessita de qualquer forma de sincronização global
  - Minimiza o impacto na invocação dos objeto remotos (executado apenas quando referências remotas são criadas ou removidas)




## Coleta de lixo distribuída

- Aspectos de tolerância a falhas:
  - Risco de coleta “indevida” de um objeto se sua única referência remota for removida antes da criação de uma nova referência previamente solicitada (isto é, se uma chamada a *removeRef* for processada antes da chamada a *addRef* no servidor)
    - Solução: adicionar uma entrada temporária na tabela de referências até que a criação da nova referência seja efetivada
  - Risco de inconsistência na contagem das referências diante de falhas de comunicação ou no servidor
    - Solução: implementar *addRef* e *removeRef* como operações idempotentes, e imediatamente chamar *removeRef* no caso de *addRef* falhar
    - Falha na execução de *removeRef* pode ser contornada com um mecanismo de “empréstimo” de referências (descrito a seguir)

## Coleta de lixo distribuída

- Aspectos de tolerância a falhas (cont.):
  - Risco de referências "órfãs" diante de falhas nos clientes ou na execução de *removeRef*
    - Solução: servidor apenas "empresta" referências remotas aos clientes por um prazo pré-determinado – referências não renovadas são removidas ao final desse prazo
    - Mecanismo similar ao utilizado na tecnologia Jini (Sun)




## Agenda

- Fundamentos de objetos distribuídos
- Comunicação entre objetos distribuídos
- Notificação distribuída de eventos
- Exemplo de aplicação utilizando Java RMI

## Notificação distribuída de eventos

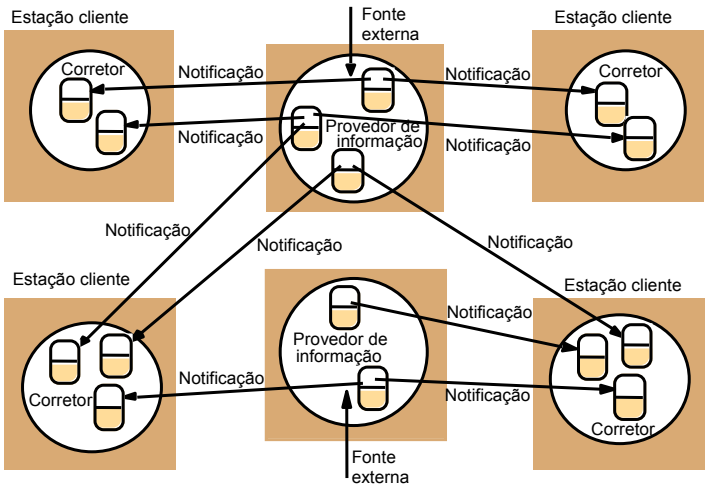
- Motivação: permitir que um objeto possa reagir a mudanças (eventos) ocorridos em outros objetos remotos
  - Extensão do modelo de notificação de eventos tradicionalmente utilizado na construção de GUIs
  - Comunicação assíncrona entre objetos controlada pelo objeto notificado
- Baseada no paradigma *publicar-associar* (*publish-subscribe)*
  - Objetos que geram eventos "publicam" os tipos de eventos que estarão disponíveis para observação por outros objetos remotos
  - Objetos que desejam receber notificações de um objeto que tenha publicado seus eventos "se associam" aos tipos de eventos nos quais tenham interesse
- Notificações sobre eventos transmitidas na forma de objetos




## Notificação distribuída de eventos (cont.)

- Exemplos de aplicações:
  - Alterações em documentos compartilhados
  - Monitoramento de sistemas (usinas nucleares, reatores, etc)
  - Entrada/saída de pessoas em ambientes controlados
  - Variações em índices econômicos (ações, dólar, etc)
- Característica dos sistemas distribuídos baseados em notificação de eventos:
  - Heterogêneos/desacoplados
    - Permitem a comunicação entre componentes que não tenham sido original projetados para tal
  - Assíncronos
    - Objetos que publicam eventos não precisam estar sincronizados com os objetos associados ao tipo dos eventos publicados

# Exemplo: corretora de ações da bolsa



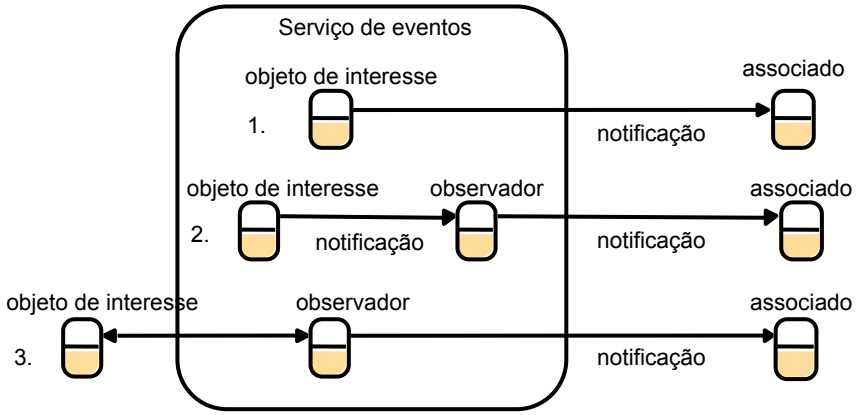




# Tipos de eventos

- Um objeto pode gerar eventos de vários tipos
- Informações sobre um evento são representadas na forma de atributos (nome do evento, ID do objeto gerador, data/hora da geração,nome e parâmetros da operação, etc)
- Usados tanto para associação quanto para notificação
  - Associação baseada no tipo dos eventos de interesse, bem como no valor de um ou mais de seus atributos
  - Notificação enviada apenas quando o evento gerado satisfaz o tipo e os valores de atributos especificados pelo objeto associado

## Arquitetura para a notificação distribuída de eventos







## Arquitetura para a notificação distribuída de eventos (cont.)

- Projetada para desacoplar os objetos geradores de eventos dos objetos associados (consumidores de eventos)
- Serviço de eventos
  - Principal componente da arquitetura
  - Mantém uma base de dados com informações sobre os eventos gerados e os interesses registrados pelos associados
- Papéis dos participantes:
  - Objeto de interesse
    - Detecta mudanças de estado (decorrentes da invocação de uma ou mais de suas operações) que podem ser de interesse para outros objetos
    - Considerado parte do serviço de eventos caso o próprio objeto envie as notificações aos objetos associados

## Arquitetura para a notificação distribuída de eventos (cont.)

- Papéis dos participantes (cont.):
  - Notificação
    - Contém informações sobre um evento (tipo e atributos)
  - Associado
    - Informa ao serviço de eventos os tipos de evento nos quais tem interesse, e recebe notificações sobre a ocorrência de tais eventos
  - Observador
    - Desacopla um objeto de interesse de seus associados, evitando que o objeto de interesse tenha que distinguir entre os vários interesses registrados pelos seus associados
  - Gerador de eventos
    - Declara que irá gerar eventos de um determinado tipo
    - Pode tanto exercer o papel de objeto de interesse quanto de observador




## Arquitetura para a notificação distribuída de eventos (cont.)

- Variações de implementação:
  - Objeto de interesse localizado dentro do serviço de eventos, sem a presença de um observador
    - O objeto de interesse envia as notificações diretamente aos associados
  - Objeto de interesse localizado dentro do serviço de eventos, com a presença de um observador
    - O objeto de interessa envia as notificações aos associados através do observador
  - Objeto de interesse localizado fora do serviço de eventos, com a presença de um observador
    - O observador consulta periodicamente o objeto de interesse para descobrir quando há a ocorrência de eventos, e, se for o caso, envia as notificações aos associados

## Arquitetura para a notificação distribuída de eventos (cont.)

- Semântica de entrega
  - Variedade de protocolos e garantias
  - Escolha dependente dos requisitos da aplicação
    - *IP Multicast* pode ser adequado em aplicações onde notificações são enviadas com freqüência
      - Ex.: jogos online
    - O mesmo protocolo pode não ser suficiente para aplicações que exijam garantias de entrega
      - Ex.: na aplicação da corretora de ações, todos os corretores devem receber as mesmas informações com relação às ações de uma mesma empresa
    - Aplicações com requisitos de tempo real exigem garantias ainda mais fortes
      - Ex.: monitoramento de pacientes e de usinas nucleares




## Arquitetura para a notificação distribuída de eventos (cont.)

- Papeis dos observadores:
  - *Encaminhamento* – um observador pode encaminhar notificações para associados em nome de um ou mais objetos de interesse
    - Objeto de interesse deve passar informações sobre os interesses de seus associados para o observador
    - Papel do objeto de interesse fica reduzido a enviar notificações para o observador, o qual se encarregará de entregá-las aos devidos associados
  - *Filtragem* – um observador pode aplicar filtros para reduzir o número de notificações recebidas pelos associados
    - Filtros podem ser definidos baseados nos valores dos atributos de uma notificação
      - Ex.: dos eventos relacionados a retiradas de uma conta bancária, um associado pode estar interessado apenas naqueles que envolvem montantes acima de R$ 100,00

## Arquitetura para a notificação distribuída de eventos (cont.)

- Papeis dos observadores (cont.):
  - *Padrões de eventos* – um observador também podem selecionar as notificações a serem entregues baseado em padrões de eventos especificados pelos associados
    - Um padrão especifica um tipo de relacionamento entre dois ou mais eventos
      - Ex.: um associado pode estar interessado apenas em eventos de retirada bancária quando três desses eventos ocorrerem em seqüência sem nenhum depósito entre eles
    - Outro tipo de padrão pode ser obtido correlacionando os eventos gerados por múltiplos objetos de interesse
      - Ex.: um associado pode estar interessado em receber eventos apenas após um número mínimo de objetos de interesse ter começado a gerá-los




## Arquitetura para a notificação distribuída de eventos (cont.)

- Papeis dos observadores (cont.):
  - *Caixa de mensagens* – um observador pode armazenar, na forma de uma caixa de mensagem persistente, notificações recebidas de diferentes objetos de interesse para envio posterior
    - Notificações apenas são enviadas quando os associados interessados estiverem aptos a recebê-la
      - Ex.: um associado pode estar isolado por falhas de conexão ou ter sido temporariamente desativado
    - Modelo de notificações de eventos utilizado pelo serviço JMS (*Java Messaging Service*)

## Agenda

- Fundamentos de objetos distribuídos
- Comunicação entre objetos distribuídos
- Notificação distribuída de eventos
- Exemplo de aplicação utilizando Java RMI




## Uma aplicação distribuída de edição compartilhada de elementos gráficos

- Aplicação distribuída utilizada no livro como um estudo de caso para ilustrar o uso de Java RMI e CORBA
  - Permite que um grupo de usuários compartilhe uma visão comum de uma tela de desenho contendo vários elementos gráficos, cada um desenhado por um usuário diferente.
- Servidor da aplicação mantém o estado corrente da tela de desenho oferecendo operações para os clientes remotos poderem:
  - adicionar um novo elemento gráfico, recuperar um elemento gráfico existente, ou recuperar todos os elementos gráficos existente
  - recuperar o seu número de versão ou o número de versão de um elemento gráfico existente

## As interfaces remotas *Shape* e *ShapeList*

A classe *GraphicalObject* deve implementar a interface *Serializable*.

```
import java.rmi.*;
import java.util.Vector;
public interface Shape extends Remote {
    int getVersion() throws RemoteException;
    GraphicalObject getAllState() throws RemoteException;
}
public interface ShapeList extends Remote {
    Shape newShape(GraphicalObject g) throws RemoteException;
    Vector allShapes() throws RemoteException;
    int getVersion() throws RemoteException;
}
```




## A classe *Naming* (serviço de nomes de Java RMI)

*void rebind (String name, Remote obj)*
Método utilizado pelo servidor para registrar o identificador de um objeto remoto por nome (substitui o registro anterior se o nome já estiver registrado para outro objeto).

*void bind (String name, Remote obj)*
Método alternativo utilizado pelo servidor para registrar o identificador de um objeto remoto por nome (gera uma exceção se o nome já estiver registrado para outro objeto).

*void unbind (String name, Remote obj)*
Método utilizado pelo servidor para remover um registro (nome e referência).

*Remote lookup(String name)*
Método utilizado pelos clientes para procurar um objeto remoto por nome. Retorna uma referência remota.

*String [] list()*
Retorna todos os nomes registrados no serviços de registro.

## A classe *ShapeListServer*

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




## A classe *ShapeListServant*

```
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
    private int version;
    public ShapeListServant()throws RemoteException{...}
    public Shape newShape(GraphicalObject g) throws RemoteException {
        version++;
            Shape s = new ShapeServant( g, version);
            theList.addElement(s);
            return s;
    }
    public  Vector allShapes()throws RemoteException{...}
    public int getVersion() throws RemoteException { ... }
}
```

## A classe *ShapeListClient*

```
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") ;
          Vector sList = aShapeList.allShapes();
     } catch(RemoteException e) {System.out.println(e.getMessage());
     }catch(Exception e) {System.out.println("Client: " + e.getMessage());}
   }
}
```




## Comunicação entre objetos distribuídos via *callbacks*

- O termo *callback* se refere ao mecanismo através do qual o servidor notifica os clientes sobre a ocorrência de eventos de seu interesse
- Utilizado para evitar os problemas causados quando os clientes precisam fazer consultas periódicas ao servidor
  - Degradação do desempenho do servidor
  - Descompasso entre a ocorrência de um evento e a notificação dos clientes interessados nele
- Também introduz novos problemas:
  - Servidor precisa estar sempre atualizado sobre a lista de clientes a serem notificados (Como manter o servidor atualizado?)
  - Notificação envolve uma série de invocações remotas (potencialmente demoradas) para cada um dos clientes da lista (Como amenizar o impacto das invocações remotas?)

## Implementação de *callbacks* em Java RMI

- O cliente define uma interface remota com um método de *callback*:

```
public interface Callback implements Remote {
    void callback(int version) throws Remote Exception;
}
```

- O objeto remoto inclui em sua interface remota métodos para os clientes registrarem objetos *callback*:

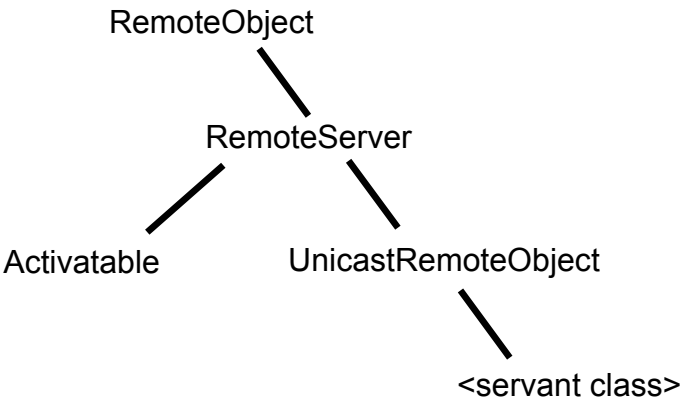
```
int register(Callback callback) throws Remote Exception;
void deregister(int callbackID) throws Remote Exception;
```




## Aspectos de projeto e implementação de Java RMI

- Uso de reflexão
  - As primeiras versões de Java RMI implementavam um modelo de arquitetura similar ao estudado neste capítulo
  - A partir de Java 1.2 a implementação do RMI passou a utilizar o mecanismo de reflexão da linguagem para criar um *Dispatcher* genérico, eliminando a necessidade de *Skeletons*
- Hierarquia de classes
  - Java RMI oferece um hierarquia de classes para a implementação de objetos remotos com diferentes características (ex: “ativável”, “replicável”, etc)
  - A classe *UnicastRemoteObject* é a mais utilizada na prática para implementar objetos remotos “simples”

## Hierarquia de classes em Java RMI







## Exercícios recomendados

- No livro: 5.1-5, 5.12, 5.14, 5.16-18