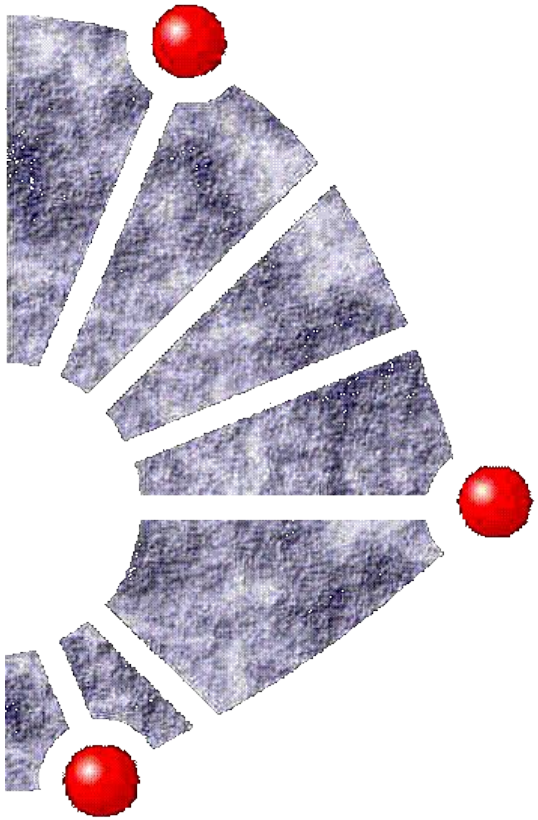
Implementando Aplicações Multi-Thread e Distribuídas



AULA 3

Processos e Threads

- Time-sharing
  - mecanismo pelo qual um certo número de "processos" que se deseja que sejam executados concorrentemente tenham acesso a "fatias" do tempo do processador, sendo interrompidos assim que estas "fatias" se esgotam e retomando sua execução quando lhe são atribuídas novas "fatias" de tempo
- Cada Processo
  - possui sua própria porção de memória
  - executado independentemente de outros processos
- Recursos comuns
  - compartilhados via mecanismos de sincronização
    - semáforos, filas, monitores, etc.

Processos e Threads

- Comunicação Inter-processos
  - variáveis globais não protegidas
  - variáveis compartilhadas protegidas por monitores
  - passagem de mensagens via recursos do sistema operacional
- Threads
  - trechos de código, dentro de um mesmo processo, que se deseja que rodem concorrentemente
  - grânulos finos concorrentes de um mesmo processo
  - compartilham todos os recursos do processo do qual são originados
  - são criados dentro de um processo e a ele pertencem
  - dependem da disponibilidade do sistema operacional
  - ou … podem ser emulados por meio de um processo que os gerencie

Threads e Java

- Algumas linguagens de programação
  - contém diretivas para o uso de threads
- Java
  - contém toda a infra-estrutura para a programação multi-thread
  - interpretador Java emula os threads dentro do processo ao qual pertence
  - Suporte à programação multi-thread é centralizado em torno da classe java.lang.Thread
- Classe Thread
  - permite a criação de objetos do tipo Thread, onde cada objeto terá seu próprio fluxo de controle independente

Threads e Java

- Dois mecanismos para a criação de Threads
  - Mecanismo 1:
    - cria-se uma sub-classe da classe Thread
    - efetua-se um override to método run(), que é o ponto de entrada para a execução do thread
    - após a criação de um objeto desta classe, invoca-se o método start(), que inicia o novo thread e por sua vez invoca o método run()
  - Mecanismo 2:
    - cria-se uma classe implementando a interface Runnable
    - implementa-se o método run() da interface Runnable
    - cria-se um objeto da classe Thread, passando para o construtor um objeto da classe implementando Runnable
    - invoca-se o método start() do objeto do tipo Thread

Threads e Java

- Estados de um Thread
  - new thread - assim que é criado
  - runnable - depois que é startado
  - not runnable - bloqueado esperando por um evento
  - dead - quando o método run() retorna
- Métodos Importantes
  - sleep() - faz o thread "dormir" por um certo tempo
  - interrupt() - interrompe definitivamente a execução do thread
  - join() - bloqueia o thread corrente até que o thread indicado morra
  - is_alive() - verifica se um thread está ainda "vivo"

Threads e Java

- Prioridades
  - set_Priority(), get_Priority()
  - MIN_PRIORITY(1), NORM_PRIORITY(5), MAX_PRIORITY(10)
- Sincronização
  - Métodos Sincronizados
    - synchronized function(…)
  - Statements Sincronizados
    - synchronized (expression) statement
- Monitores e Sincronização de Processos
  - wait()
  - notify()
  - notifyAll()

Threads e Java

- Threads do tipo Daemon
  - Threads que executam em background e provêm serviços a outras threads ou processos
  - normalmente executam continuamente um conjunto de instruções onde esperam por uma requisição de serviço, realizam o serviço solicitado e passam a aguardar nova requisição de serviço
  - uma característica de Threads do tipo Daemon é que eles não são destruídos pelo mecanismo de Garbage Collection mesmo que não sejam referenciados por nenhum outro objeto rodando na máquina virtual
- Para criar um Daemon Thread
  - método setDaemon() deve ser invocado antes de se dar o start() da thread

Processamento Distribuído

- Redes de Computadores
  - Software para o uso da rede
    - Compartilhamento de Arquivos e Impressoras
    - Serviços de Rede (autenticação, acesso a recursos, etc)
  - Emergência de um novo paradigma : sistemas distribuídos
- Sistemas Distribuídos
  - Sistemas Operacionais Distribuídos
    - Software do sistema é distribuído ao longo da rede
  - Aplicações Distribuídas
    - Novos métodos de programação
    - Tecnologias Básicas
      - Sockets, RPC, Objetos Distribuídos

Programação em Rede

- Suite de Programação Internet
  - IP - Internet Protocol
  - TCP - Transport Control Protocol - orientado a conexão
  - UDP - User Datagram Protocol - sem conexão
- Comunicação
  - comutação de pacotes de dados - pacotes IP
- Dispositivo de Acesso à Rede
  - Endereço IP - números de 32 bits - NNN.NNN.NNN.NNN
  - Domain Name System (DNS) - associação de nomes a endereços IP - DNS Servers
- Portas
  - endereço dentro de um computador (16 bits)
  - número associado a um tipo de serviço

Programação em Rede

- TCP - Serviço Orientado a conexão
  - estabelece uma conexão entre uma origem (porta/end.IP) e um destino (porta/end.IP) que perdura até que a mesma seja explicitamente encerrada
  - comunicação confiável
- UDP - Serviço sem Conexão
  - Não estabelece um vínculo direto entre origem e destino
  - envia datagramas sem confirmação de resposta e sem técnicas de correção de erros
  - mais rápidos que o TCP
- Unicast x Multicast
  - unicast - comunicação ponto a ponto
  - multicast - grupo de hosts recebendo um mesmo endereço IP

Programação com Sockets

- Programação Socket
  - Base para programação em rede utilizando TCP/IP
  - Apareceu como uma abstração para programação em rede dentro de sistemas UNIX, tornando-o compatível com o paradigma básico de I/O do UNIX
- Paradigma básico de I/O do UNIX
  - open-read-write-close
  - insuficiente para gerenciar todos os serviços e protocolos de rede
- Socket
  - generalização do mecanismo de acesso a arquivos do UNIX, provendo um ponto de conexão para comunicação
  - programas aplicativos solicitam a criação de um socket ao sistema operacional, quando necessário

Sockets e Java

- java.net
  - InetAddress - encapsula endereços IP e suporta conversão entre notação decimal pontuada e nomes de hosts
  - Socket, ServerSocket, DatagramSocket, MulticastSocket - implementa sockets clientes e servidores
  - SocketImpl e DatagramSocketImpl (classes) e SocketImplFactory (interface) - auxiliam a criação de sockets customizados
  - URL, URLConnection, HttpURLConnection e URLEncoder - implementam conexões Web de alto nível
  - FileNameMap (interface) - usada no mapeamento de nomes de arquivos a tipos MIME

Sockets e Java

- Classe InetAddress

`boolean equals(Object obj)`	Compares this object against the specified object.
`byte[] getAddress()`	Returns the raw IP address of this InetAddress object.
`static InetAddress[] getAllByName(String host)`	Determines all the IP addresses of a host, given the host's name.
`String toString()`	Converts this IP address to a String.
`static InetAddress getByName(String host)`	Determines the IP address of a host, given the host's name.
`String getHostAddress()`	Returns the IP address string "%d.%d.%d.%d".
`String getHostName()`	Returns the hostname for this address.
`static InetAddress getLocalHost()`	Returns the local host.
`int hashCode()`	Returns a hashcode for this IP address.
`Boolean isMulticastAddress()`	Utility routine to check if the InetAddress is a IP multicast address.

Sockets e Java

- Classe Socket

`Socket(InetAddress address, int port)`	Creates a stream socket and connects it to the specified port number at the specified IP address.
`Socket(String host, int port)`	Creates a stream socket and connects it to the specified port number on the named host.
`void close()`	Closes this socket.
`InetAddress getInetAddress()`	Returns the address to which the socket is connected.
`InputStream getInputStream()`	Returns an input stream for this socket.
`InetAddress getLocalAddress()`	Gets the local address to which the socket is bound.
`int getLocalPort()`	Returns the local port to which this socket is bound.
`OutputStream getOutputStream()`	Returns an output stream for this socket.
`int getPort()`	Returns the remote port to which this socket is connected.

Sockets e Java

- Classe ServerSocket

`ServerSocket(int port)`	Creates a server socket on a specified port.
`ServerSocket(int port, int backlog)`	Creates a server socket and binds it to the specified local port number.
`ServerSocket(int port, int backlog, InetAddress bindAddr)`	Create a server with the specified port, listen backlog, and local IP address to bind to.
`Socket accept()`	Listens for a connection to be made to this socket and accepts it.
`void close()`	Closes this socket.
`InetAddress getInetAddress()`	Returns the local address of this server socket.
`int getLocalPort()`	Returns the port on which this socket is listening.
`Protected void implAccept(Socket s)`	Subclasses of ServerSocket use this method to override accept() to return their own subclass of socket.
`String toString()`	Returns the implementation address and implementation port of this socket as a String.

Sockets e Java

- Classe DatagramSocket
  - Define socket para enviar e receber pacotes datagramas pela rede
    - sem conexão
    - transporte não-confiável
  - Porta local
    - especificada no método construtor ou designada pelo sistema
  - Envio ou recepção de pacotes
    - argumento do método é um DatagramPacket
- Classe DatagramPacket
  - Pacote de dados binários
    - conteúdo é um arranjo de bytes
    - enviado ou recebido por DatagramSocket
      - envio: construtor especifica conteúdo e destino
      - recepção: construtor indica espaço para pacote
  - Métodos para extração de informação
    - obter fonte do pacote: endereço, porta
    - extrair arranjo de bytes: comprimento, dados

Sockets e Java

Cliente

```
…
Socket cl = new Socket(destination,port);
…
BufferedReader inS = new
BufferedReader( cl.getInputStream())
DataOutputStream outS = new
DataOutputStream( cl.getOutputStream());
…
/* use inS and outS */
```

Servidor

```
…
ServerSocket s = new ServerSocket(PORT);
Socket cl = s. accept();
…
String dnm =
    cl.getInetAddress().getHostName();
int dpt = cl.getPort();
…
BufferedReader inS = new
    BufferedReader( new InputStreamReader
    (cl.getInputStream()));
DataOutputStream outS = new
    DataOutputStream( cl.getOutputStream());
…
/* use inS and outS  */
```

Classes p/ Programação de Alto Nível

- Classe URL
  - Permite referenciar e transferir dados URL
  - Referências
    - construção com *string* completa ou com especificações separadas
  - Alternativas para transferências simples
    - conteúdo completo, abrir conexão URL ou ler de *stream* de entrada de dados
- Classe URLConnection
  - Permite maior controle sobre a transferência de URLs
  - Consulta à informação de cabeçalho
    - tipo de conteúdo, data de última modificação
  - Transferências de conteúdo completo e por *streams* de entrada e saída
  - Comportamento da conexão
    - leitura/escrita, interação de usuário, uso de cache para obter recurso

Classes p/ Programação de Alto Nível

- Classe HttpURLConnection
  - Determinação do método de requisição (comando) HTTP
    - enviado pelo *stream* de saída obtido pelo método da superclasse
  - Manipulação do retorno de servidor HTTP
    - Consulta a código e mensagem de retorno
    - Definição de constantes associadas aos códigos de retorno de servidores HTTP

Java e Objetos Distribuídos

- java.rmi
  - Pacote RMI - Remote Method Invocation
  - Implementação nativa do Java para objetos distribuídos
  - exige que os objetos distribuídos sejam objetos Java
- org.omg.CORBA
  - Implementação Java de um ORB CORBA
  - permite objetos de múltiplas linguagens
  - é computacionalmente mais custoso do que o RMI
- Sockets x RMI x CORBA
  - caso a comunicação seja simples, deve-se utilizar Sockets
  - caso se deseje abstrair a passagem de mensagens, e o sistema distribuído utilizará somente Java, deve-se utilizar RMI
  - caso seja necessário (acessar objetos desenvolvidos/ oferecer objetos para programas) em outras linguagens, utilizar CORBA

Java RMI
Remote Method Invocation

- Mecanismo nativo Java para programação cliente-servidor
  - Modelo de computação distribuída para a linguagem Java
- Diversas similaridades com RPC
  - RPC é neutro
    - denominador comum entre diversas linguagens
  - RPC não pode explorar devidamente todos os recursos oferecidos pelas linguagens de programação
- RMI é integrado à linguagem Java
  - Todos os recursos de Java podem ser usados em RMI

Principais Características do RMI

- RMI é orientado a objetos
  - RPC transfere apenas tipos primitivos de dados
    - estruturas devem ser decompostas
  - RMI permite transferência de objetos como um argumento único
- RMI pode mover comportamento.
  - Implementação de classe pode ser transferida
- RMI incorpora mecanismos de segurança
  - Necessidade para execução de código remoto
- RMI não exclui interação com sistemas legados
  - aplicações existentes implementadas em outras linguagens podem ser integradas
    - JNI, JDBC realizam ponte para Java

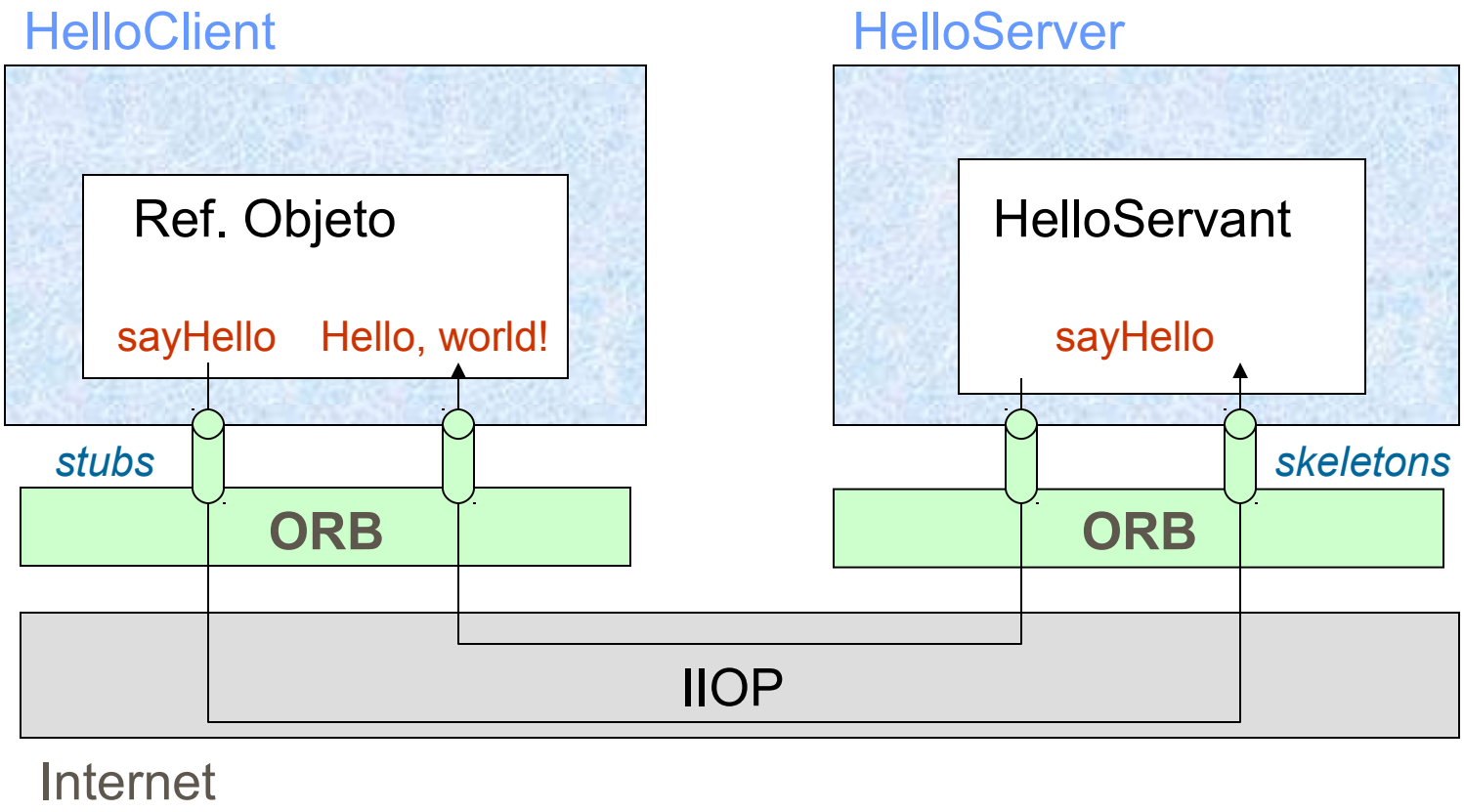
Criando Aplicações Java RMI

- Registry do RMI
  - rmiregistry, rodando na máquina do servidor
- Objeto Servidor
  - instalar um RMISecurityManager
  - definir interface que estende a interface Remote
  - implementar esta interface
  - cadastrar nome utilizando Naming.rebind()
- Geração de Stubs e Skeletons
  - rmic
- Objeto Cliente
  - instala um RMISecurityManager
  - obtém stub do servidor via Naming.lookup()
  - Acessa normalmente os métodos do servidor

CORBA

- Especificação OMG
  - padronização, diversos serviços
- Incorporado a diversos produtos
  - Na Web: netscape, explorer, Java
- Cliente
  - utiliza referência para objeto remoto
  - método *stub* ligado ao ORB repassa invocação
- Servidor
  - *skeleton* traduz invocação remota
- Java CORBA
  - Permite que aplicação Java comunique-se com objetos remotos
  - Objetos remotos podem ter sido implementados em qualquer linguagem

CORBA


HelloClient
HelloServer
Ref. Objeto
sayHello
Hello, world!
HelloServant
sayHello
stubs
skeletons
ORB
ORB
IIOP
Internet

Java CORBA

- Implementa mapeamento IDL/Java
  - definido pelo OMG
- Incorpora ORB
  - permite comunicação entre Java e outras aplicações CORBA
- Arquivo IDL
  - Nome.idl
- idltojava
  - gera os seguintes arquivos:
    - _NomeImplBase.java - skeleton do servidor. É extendida por NomeImplBase no programa servidor
    - _NomeStub.java - stub do cliente - implementa Nome.java
    - Nome.java - interface correspondente ao arquivo IDL
    - NomeHelper.java - funcionalidades auxiliares
    - NomeHolder.java - possui implementação da classe Nome

Processo de Desenvolvimento Java CORBA/IDL

1. definição da interface remota
   - Usa linguagem IDL (OMG)
   - independente da linguagem de programação utilizada pelo cliente ou servidor
2. compilação da interface remota
   - compilador **idltojava**
     - ferramenta de Java IDL
   - a partir da interface IDL gera código da interface Java, de stubs e skeletons, além de código de infra-estrutura
3. implementação do código servidor
   - utliza skeletons para integrar servidor ao ORB
   - implementa métodos da interface remota
   - incorpora mecanismo para iniciar ORB e esperar pela invocação do cliente remoto
4. implementação do código cliente
   - utiliza stubs como base para implementação
   - inicia seu ORB, localiza servidor e obtém referência para objeto remoto

Execução de Aplicações

- Ativar serviço de nomes

```
tnameserv -ORBInitialPort 1050
```

  - único serviço CORBA oferecido atualmente em Java IDL
  - permite localizar objetos pelo nome
- Executar servidor
  - registra serviço e aguarda invocações
- Executar cliente
  - utiliza serviço de nome e servidor

Exemplo de Programa Servidor

```
// Create and initialize the ORB
      ORB orb = ORB.init(args, null);
// Create the servant and register it with the ORB
      HelloServant helloRef = new HelloServant();
      orb.connect(helloRef);
// Get the root naming context
      org.omg.CORBA.Object objRef =
           orb.resolve_initial_references("NameService");
      NamingContext ncRef = NamingContextHelper.narrow(objRef);
// Bind the object reference in naming
      NameComponent nc = new NameComponent("Hello", "");
      NameComponent path[] = {nc};
      ncRef.rebind(path, helloRef);
// Wait for invocations from clients
      java.lang.Object sync = new java.lang.Object();
      synchronized(sync){
        sync.wait();
      }
```

Exemplo de Programa Cliente

```
// Create and initialize the ORB
     ORB orb = ORB.init(args, null);

// Get the root naming context
     org.omg.CORBA.Object objRef =
           orb.resolve_initial_references("NameService");
     NamingContext ncRef = NamingContextHelper.narrow(objRef);

// Resolve the object reference in naming
     NameComponent nc = new NameComponent("Hello", "");
     NameComponent path[] = {nc};
     Hello helloRef = HelloHelper.narrow(ncRef.resolve(path));

// Call the Hello server object and print results
     String Hello = helloRef.sayHello();

     System.out.println(Hello);
```

Java IDL ou Java RMI

- IDL
  - solução padronizada
  - voltado para integração ao legado de software
  - protocolo de comunicação IIOP
    - multiplicidade de plataformas
  - objetos exclusivamente por referências
- RMI
  - solução 100% Java
  - independente de plataforma
    - sem integrar código nativo
  - protocolo de comunicação JRMP
    - Java Remote Messaging Protocol
  - objetos por referência ou por valor
    - download do objeto para execução local

Java Servlets

- Applet
  - aplicação Java cujo código é carregado de um servidor e executado na máquina cliente
- Servlet
  - Aplicação Java para Web que é executada no lado servidor
    - Servlets executam no contexto do processo servidor Web.
- Local de Execução
  - Servlet está para o servidor Web
  - assim como o applet está para o cliente Web

Requisitos para Execução de Servlets

- Servidor Web deve oferecer JVM
- Servidor deve suportar a API Java Servlet
  - Estabelece definição de mecanismos de comunicação entre servidor Web e o servlet
    - acesso a variáveis definidas pelo servidor
    - redirecionamentos
    - envio de mensagens de erro.
  - Abordagens de integração
    - Nativo, Plug-ins ou Ponte para código nativo
- Vantagens da linguagem Java
  - Facilidade de desenvolvimento de código sem bugs.
  - Portabilidade de plataforma

Vantagens de Servlets

- Dinamismo
  - Servlets podem ser carregados dinamicamente para um servidor para aumentar suas funcionalidades
- Desempenho
  - Execução de servlet não está associada a um processo
    - ativação do servlet é mais leve
- Operação eficiente sob múltiplas solicitações
  - não necessariamente implicam em carregamentos múltiplos do código do servlet
- Servlet chaining
  - Na invocação de servlets, a saída de um servlet pode ser direcionada para a entrada de outro servlet.
  - Redirecionamento é transparente para o servlet.
    - Útil para filtragem de informação

Requisitos para Criação de Servlets

- Classe que implementa servlet estende
  `javax.servlet.GenericServlet`
  - redefine o método
    service(ServletRequest, ServletResponse)
- Classe que implementa servlet para HTTP estende
  `javax.servlet.HttpServlet`
  - redefine pelo menos um dos métodos
    doGet(HttpServletRequest,HttpServletResponse) ou
    doPost (HttpServletRequest,HttpServletResponse)

Requisitos para Criação de Servlets

- Compilação requer pacote javax.servlet
  - não é parte do JDK
- Localização do arquivo bytecode
  - diretório *server_root*/servlets
- Integração de servlet ao servidor Web
  - registra nome simbólico para o servlet
  - nome é mapeado à classe no diretório de servlets

Invocação de servlets

- URL com nome do servlet

  http://www.server.com.br/servlet/hello
- URL mapeada para servlet
  - administração associa um nome de arquivo que será associado à execução do servlet

  http://www.server.com.br/hello.txt
- tag <servlet>
  - Server-side includes (arquivos .shtml)