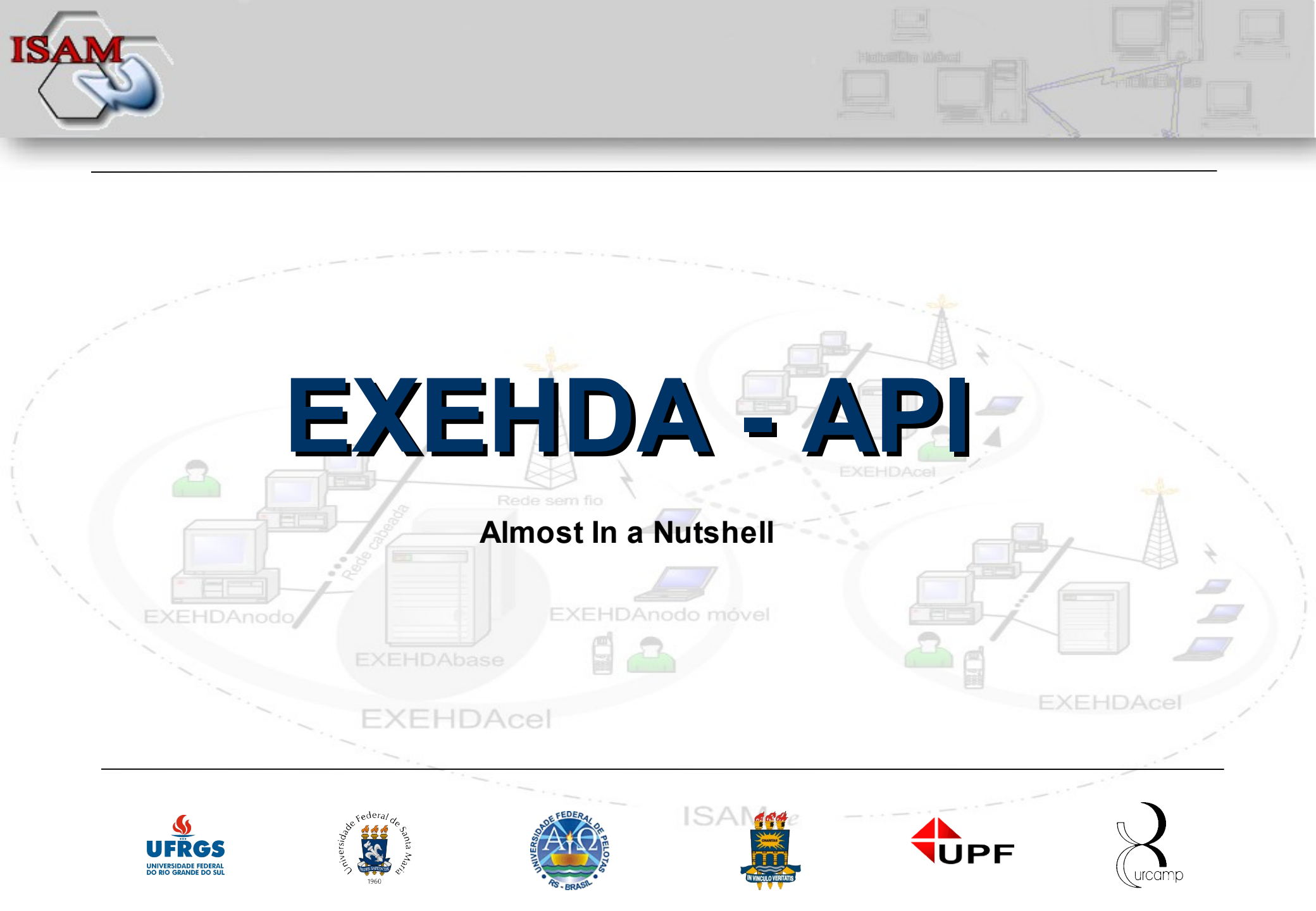

EXEHDA - API
Almost In a Nutshell
Rede sem fio
Rede cabeada
EXEHDAnodo
EXEHDAbase
EXEHDAnodo móvel
EXEHDAcel
EXEHDAcel
EXEHDAcel
ISAM
UFRGS
UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Universidade Federal de Santa Maria
1960
UNIVERSIDADE FEDERAL DE PELOTAS
RS - BRASIL
UPF
urcamp

Autoria

- **Autores**
  - Ingrid Oliveira de Nunes
  - Fábio Reis Cecin
  - Adenauer Yamin
  - Rodrigo Real
  - Luciano Cavalheiro da Silva
- **Data**
  - Outubro de 2006
- **Versão**
  - PDP-POS

Desafios de uma Grade Pervasiva

Integrando conceitos

Computação móvel

+

Computação em grade

+

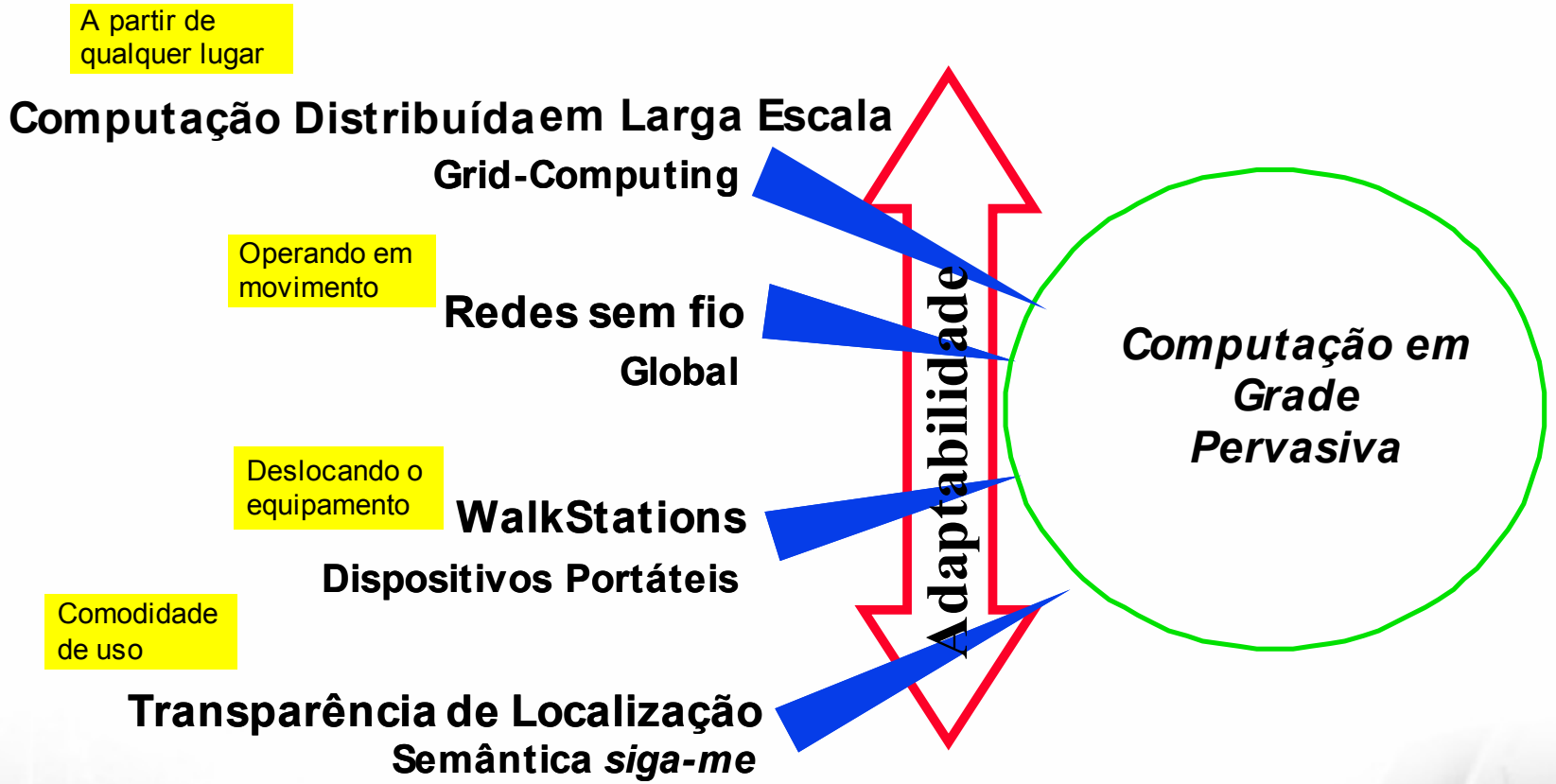
Computação consciente do contexto

Infra-estrutura para

Grade Pervasiva

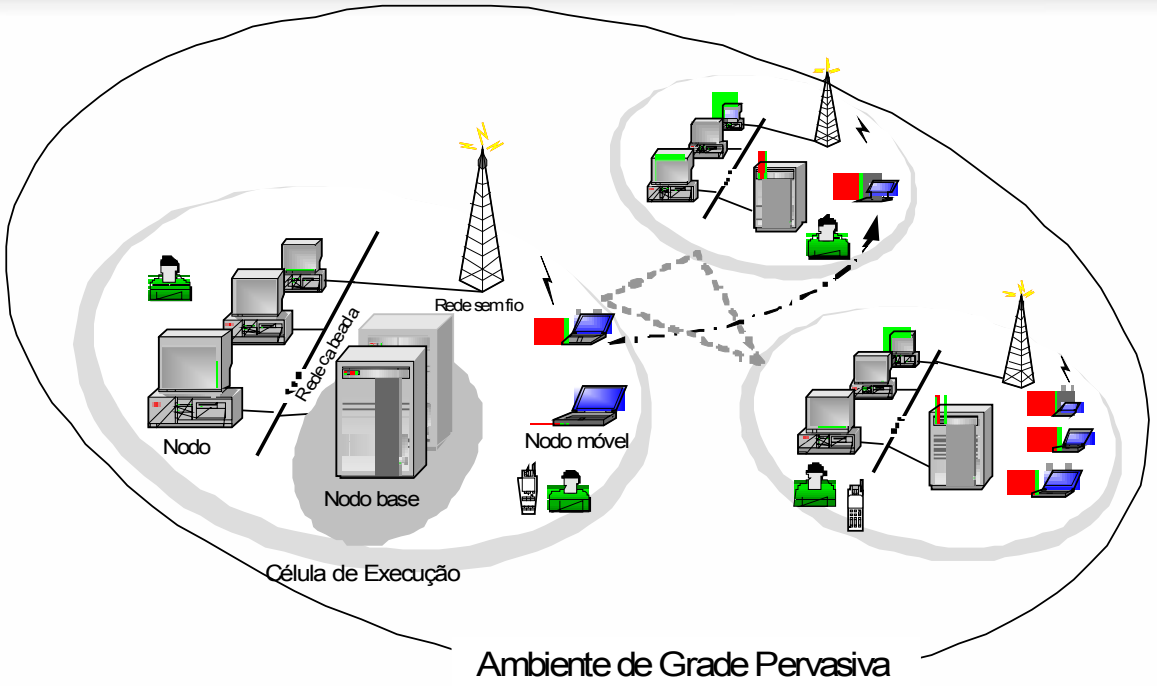
Desafios de uma Grade Pervasiva

Principais tecnologias envolvidas



Desafios de uma Grade Pervasiva

A organização física do GRADEp



Combina rede infra-estruturada, com suporte a operação sem fio.

Computação móvel

Características:

- Organização celular (critério de vizinhança estático e/ou dinâmico)
- Gerência multi-institucional
- Composição dinâmica (incremental)

Computação em grade

Desafios de uma Grade Pervasiva

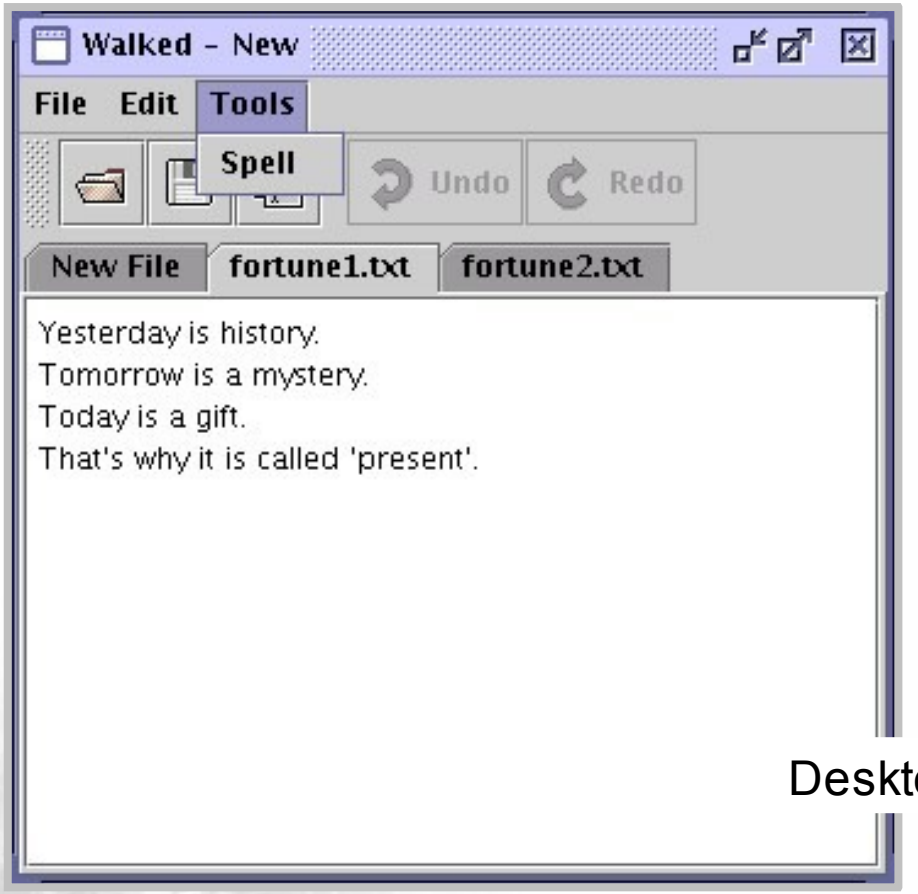
Principais aspectos operacionais

- **Potencialização da:**
  - Heterogeneidade
    - de recursos;
    - comportamentos operacionais (flutuação na banda das conexões, re-acomodação física dos nodos, etc.).
  - Escalabilidade (necessidade de)
  - Variabilidade na disponibilidade e no acesso aos recursos
    - gerência multi-institucional;
    - conexões transientes.
- **Semântica Siga-me (*follow-me*)**
- **Adaptação funcional e não-funcional as condições dinâmicas do meio**

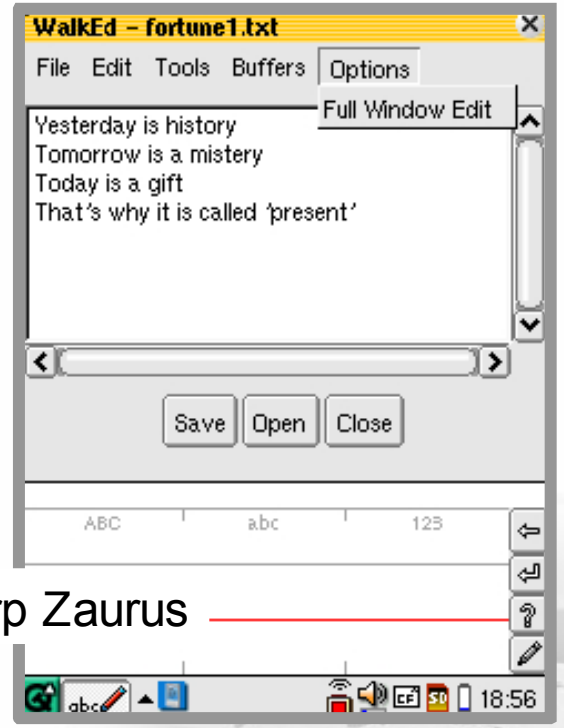
EXEHDA: cenários de utilização por exemplos

WalkEd (Editor de Texto Pervasivo)

Perfil: integração computação móvel e computação distribuída;
Objetivo: funcionalidade e flexibilidade no uso da ferramenta de edição.



Desktop



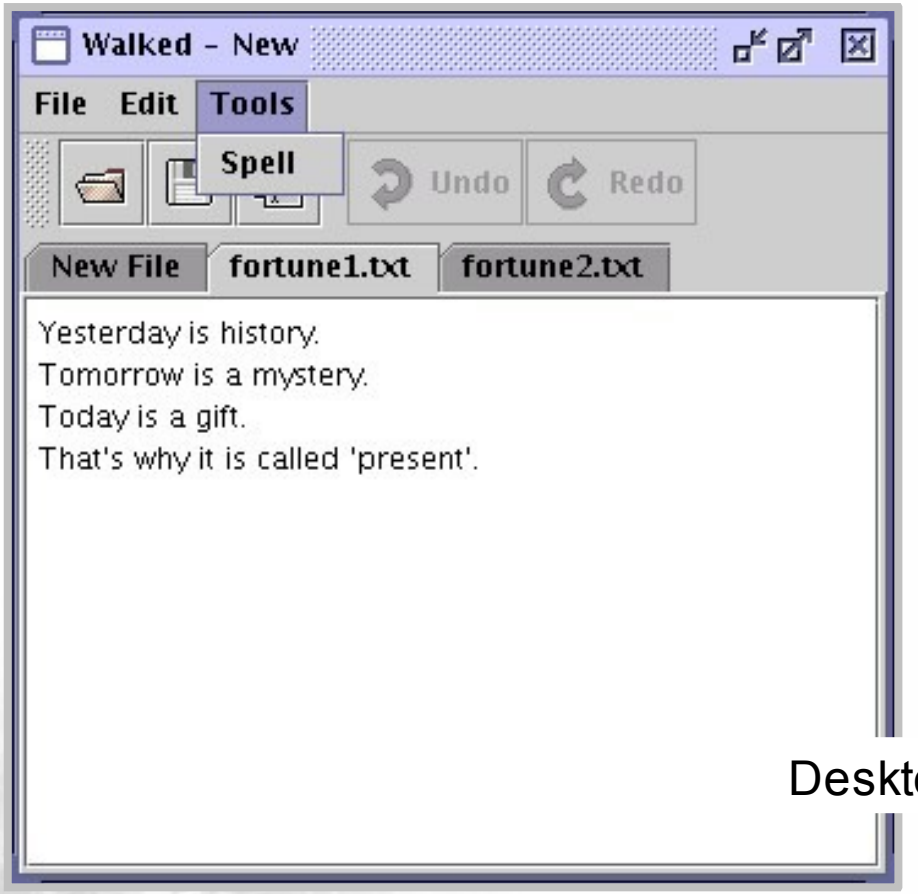
Sharp Zaurus

EXEHDA: cenários de utilização por exemplos

WalkEd (Editor de Texto Pervasivo)

Perfil: integração computação móvel e computação distribuída;
Objetivo: funcionalidade e flexibilidade no uso da ferramenta de edição.

Adaptação ao tipo do dispositivo



Desktop

PDA Sharp Zaurus

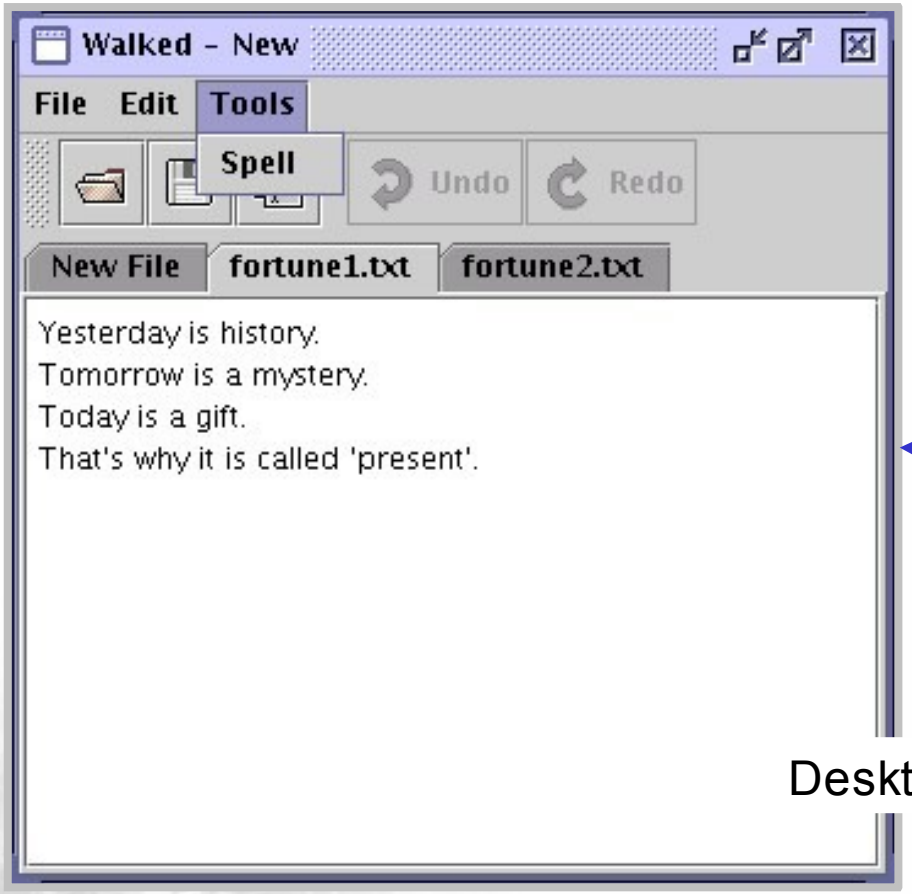
EXEHDA: cenários de utilização por exemplos

WalkEd (Editor de Texto Pervasivo)

Perfil: integração computação móvel e computação distribuída;
Objetivo: funcionalidade e flexibilidade no uso da ferramenta de edição.

Adaptação ao tipo do dispositivo

Adaptação Funcional



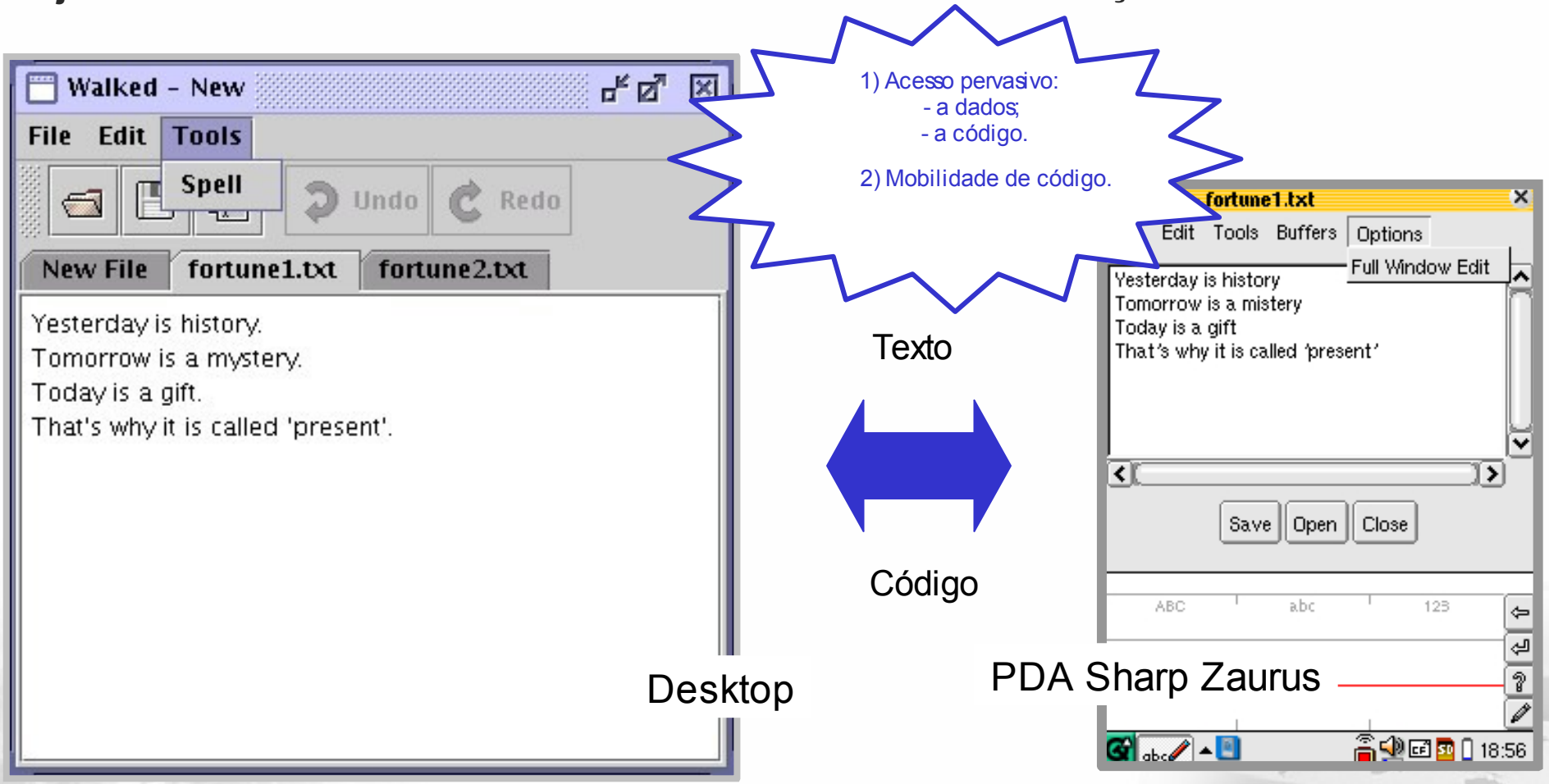
Desktop

PDA Sharp Zaurus

EXEHDA: cenários de utilização por exemplos

WalkEd (Editor de Texto Pervasivo)

Perfil: integração computação móvel e computação distribuída;
Objetivo: funcionalidade e flexibilidade no uso da ferramenta de edição.



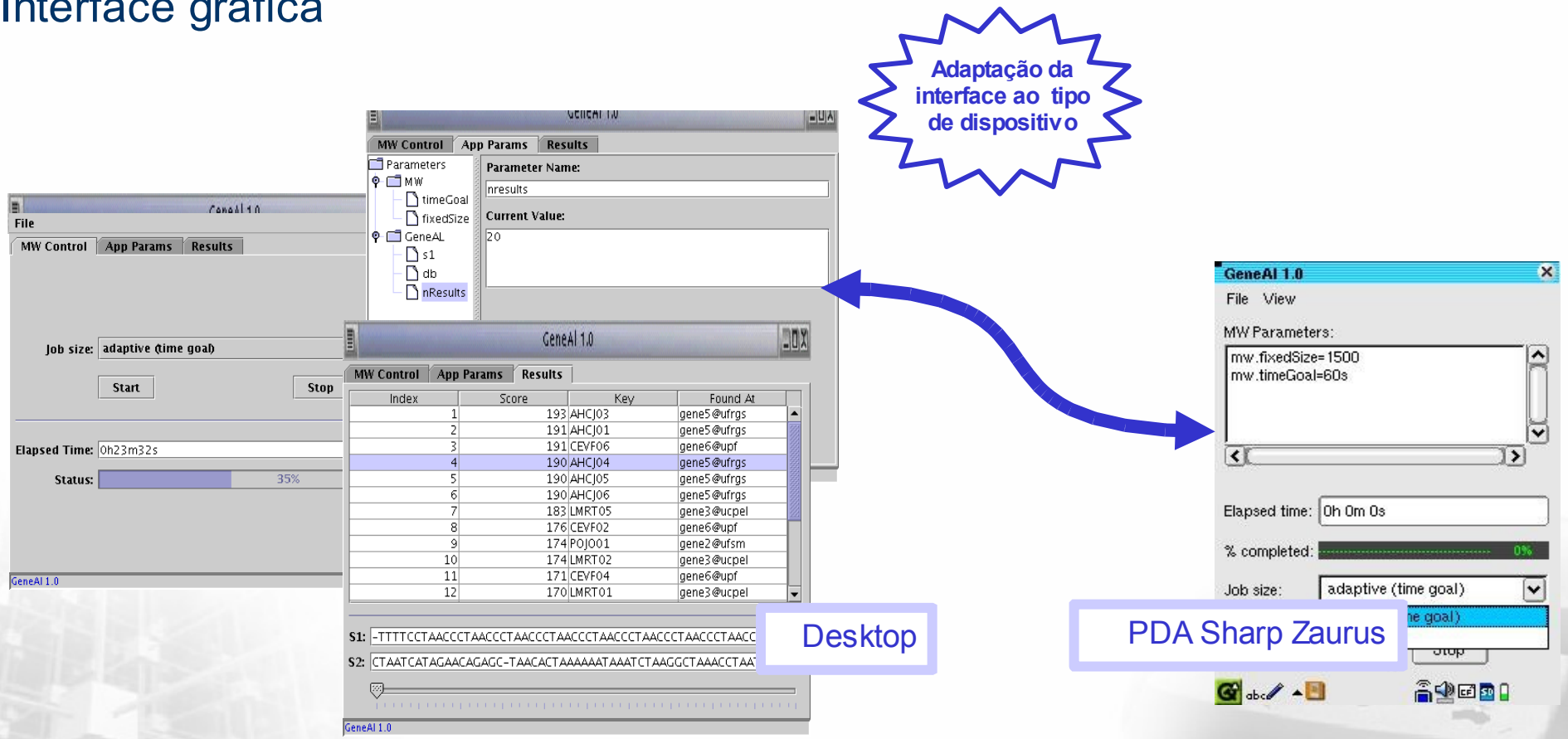
EXEHDA: cenários de utilização por exemplos

Aplicação GeneAI

Perfil: execução distribuída multi-institucional;

Objetivo: busca de desempenho e flexibilidade na interação.

Interface gráfica



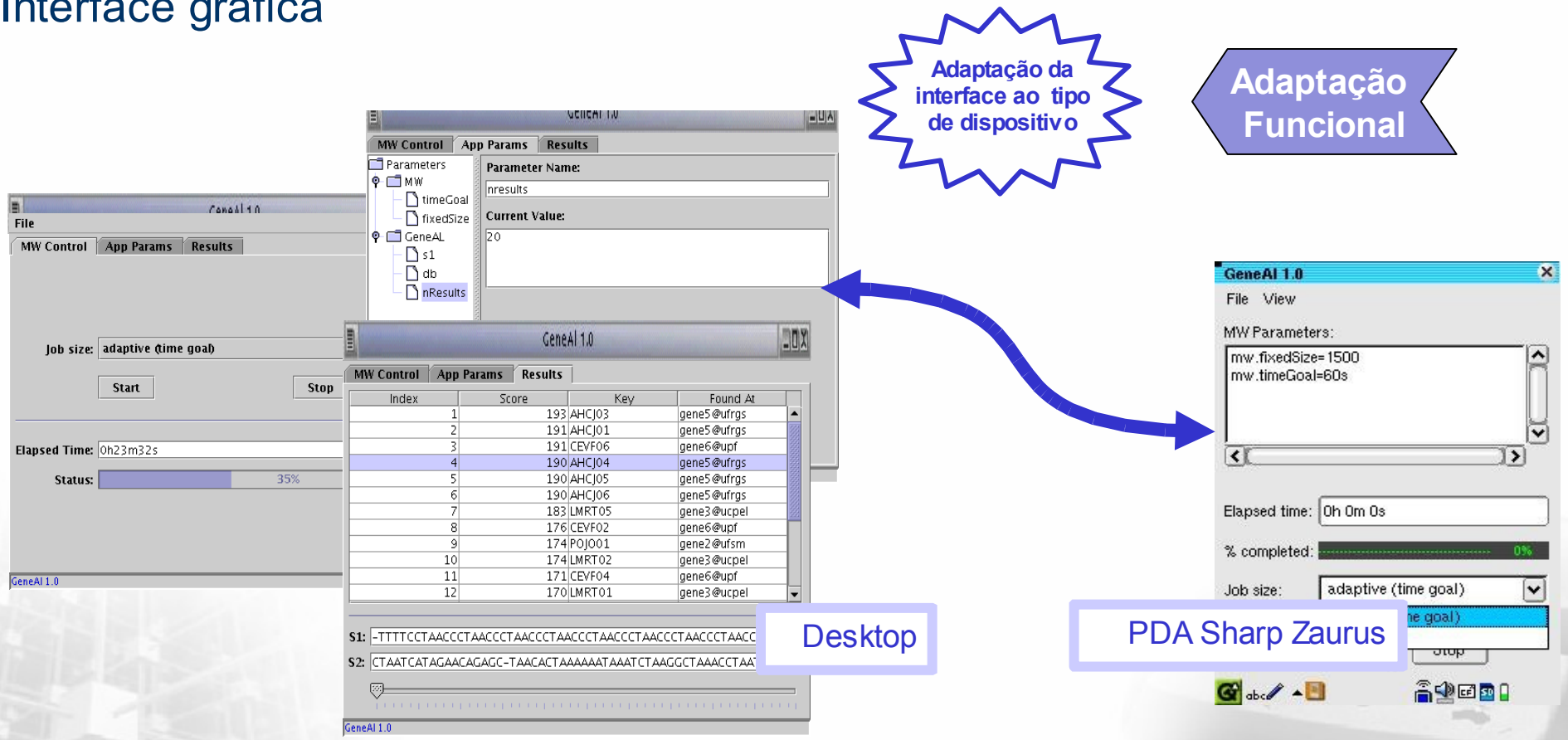
EXEHDA: cenários de utilização por exemplos

Aplicação GeneAI

Perfil: execução distribuída multi-institucional;

Objetivo: busca de desempenho e flexibilidade na interação.

Interface gráfica

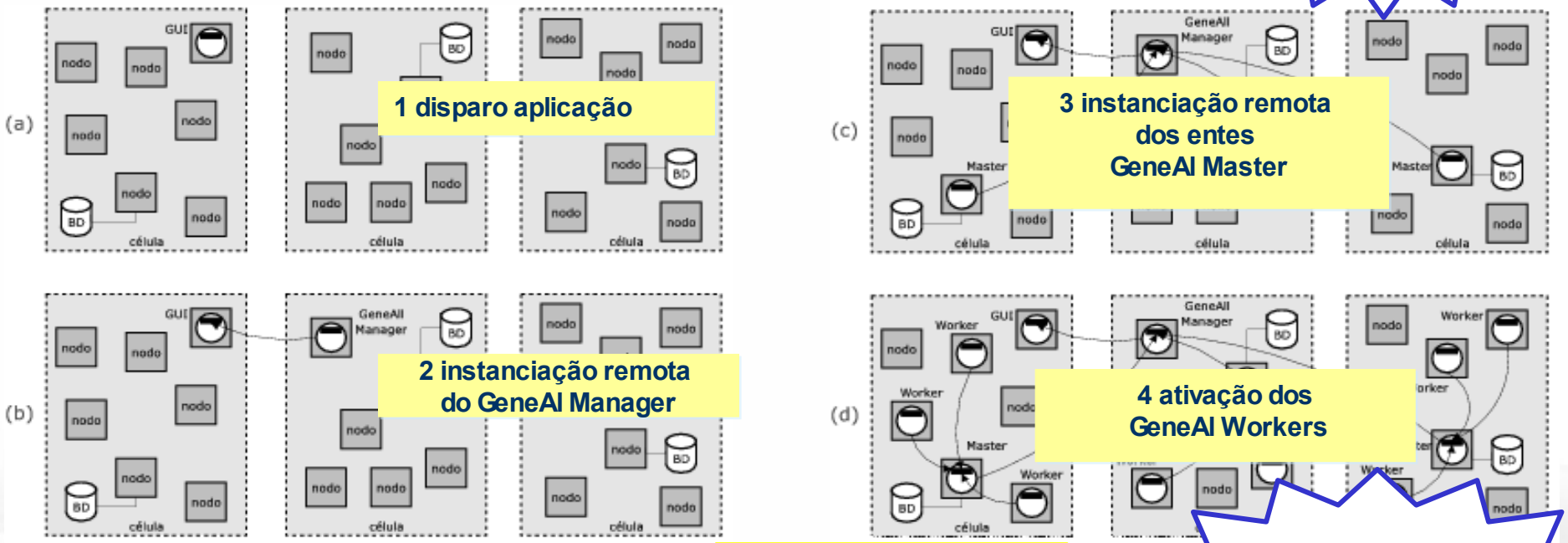


EXEHDA: cenários de utilização por exemplos

Aplicação GeneAI

Descoberta de recursos: localização de banco de genes em ambiente multi-institucional

Comportamento da distribuição

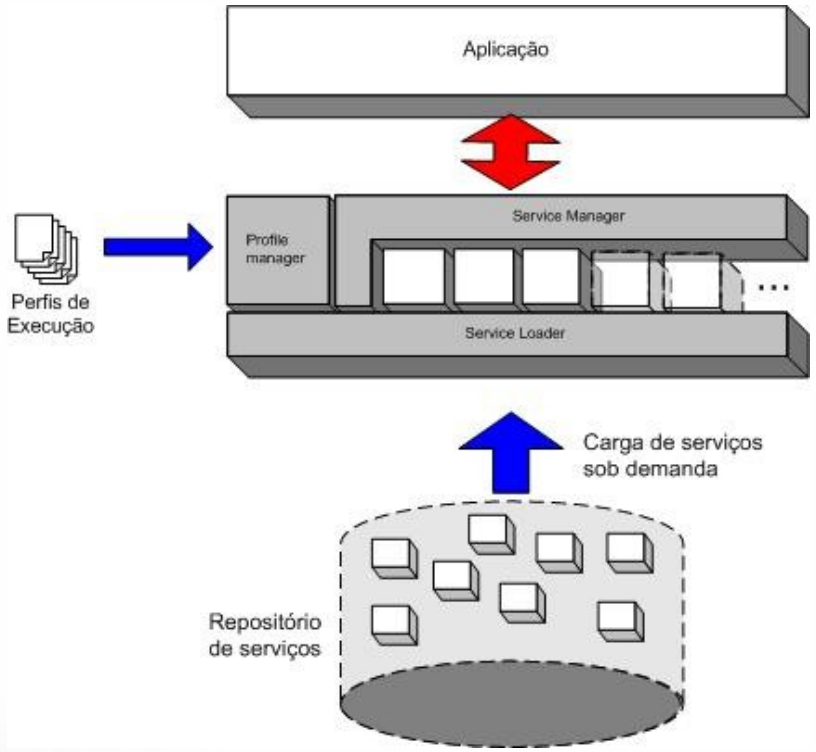


Adaptação não-funcional (escalonamento)

Mobilidade e instanciação remota de código: os componentes a aplicação são ativados em nodos distribuídos

Funcionalidades do EXEHDA

Organização em Núcleo Mínimo



Serviços Adaptativos:

Perfis (*profile*) definem qual instância do serviço será usada em um dispositivo específico através do processo de adaptação em tempo de carga

O serviço pode adaptar-se também considerando o contexto dinâmico como estado da conectividade através do processo de adaptação em tempo de execução.

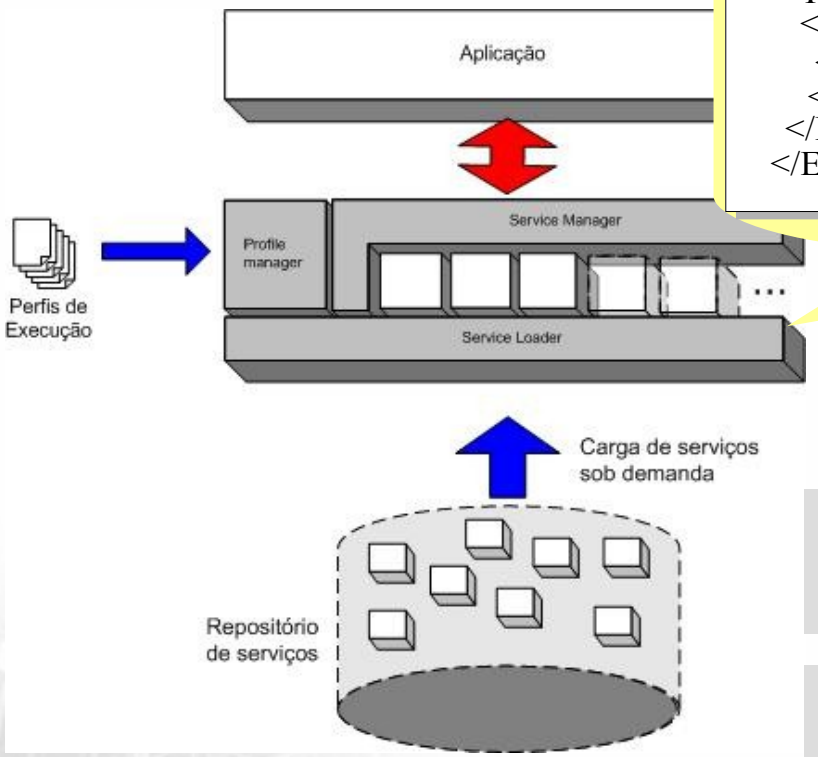
Supre os requisitos de:

- **Adaptação ao contexto (tipo de recurso);**
- **Economia de recursos (instalação sob demanda).**

API Tip: a classe `org.isam.Exehda` é o ponto de acesso para os serviços do middleware.

Funcionalidades do EXEHDA

Organização em Núcleo Mínimo




```
<EXEHDA>
 <PROFILE name="profileName">
  <SERVICE name="sName" impl="className" loadPolicy="boot"|"demand">
   <PROP name="paramName" value="paramValue" />
  </SERVICE>
 </PROFILE>
</EXEHDA>
```

Supre os requisitos de:

Adaptação ao contexto (tipo de recurso);

Economia de recursos (instalação sob demanda).

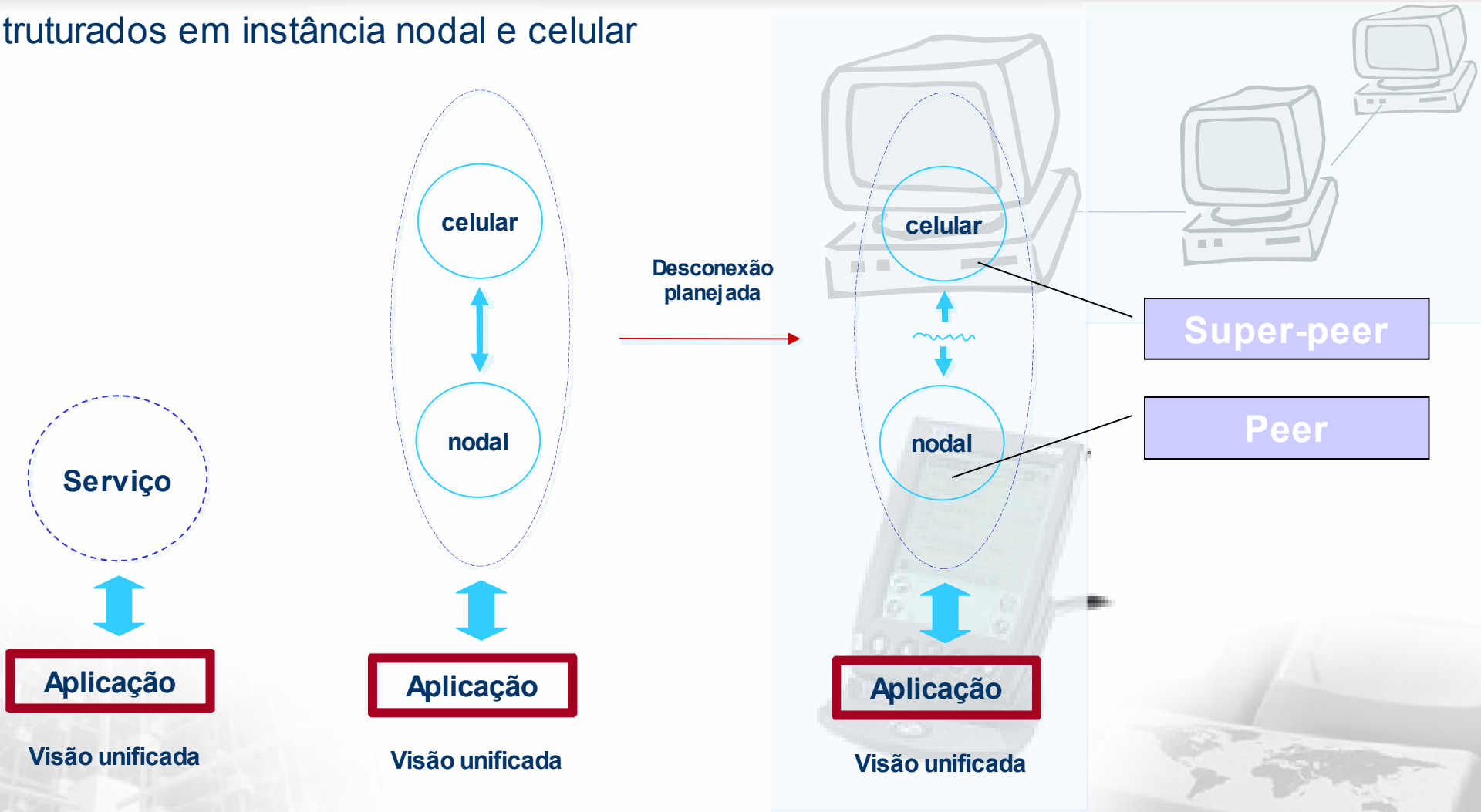
API Tip: O método `Exehda.getService(String)` permite obter uma referência para um serviço a partir de seu nome (definido no profile).

API Tip: as interfaces dos serviços do middleware encontram-se definidas na package `org.isam.exehda.services`

Funcionalidades do EXEHDA

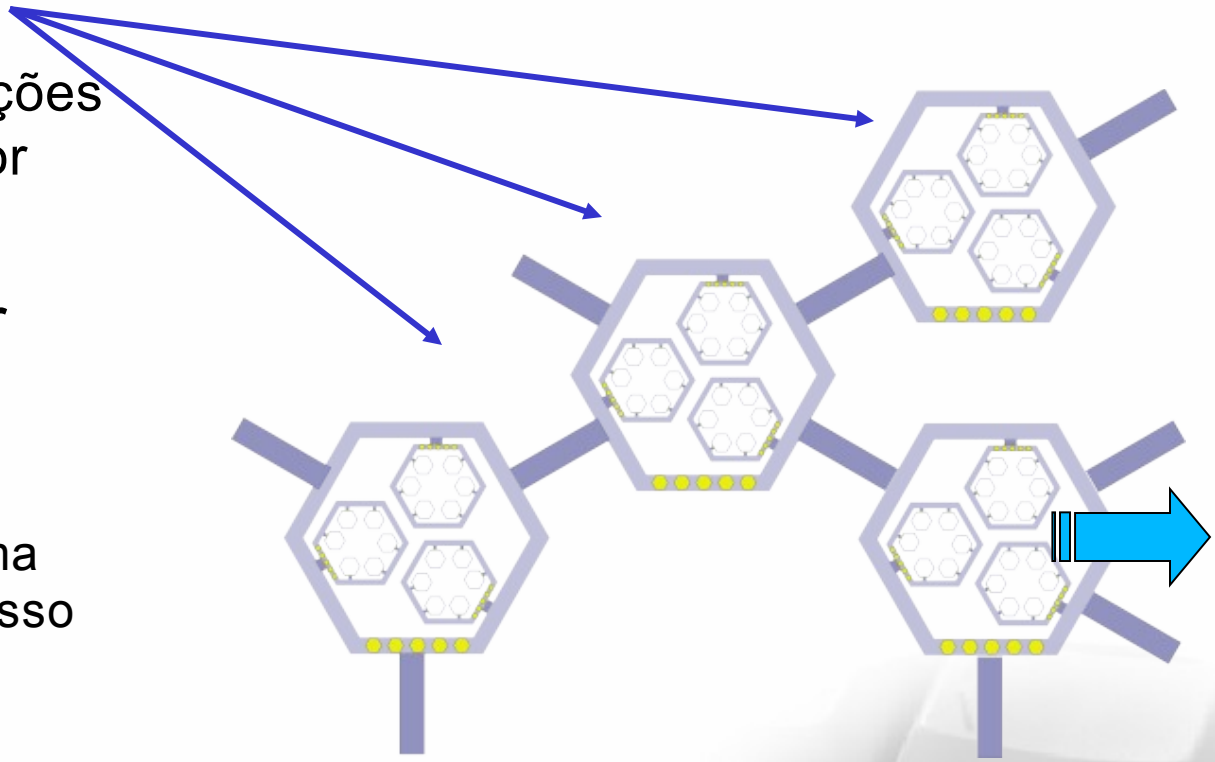
Operação desconectado: serviços em duas instâncias

Estruturados em instância nodal e celular



Organização do Ambiente Pervasivo

- Organização Super-peer
  - Rede P2P de células
  - Célula centraliza requisições originadas no seu interior

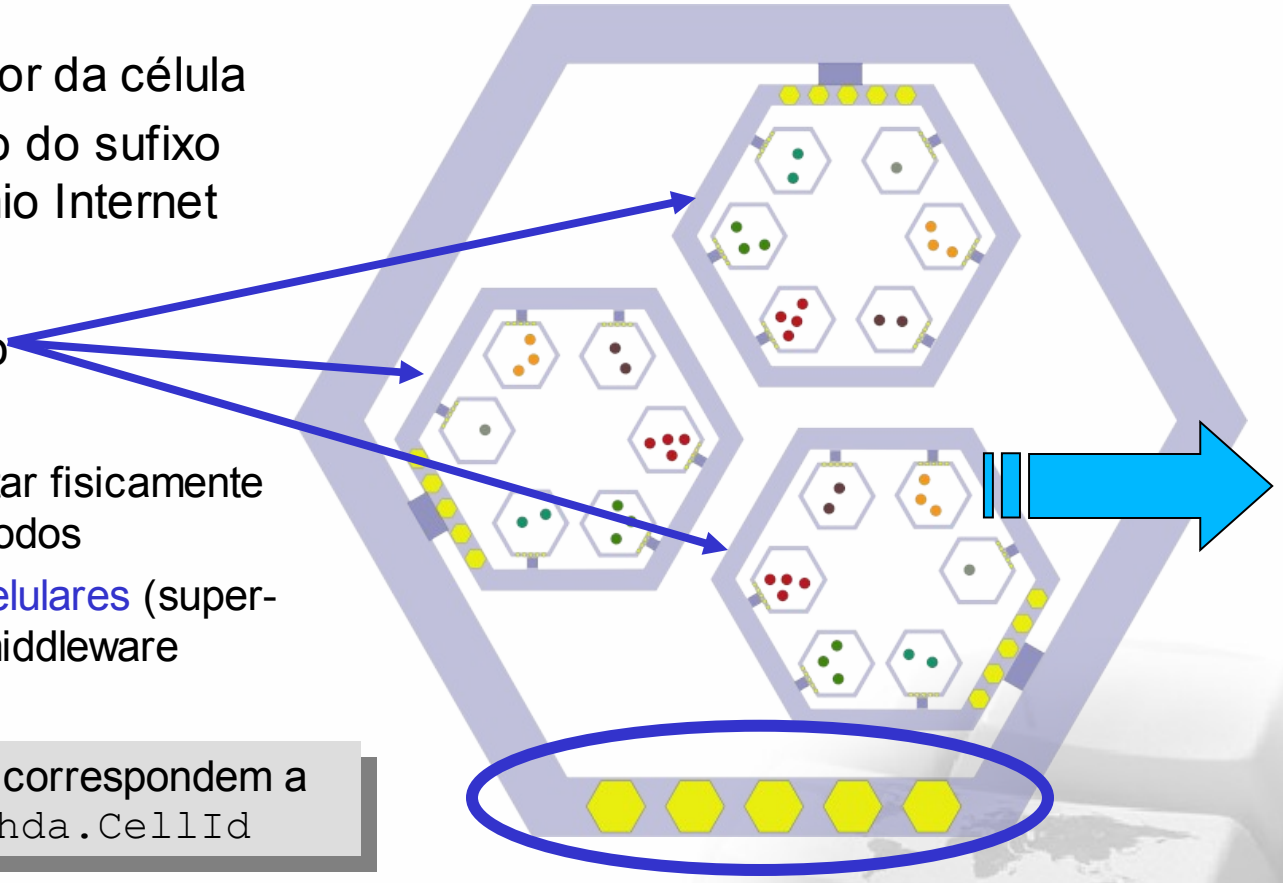


- Células são formas por critérios de afinidade:
  - Administrativos
    - Recursos sujeitos a uma mesma política de acesso
  - Lógica (localidade)
    - proximidade física
    - velocidade de interconexão entre os nodos

Organização do Ambiente Pervasivo

A Célula

- Cada célula é identificada por um ID único no ISAMpe
  - Definido pelo administrador da célula
  - Usualmente, um derivativo do sufixo DNS associado ao domínio Internet
- Constituição da Célula
  - Nodos de processamento
  - Base da célula
    - Entidade virtual, pode estar fisicamente distribuída entre vários nodos
    - Hospeda as instâncias celulares (super-peers) dos serviços do middleware

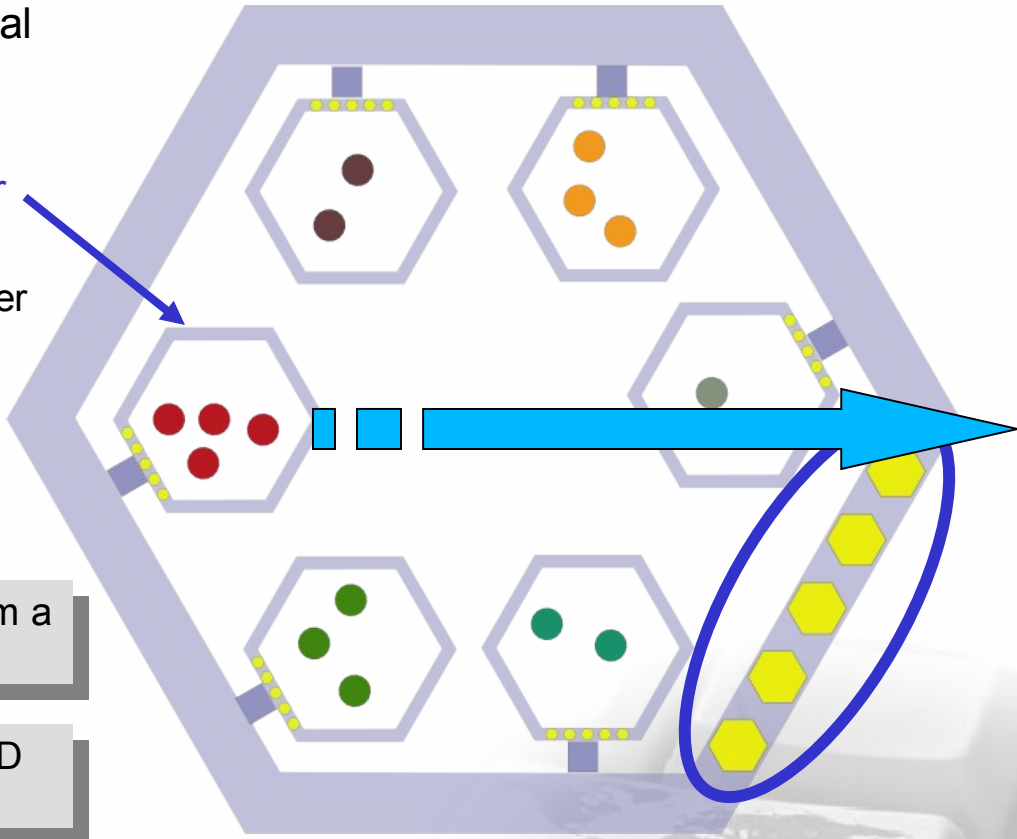


API Tip: os identificador de célula correspondem a objetos da classe `org.isam.exehda.CellId`

Organização do Ambiente Pervasivo

Os Nodos de Processamento

- Cada nodo é identificado por ID numérico único no escopo da célula
  - ID local + ID célula formam um ID único global para o nodo
- Composição do Nodo
  - cada aplicação é executada em um container
  - Instâncias peer dos serviços
    - Intermediam acessos as instâncias super-peer
    - Viabilizam a operação desconectada do nodo
    - compartilhadas entre containers de aplicação



Serviços (peer) do midleware

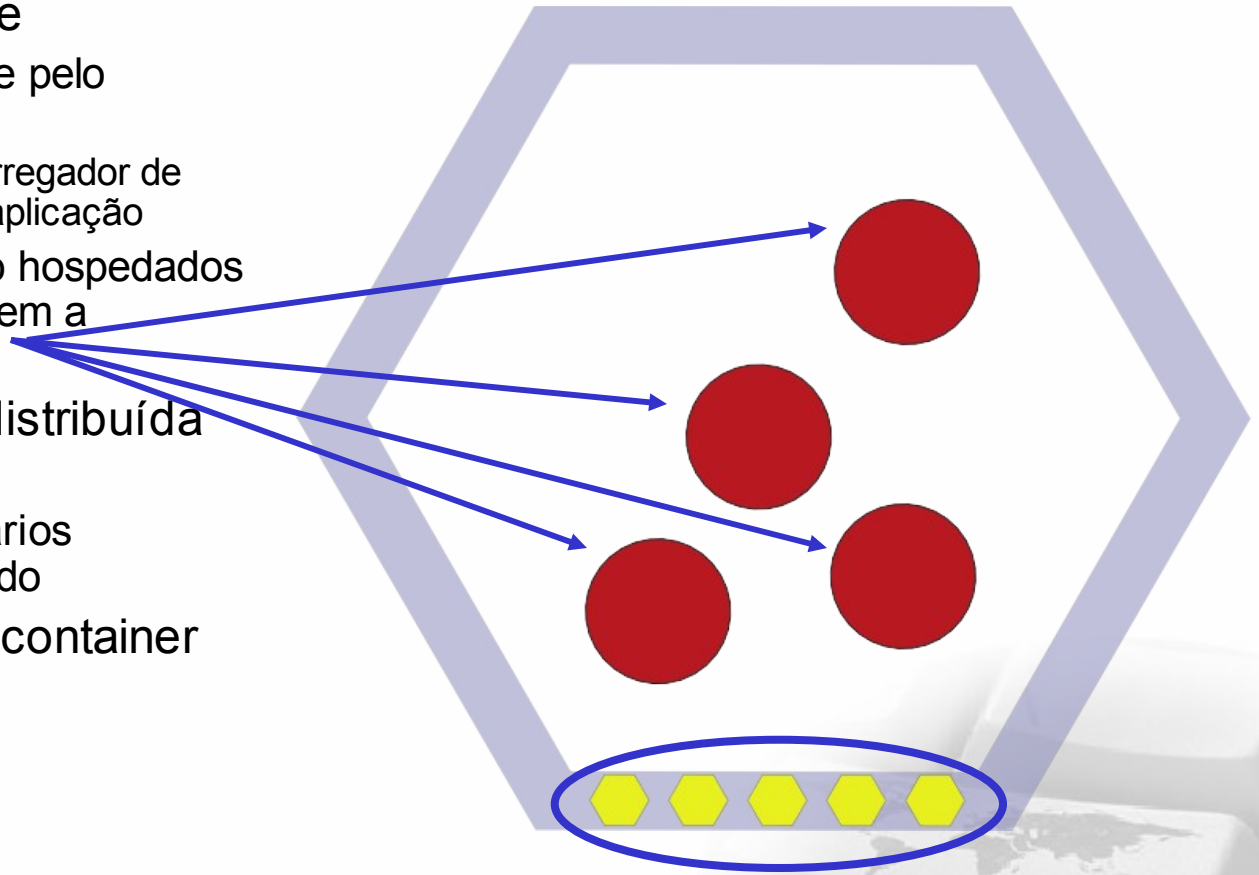
API Tip: os identificadores (globais) de nodo correspondem a objetos da classe `org.isam.exehda.HostId`

API Tip: o método `HostId.getLocalHost()` retorna o ID global do nodo local

Organização do Ambiente Pervasivo

Containers de Aplicação

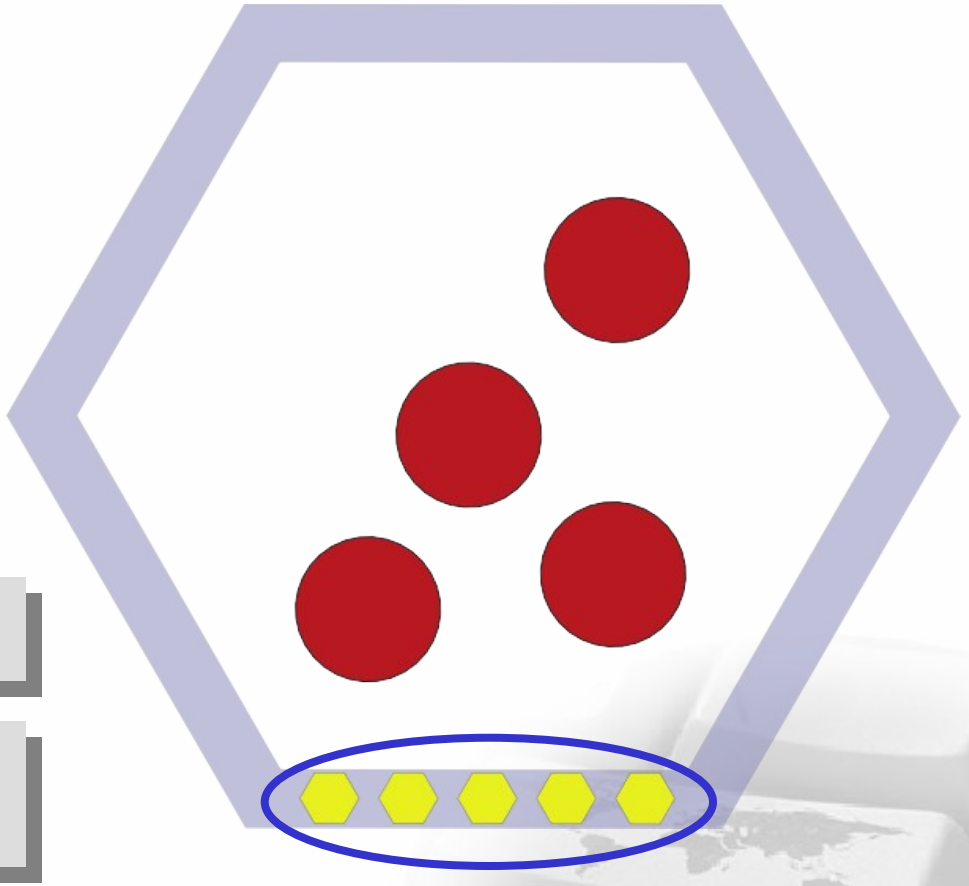
- Os container não recebem identificadores, i.e. não podem ser referenciados explicitamente
  - São alocados implicitamente pelo middleware
    - O container define um carregador de código específico para a aplicação
  - Dentro de um container são hospedados os objetos (OX) que compõem a aplicação
- Uma aplicação pode estar distribuída em diversos nodos
  - Nessa situação, existirão vários containers, um em cada nodo
- Atualmente, o único tipo de container suportado é o Java
  - Futuro: python...



Organização do Ambiente Pervasivo

Containers de Aplicação

- Cada OX é identificado unicamente no escopo do ISAMpe por um ID hierárquico
  - ID local objeto + ID nodo local
- A princípio, todos os OX representam entidades isoladas de computação
  - Mecanismos de interação presentes no middleware
    - Base de Informações da célula
    - Meta-atributos dos OX
    - Referencias para invocação remota de métodos
    - Espaços de tuplas



API Tip: os identificadores (globais) de OX correspondem a objetos da classe `org.isam.exehda.ObjectId`

API Tip: os OX são criados utilizando-se o método `createObject()` do serviço `Executor`, a qual retorna o `ObjectID` do OX instanciado

- **Interagindo com Serviços do Middleware**
  - Procedimento Geral
- **Casos de Uso**
  - Instanciação Remota e Migração de Objetos
  - Monitoramento
  - Adaptação ao Contexto

Interagindo Com Serviços do Middleware

Procedimento Geral

```
// núcleo do middleware e conceitos centrais
import org.isam.exehda.*;

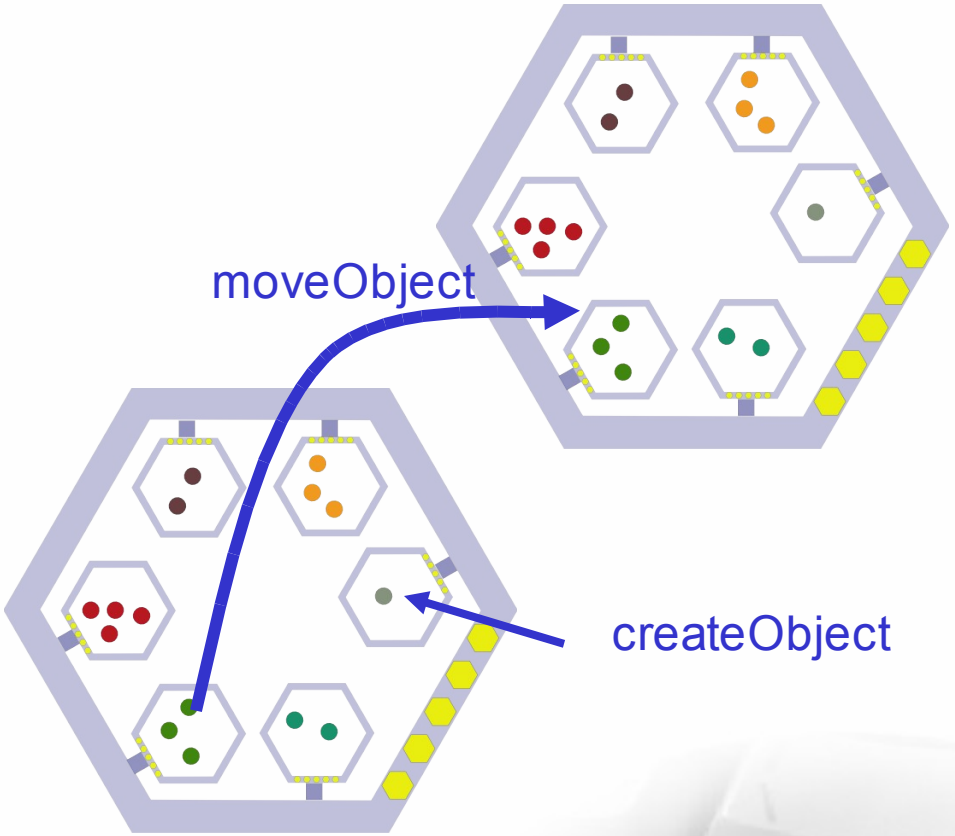
// importa classes relativas ao serviço de interesse
import org.isam.exehda.services.ServiceX;
import org.isam.exehda.services.ServiceX.*;

// obtem referência para o Service
ServiceX service = Exehda.getService(ServiceX.SERVICE_NAME);

// invoca métodos do serviço
res = service.op(...);
```

Serviço Executor

- **Define métodos para:**
  - Disparar e terminar aplicações;
  - Criar e migrar objetos (*OXes*);
  - Configurar heurísticas de escalonamento por aplicação
    - Guia a seleção de *host* tanto na instanciação como na migração remotas dos objetos.
- **Aspetos operacionais**
  - Noção de um *Activator* por objeto
  - Instanciação através do *Design Pattern* Strategy



Serviço Executor API

Instânciação Remota e Migração de Objetos

API createObject(Class c,Object[] p, Executor.Activator a,Object hint): ObjectId *definição*

Cria um novo objeto (OX) na aplicação, potencialmente em outro nodo

API moveObject(ObjectId ox,Object hint): void

Migra um OX existente para outro nodo

API setHeuristic(Executor.SchedulingHeuristic sh): void *definição*

Configura a heurística de escalonamento a ser usada nos métodos acima. Ela provê a interpretação para o parâmetro *hint*

Ciclo de vida da aplicação

API startApplication(org.isam.util.xml.XmlElement launchingDesc): void

Inicia a execução da aplicação representada pelo descritor XML fornecido

API exitApplication(): void

Faz a aplicação abortar sua execução, parando todas as suas threads

Serviço Executor
API: Tipos Auxiliares

- **interface Executor.Activator**

API activate(ObjectId oxID,Object obj, MarshaledOX extState):void

API deactivate(ObjectId oxID,Object obj, MarshaledOX oxState): void

Responsável por conectar o objeto ao seu ambiente de execução

- E.g. identificar contexto e reconfigurar o objeto em função da disponibilidade de determinados recursos

Utilizado para carregar e inicializar qualquer outro recurso extra antes do objeto iniciar sua execução

- E.g conexão a um banco de dados ou dispositivo de hardware; dispara threads auxiliares

Especificado quando o OX é instanciado, está ligado ao objeto

- Carregado com o OX mesmo que haja migração
- Caso seja omitido, o sistema procurará um *Activator* que sirva para o objeto

- **interface Executor.SchedulingHeuristic**

API chooseCreationHost(String cls,Object[] params,Object hint,Vector toAvoid): HostId

API chooseMigrationHost(ObjectId oxID,Object hint,Vector toAvoid): HostId

Serviço Executor API (*cont.*)

Operações auxiliares para a migração

API deactivateObject(ObjectId ox): Executor.MarshaledOX

Desativa um OX pela invocação da operação correspondente de seu *activator*;

Retorna uma representação do objeto suspenso passível de ser transferida a outro nodo

API activateObject(Executor.MarshaledOX oxState): void

Re-ativa um OX previamente desativado a partir da representação persistente provida e da invocação da operação correspondente de seu *activator*

Outras operações definidas pelo serviço Executor

API currentApplication(org.isam.util.xml.XmlElement launchingDesc): ApplicationId

Retorna o ID global da aplicação corrente, indiretamente, identifica o container da aplicação no nodo local

API runAction(ApplicationId app, Runnable action): void

Executa a ação informada no contexto de execução (i.e., container, classloader, permissões) da aplicação selecionada.

NOTA: Apenas serviços do middleware podem utilizar essa operação para atuar em containers de outras aplicações que a corrente (retonada por currentApplication()).

- **Exemplo de uma aplicação do serviço Executor**
  1. Instancia um objeto (classe Walker) em um nodo passado na linha de comando,

     Após a instanciação o objeto precisa ser ativado
  2. Posteriormente, permanece em um laço migrando o objeto entre dois outros nodos também informados na linha de comando
     - O objeto instanciado precisa ser desativado e reativado quando da migração, ou seja, sua thread é interrompida e re-iniciada a cada nova migração

Classe Walker

Implementa as interfaces Runnable e Serializable

Atributos:

- String name – nome do objeto
- String where – localização do objeto
- int count – contador de passos da execução do objeto

Métodos

- run() – a cada 2 segundos, imprime a localização do objeto, enquanto a Thread não é interrompida
- setWhere(String where) – seta a localização do objeto

Classe WalkerActivator

Activador da classe Walker

Ativação:

```
public void activate(ObjectId oxID,Object obj, MarshaledOX extState)
```

- Seta localização do objeto
- Cria uma *thread* para o objeto
- Inicia a execução da *thread*

Desativação:

```
public void deactivate(ObjectId oxID,Object obj, MarshaledOX extState)
```

- Interrompe a *thread*
- Espera término da *thread*

main(String[] args)

Recebe como parâmetro três identificadores de Host (*HostId*)

Cria instância do serviço *Executor:*

```
Executor executor = (Executor)
                        Exehda.getService(Executor.SERVICE_NAME);
```

Cria objeto no primeiro Host fornecido como parâmetro:

```
ObjectId walkerID =
            executor.createObject(
                Walker.class,                    // Classe a ser instanciada
                new Object[] { "Walker#0" }, // Argumentos para o construtor
                new WalkerActivator(),        // Ativator
                h1);                                  // Host
```

Migra o objeto entre o segundo e o terceiro host indefinidamente:

```
while (true) {
        sleep(10000);
        executor.moveObject(walkerID, h2);
        System.out.println("Moved to "+h2);

        sleep(10000);
        executor.moveObject(walkerID, h3);
        System.out.println("Moved to "+h3);
}
```

Método sleep(long millis): faz a thread corrente parar sua execução durante os milesegundos fornecidos

Exercício 1: Projetar um “agente móvel” Web Cache.

Comportamento

O agente aceita conexões em um porta padrão (3980), lendo requisições HTTP.

- Utiliza então a classe URL de Java para fazer download da página e retorna o conteúdo para o cliente.
- O agentem mantém uma cache em disco das páginas recentemente visitadas

Periodicamente, o agente é migrado entre hosts, como no exemplo da classe Walker.

Restrições

O servidor socket deve ser liberado no nodo origem e realocado no nodo destino. Essa operação fica a cargo do activator do objeto.

As 10 páginas mais visitadas devem ser migradas com o agente, entretanto não como atributos do objeto.

Dica: utilizar os atributos estendidos disponibilizados na classe MarshaledOX (no activator)

Exercício 2: Projetar um “agente móvel” FileFinder

Comportamento

O Agente recebe como parâmetro um string, um padrão de nome de arquivo (e.g. *.txt) e uma lista de nodos.

Retorna como saída a lista de todos os arquivos encontrados cujo nome é compatível como padrão fornecido cujo conteúdo contém o string informado que tenha sido modificado na última hora.

Restrições

A saída final é mostrada no nodo origem

A última hora da modificação tem como referência o relógio local da nodo sendo pesquisado

Em cada nodo pesquisado, a lista parcial identificada localmente deve ser mostrada na tela usando algum editor de texto nativo disponível na plataforma (e.g. notepad se windows, gedit ou kwrite se linux).

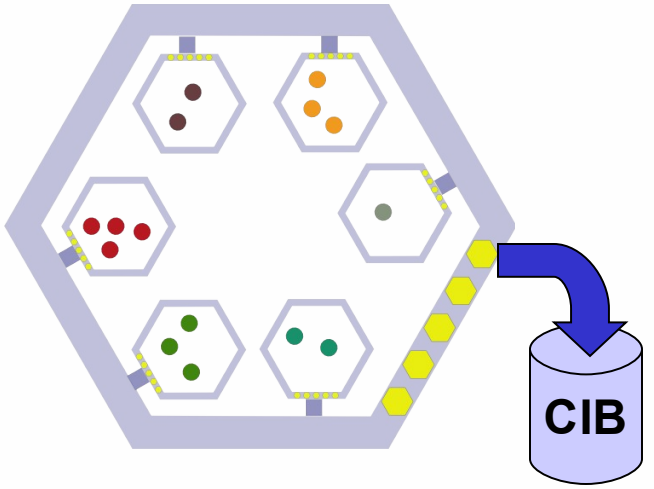
Dica: usar o activator para adaptar o agente a característica da plataforma.

Exemplos de Utilização da API

- **Distribuindo componentes**
  Serviço Executor
- **Utilizando o catálogo de recursos**
  Serviço CellInformationBase
- **Observando diretamente o ambiente de execução**
  Serviço Collector
- **Percebendo o contexto**
  Serviço ContextManager

Serviço CellInformationBase

- **Define métodos para:**
  - Registrar e remover recursos;
  - Consultar e modificar atributos dos recursos cadastrados
  - Pesquisar por recursos que disponibilizem determinados atributos



- **Aspetos operacionais**
  - Organização hierárquica em Namespaces tipados
  - Operações de matching são relativamente restritas e busca em larga escala não muito otimizada
    - Funcionalidade complementado pelo serviço **Discoverer**

cell:ufrgs = {att1=v1; att2=v2...}
- **service:discover** = {contactAddress=...}
- **host:0x01A3** = {att1=v1; att2=v2...}
  - **service:discover** = {contactAddress=...}
  - **service:executor** = {contactAddress=...}
- **host:0x05B7** = {att1=v1; att2=v2...}

Serviço CellInformationBase API

Gerenciamento de Recursos

API addResource(ResourceName r): void

Adiciona um recurso ao catálogo

API removeResource(ResourceName r): void

Remove um recurso do catálogo

API getAttribute(ResourceName r, String att): String

Consulta o valor de um atributo do recurso

API setAttribute(ResourceName r, String att, String val): String

Modifica o valor de um atributo do recurso

API Tip: objetos correspondentes a identificadores (globais) `XXXId` (e.g. `HostId`, `CellId`, `ObjectId`) podem ser facilmente transformadas em objetos `NameSpace` ou `ResourceName` utilizando-se os métodos `toNameSpace()` e `toResourceName()` respectivamente.

Pesquisa

API select(SelectionConstraint filter, int maxResults): ResourceName[]

Retorna os recursos catalogados que satisfazem o critério de restrição especificado

API selectByType(String type, NameSpace ns, int maxResults): ResourceName[]

API selectByName(String name, NameSpace ns, int maxResults): ResourceName[]

Atalhos para dois tipos de pesquisa considerados freqüêntes

Exercício 3: Registrar o “agente móvel” Web Cache no catálogo de recursos do célula, de forma que ele possa ser localizado por potenciais clientes

Comportamento

Modifique a solução do Exercício 1, o agente registre-se na célula como sendo um recurso do tipo ‘webcache’ e cadastre um atributo ‘contactAddress’ que corresponde a seu <ip,porta> atual

Restrições

O atributo contactAddress deve ser atualizado a cada migração do Agente.

Dica: Implemente essa fucnionalidade também no activator do agente

.

EXEHDA - API

Almost In a Nutshell

PARTE - II

Casos de Uso da API GRADEp

- **Distribuindo componentes**
  Serviço Executor
- **Observando diretamente o ambiente de execução**
  Serviço Collector
- **Percebendo o contexto**
  Serviço ContextManager

- **Monitoração e sensores**

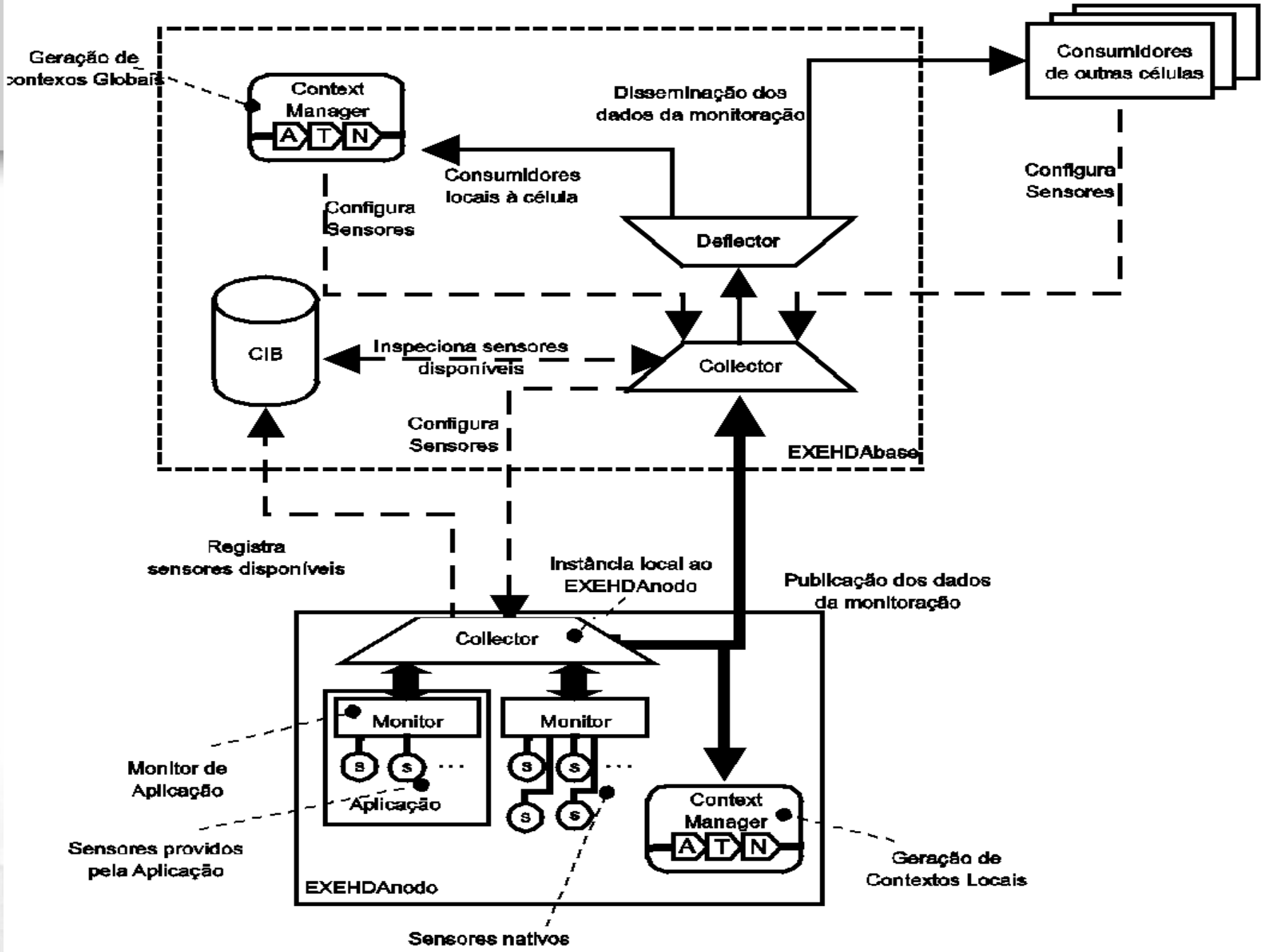
  As aplicações suportadas pelo middleware podem executar com comportamentos diferentes de acordo com o ambiente em que estão inseridas, visando adaptação.

  As aplicações conscientes de contexto suportadas pelo middleware dependem, direta ou indiretamente, da informação extraídas por "sensores" para perceberem o ambiente e tomarem decisões

  Exemplos de sensores:

  - Ocupação atual da CPU (%)
  - Memória livre (bytes)
  - ...

  Os sensores disponibilizados pelo serviço *Collector* são utilizados para alimentar o serviço *ContextManager*


Geração de
contexos Globais
Context
Manager
A T N
Disseminação dos
dados da monitoração
Consumidores
de outras células
Consumidores
locais à célula
Configura
Sensores
Configura
Sensores
Deflector
CIB
Inspeciona sensores
disponíveis
Collector
Configura
Sensores
EXEHDAbase
Registra
sensores disponíveis
Instância local ao
EXEHDAnodo
Publicação dos dados
da monitoração
Collector
Monitor
Monitor
S
Aplicação
Monitor de
Aplicação
Sensores providos
pela Aplicação
Context
Manager
A T N
Geração de
Contextos Locais
EXEHDAnodo
Sensores nativos

- **Coleta de dados de sensores: visão da aplicação**

  A coleta de informações de sensores é feita pelo serviço Coletor ("Collector") local de um nodo.

  Obtendo uma referência para o coletor:

  - ```
    Collector collector = (Collector)
    Exehda.getService(Collector.SERVICE_NAME);
    ```

  A partir do coletor, a aplicação pode consultar sensores de duas formas:

  - **Consulta eventual** ("*poll*"): através do Coletor, realiza uma consulta direta a um sensor específico;
  - **Assinatura** (modelo "publicador-assinante"): um objeto **consumidor** (da aplicação) registra, junto ao Coletor, interesse em determinados sensores e recebe uma chamada quando o Coletor detecta uma mudança de valores.
    ```

Visão geral

- **Coletor gerencia 0 ou mais monitores**
- **Cada monitor exporta N sensores para o coletor**
  - Monitor nativo (Solaris, Linux)
  - Monitor da aplicação
  - Monitor do runtime Java
- **Para expandir o serviço coletor, o desenvolvedor deve implementar a interface Monitor e registrar o seu serviço monitor junto ao Collector**
  - As aplicações devem utilizar a classe AppMonitor ao invés de implementar a interface Monitor devido a aspectos de segurança
- **Coletor e consumidores:**
  - Um coletor possui um conjunto de consumidores que estão interessados no estado dos sensores

- **Estrutura da Aplicação Exemplo 1**
  1. *Obtém credencial de consumidor da monitoração*
  2. *Lista todos os sensores disponíveis*
  3. *Habilita, na sua visão da monitoração, todos os sensores disponíveis*
     1. *Só nesse momento aquela informação passará a ser produzida*
  4. *Periodicamente realiza uma operação de polling, atualizado o valor corrente do sensor na interface gráfica da aplicação*

CollectorDemo – imports

CollectorDemo.java: (trechos selecionados)

```
// Importa Coletor, Sensor, MonitoringData (valor de retorno
// que contém dados monitorados) e ConsumerId
// (identificador de consumidor registrado)
import org.isam.exehda.Exehda;
import org.isam.exehda.services.Collector;
import org.isam.exehda.services.Collector.Sensor;
import org.isam.exehda.services.Collector.MonitoringData;
import org.isam.exehda.services.Collector.ConsumerId;
```

CollectorDemo – variáveis

```
public class CollectorDemo /* ... */ {
    // o serviço Coletor local
    private Collector collector;
    // lista de sensores conhecidos pela aplicação (retornada pelo coletor)
    private Sensor[ ] sensors;
    // identificador de consumidor desta aplicação
    private ConsumerId id;
    // painel (interface gráfica) que exibe o valor dos sensores
    private Panel sensorsPane;
    // mapeia um nome de sensor para o componente de interface gráfica
    // (etiqueta de texto - label) responsável pela sua exibição
    private Hashtable labelsBySensorName;
```

CollectorDemo – main()

```
// ponto de entrada da aplicação
public static void main( String[] args ) throws Exception {
    // 1. inicialização
    CollectorDemo frame = new CollectorDemo();

    /* ... aqui mais inicialização da interface ... */

    // 2. Consulta periódica a sensores (uma vez por segundo)
    for ( int i=0; i<100; i++) {
        Thread.currentThread().sleep(1000);
        frame.updateSensors();
    }
}
```

Próximo slide...

CollectorDemo – Construtor

```
public CollectorDemo() // construtor ----------
{
    // ... aqui inicializa interface gráfica ...

    // Obtém referência para o coletor local
    collector = (Collector) Exehda.getService(Collector.SERVICE_NAME);

    // Registra aplicação como consumidora e obtém ConsumerId
    // (fornece "null" como referência de interface de callback e, portanto,
    // a aplicação não será chamada pelo Collector)
    id = collector.addConsumer( null );
```

Próximo slide...

Collector.addConsumer()

- O método addConsumer() da interface Collector registra um novo consumidor para o coletor do nodo:

  Retorna um objeto ConsumerId (identificador do consumidor);

  - ConsumerId é usado depois para escolher os sensores de interesse do consumidor.

  Recebe como argumento um objeto que implementa a interface MonitoringConsumer

  - O MonitoringConsumer é chamado depois quando um sensor de interesse do consumidor fornece dados de interesse do consumidor.
  - Se for passado **null** neste parâmetro, o consumidor deverá fazer consulta periódica aos sensores pois não receberá callbacks

API

```
public interface Collector {
  public ConsumerId addConsumer(MonitoringConsumer c);
```

CollectorDemo – Construtor (cont.)

```
// obtém referência para a lista de sensores disponíveis no coletor
sensors = collector.getSensors();   // retorna array:   Sensor [ ]

// "Este" consumidor (this.id) habilita todos os sensores disponíveis.
// Necessário para que este consumidor possa depois consultar
// qualquer um dos sensores.
for ( int i=0; i< sensors.length; i++ ) {
  collector.setSensorEnabled(  this.id ,  sensors[i] ,  true  );
}
} // ---------- fim construtor
```

Próximo slide

Collector.setSensorEnabled()

- O método setSensorEnabled() da interface Collector controla a disponibilidade de um sensor para um consumidor específico:

  Argumentos:

  - ConsumerId: ID do consumidor;
  - Sensor: referência ao sensor a ser habilitado ou desabilitado junto ao consumidor especificado;
  - boolean: “true” para habilitar o sensor, “false” para desabilitar.

API

```
public interface Collector {
  public void setSensorEnabled(ConsumerId c, Sensor s, boolean
e);
```

CollectorDemo – de volta ao main()...

```
// .... voltando ao main(), após examinar o construtor ...
public static void main( String[ ] args ) throws Exception {
        // 1. inicialização
        CollectorDemo frame = new CollectorDemo();

        /* ... Inicialização da interface ... */

        // 2. Consulta periódica a sensores (uma vez por segundo)
        for ( int i=0; i<100; i++) {
                Thread.currentThread().sleep(1000); // aguarda 1 segundo
                frame.updateSensors();
        }
}
```

Próximo slide

CollectorDemo – updateSensors()

```
// realiza uma consulta a todos os sensores, exibindo na interface
// o valor mais atualizado de cada sensor
public void updateSensors() {
    // para cada sensor "i"...
    for ( int i=0; i< sensors.length; i++) {
     // consumidor "id" coleta diretamente dados do sensor "i"
          // ("null": objeto MonitoringData pré-alocado - opcional)
     MonitoringData data =
       collector.probeSensor( id , sensors[i] , null );

          // atualiza interface: (nome do sensor, dados atuais do sensor)
          // obs.: cria label para cada nome de sensor sob demanda
          addSensor( sensors[i].getName().getSimpleName() , data.getString() );
    }
}
```

Próximos 2 slides

Nome do sensor

Valor sensorado

Collector.probeSensor()

- O método probeSensor() da interface Collector realiza uma medição pontual de um dado sensor:

  Argumentos:

  - ConsumerId: consumidor que está fazendo a medição;
  - Sensor: sensor desejado;
  - MonitoringData: objeto (opcional) MonitoringData pré-alocado para o valor de retorno;

  Retorna um objeto MonitoringData que contém o valor (dado) sensorado;

API

```
public interface Collector {
  public void probeSensor(ConsumerId c, Sensor s, MonitoringData
d);
```

MonitoringData

- Um objeto do tipo MonitoringData representa um dado de monitoração.
- Métodos de MonitoringData:

API

getSensorName() retorna o ResourceName do sensor que gerou este dado de monitoração;

- Por sua vez, ResourceName.getSimpleName() retorna o "nome simples" do sensor (em formato String).

API

getString(), getInt() e os outros métodos "get" fazem a conversão automática do tipo de dado sensorado para o tipo requisitado (String, Integer, etc).

- **Estrutura da Aplicação Exemplo 2**
  1. *Obtém credencial de consumidor da monitoração*
  2. *Lista todos os sensores disponíveis*
  3. *Habilita, na sua visão da monitoração, todos os sensores disponíveis*
     1. *Só nesse momento aquela informação passará a ser produzida*
  4. *Periodicamente, realiza uma operação de polling, atualizado o valor corrente do sensor na interface gráfica da aplicação*

CollectorDemo2 – imports

CollectorDemo2.java: (trechos selecionados)

```
import org.isam.exehda.Exehda;
import org.isam.exehda.services.Collector;
import org.isam.exehda.services.Collector.Sensor;
import org.isam.exehda.services.Collector.MonitoringData;
import org.isam.exehda.services.Collector.ConsumerId;
// interface de callback – para que a aplicação possar ser
// chamada pelo Collector:
Import org.isam.exehda.services.Collector.MonitoringConsumer;
```

CollectorDemo2 -- membros

```
public class CollectorDemo2 /* ... */
    implements ItemListener,  MonitoringConsumer /* ... */ {
```

Próximo slide...

```
    private Collector collector;  // coletor
    private Sensor[ ] sensors;    // sensores
    private ConsumerId id;        // identificador deste consumer
    // painel que contém um "checkbox" para cada sensor
    private Panel sensorsPane;
    // associa um componente de interface "checkbox" a um
    // sensor (checkbox habilitado significa interesse no sensor)
    private Hashtable sensorsByBox;
```

MonitoringConsumer

- Interface **MonitoringConsumer** é implementada por um objeto que recebe mensagens de atualização de sensores através do método "**update**":

API

```
public interface MonitoringConsumer {
  public void update(long timestamp, MonitoringData[] data);
}
```

- **timestamp**: "Tempo" em que ocorreu a medição;
- **data**: Dados de medição de um ou mais sensores. Além dos dados, também contém uma referência para o(s) sensor(es).

CollectorDemo2 – main()

```
// ponto de entrada da aplicação
public static void main( String[ ] args ) throws Exception {

    // esta demo é mais orientada a eventos:
    //   1. construtor inicializa a aplicação;
    //   2. usuário habilita e desabilita sensores (“checkboxes”);
    //   3. o callback “update()” é invocado pelo serviço Collector para
    //      atualizar os sensores que estão atualmente selecionados.
    CollectorDemo2 frame = new CollectorDemo2();

    frame.pack();
    frame.setVisible(true);
}
```

Construtor – próximo slide

CollectorDemo2 -- Construtor

```
public CollectorDemo2() {
    /* ... inicializa interface ... */

    collector = (Collector) Exehda.getService(Collector.SERVICE_NAME);

    // registra um novo consumidor e especifica esta instância de
    // CollectorDemo2 como objeto de callback do consumidor
    //   (  o objeto de callback implementa o método update() da
    //       interface MonitoringConsumer )
    this.id = collector.addConsumer(  this  );

    // este método cria, associa e exibe um "checkbox" para cada sensor
    initSensors();
}
```

CollectorDemo2 – itemStateChanged()

```
// método especificado pela interface ItemListener, invocado
// quando a "checkbox" associada a um sensor é ativada ou
// desativada pelo usuário
public void itemStateChanged(ItemEvent ev) {
    // obtém Sensor associado à "checkbox" que foi clicada
    Checkbox cb = (Checkbox) ev.getSource();
    Sensor s = (Sensor) sensorsByBox.get(cb);

    // habilita ou desabilita o sensor junto a este consumidor ("id")
    // (se a checkbox foi habilitada, habilita o sensor, senão desabilita)
    collector.setSensorEnabled(  id ,  s ,
     (ItemEvent.SELECTED == ev.getStateChange())  );
}
```

CollectorDemo2 – update()

```
// update() : para implementar a interface MonitoringConsumer
// informa o consumidor ("assinante") sobre novos valores sensorados
public void update( long timeStamp, MonitoringData[ ] data ) {
   System.out.println("\nTime stamp: "+timeStamp);
   for (int i=0; i<data.length; i++) { // para cada entrada reportada...
      if ( data[i] != null ) {
         // imprime o "nome do sensor" desta entrada do relatório ...
         System.out.print( data[i].getSensorName().getSimpleName() );
         System.out.print(" \t= ");
         // ... e imprime o valor ("sensored value") fornecido por este sensor
         System.out.println( data[i].getString() );
      }
   }
}
```

Serviço Collector: outros métodos da API

- **O método getSensor() da interface Collector é utilizado para buscar um sensor pelo seu nome.**

  Parâmetros:

  - String: nome do sensor

  Retorna uma referência ao Sensor desejado

API

```
public interface Collector {
    public Sensor getSensor(String nome);
```

Serviço Collector: outros métodos da API

- **O método removeConsumer() da interface Collector remove (des-registra) um consumidor junto ao serviço coletor.**

  Parâmetros:

  - ConsumerId: ID do consumidor

API

```
public interface Collector {
    public void removeConsumer(ConsumerId c);
```

Serviço Collector: outros métodos da API

- **O método addMonitor() da interface Collector registra um novo monitor.**

  Parâmetros:

  - Monitor: novo monitor a ser registrado no serviço coletor

API

```
public interface Collector {
    public void addMonitor(Monitor m);
```

- **Interface Monitor**

  Reservada para sensores de sistema

- **Interface AppMonitor estende a interface Monitor**

  Para monitoração no nível da aplicação (sensores da aplicação)

Casos de Uso da API GRADEp

- **Distribuindo componentes**
  Serviço Executor
- **Observando diretamente o ambiente de execução**
  Serviço Collector
- **Percebendo o contexto**
  Serviço ContextManager

Visão geral

- **O serviço ContextManager é responsável pelo refinamento da informação bruta produzida pela monitoração de sensores;**

  Produz informações abstratas (mnemônicas) referentes aos elementos de contexto;

  A linguagem empregada para a definição dos elementos de contexto utiliza o formato XML.

Exemplo: contexto “load”

- EXEMPLO: contexto “load” (“ocupação”)
  - Mapear a ociosidade da CPU do nodo em um valor simbólico para a aplicação que executa neste nodo:
  - Quando o contexto muda (valor sensorado de ociosidade da CPU), a aplicação deve ser notificada, recebendo o novo valor (estado) do contexto.

Elemento de contexto (Mnemônico)	Ociosidade da CPU (%)
free for chat	valor > 60
occupied	60 > valor > 30
do not disturb!	30 > valor

Exemplo: contexto “load” (XML)

```
<CONTEXT n='load'>
 <STATES>
    <STATE n='free for chat'/>
    <STATE n='occupied'/>
    <STATE n='do not disturb!'/>
 </STATES>
 <FILTER syntax='xml:NumericRangeMapping'>
  <INDEX>
    <SENSOR n='CPU_OCCUP_IDLE'/>
  </INDEX>
  <MAPPINGS>
    <RANGE ub='30' state='do not disturb!'/>
    <RANGE lbo='30' ub='60' state='occupied'/>
    <RANGE lbo='60' state='free for chat'/>
    <DEFAULT state='occupied'/>
  </MAPPINGS>
 </FILTER>
</CONTEXT>
```

Elemento de Contexto (estados)

Sintaxe do mapeamento (neste caso, de intervalos numéricos para mnemônicos)

Sensores

Mapeamento
ub = limite superior
lbo = limite inferior

Visão geral

- **No GRADEp a informação de um contexto registrado é produzida por uma cadeia de detecção de contexto de três partes:**

  Agregador (aggregator): responsável pela composição dos dados de um ou mais sensores para a produção de um valor agregado;

  Tradutor (translator): converte um valor agregado em um valor abstrato (mnemônico – equivalente a ``pouca bateria'', ``máquina ociosa'', etc.)

  Notificador (notifier): detecta alterações no dado abstrato fornecido pelo tradutor, disparando então eventos de modificação de contexto.

- **Atualmente são suportados elementos de contexto "mono-sensor" (entrada o dado de monitoração proveniente de um único sensor).**
- **Em desenvolvimento:**

  Suporte a elementos de contexto compostos ou multi-sensor;

  Inclusão de um componente preditor (Predictor) em paralelo com o componente agregador na cadeia de detecção de contexto, para prever mudanças de contexto.

- **Estrutura da Aplicação Exemplo**
  1. Declara dois elementos de contexto
     1. Relativo a ocupação de cpu, que utiliza mapeamento por intervalos numéricos
     2. Relativo a qualidade do display, que utiliza mapeamento por expressões regulares
  2. Registra-se como listener para eventos relacionados aos dois elementos de contexto
  3. Aguarda por eventos de modificação no estado dos elementos de contexto

CtxManagerDemo – imports

CtxManagerDemo.java: (trechos selecionados)

```
// importa ContextManager, Context, ContextState
// e ContextListener
import org.isam.exehda.Exehda;
import org.isam.exehda.services.ContextManager;
import org.isam.exehda.services.ContextManager.Context;
import org.isam.exehda.services.ContextManager.ContextState;
import org.isam.exehda.services.ContextManager.ContextListener;
```

CtxManagerDemo – Contexto “load” (XML)

```
public class CtxManagerDemo {
   static final String CTXDEF = ( ""        // declara o XML do contexto “load” em uma string
      + "<CONTEXT n='load'>\n"
      + "  <STATES>\n"
      + "    <STATE n='free for chat'/>\n"
      + "    <STATE n='occupied'/>\n"
      + "    <STATE n='do not disturb!'/>\n"
      + "  </STATES>\n"
      + "  <FILTER syntax='xml:NumericRangeMapping'>\n"
      + "    <INDEX>\n"
      + "      <SENSOR n='CPU_OCCUP_IDLE'/>\n"
      + "    </INDEX>\n"
      + "    <MAPPINGS>\n"
      + "      <RANGE ub='30' state='do not disturb!'/>\n"
      + "      <RANGE lbo='30' ub='60' state='occupied'/>\n"
      + "      <RANGE lbo='60' state='free for chat'/>\n"
      + "      <DEFAULT state='occupied'/>\n"
      + "    </MAPPINGS>\n"
      + "  </FILTER>\n"
      + "</CONTEXT>\n").replace('\'','"');
```

CtxManagerDemo – Contexto “gui” (XML)

```
// declara o XML do contexto “gui”: tipo de interface de saída disponível para a aplicação
static final String CTXDEF2 = ( ""
   + "<CONTEXT n='gui'>\n"
   + "  <STATES>\n"
   + "    <STATE n='console'/>\n"
   + "    <STATE n='pda'/>\n"
   + "    <STATE n='desktop'/>\n"
   + "  </STATES>\n"
   + "  <FILTER syntax='xml:StringMapping'>\n"
   + "    <INDEX>\n"
   + "      <SENSOR n='runtime.gui.awt.mode'/>\n"
   + "    </INDEX>\n"
   + "    <MAPPINGS>\n"
   + "      <MATCH op='equals' v='none' ignoreCase='true' state='console'/>\n"
   + "      <MATCH op='equals' v='awt-low res' ignoreCase='true' state='pda'/>\n"
   + "      <MATCH op='prefix' v='swing' ignoreCase='true' state='desktop'/>\n"
   + "      <DEFAULT state='console'/>\n"
   + "    </MAPPINGS>\n"
   + "  </FILTER>\n"
   + "</CONTEXT>\n").replace('\'','"');
```

Tags XML: próximo slide...

Contexto “gui” (XML)

```
<CONTEXT n='gui'>
 <STATES>
  <STATE n='console'/>
  <STATE n='pda'/>
  <STATE n='desktop'/>
 </STATES>
 <FILTER syntax='xml:StringMapping'>
  <INDEX>
   <SENSOR n='runtime.gui.awt.mode'/>
  </INDEX>
  <MAPPINGS>
   <MATCH op='equals' v='none' ignoreCase='true' state='console'/>
   <MATCH op='equals' v='awt-low res' ignoreCase='true' state='pda'/>
   <MATCH op='prefix' v='swing' ignoreCase='true' state='desktop'/>
   <DEFAULT state='console'/>
  </MAPPINGS>
 </FILTER>
</CONTEXT>
```

Mapeamento de string: para quando o sensor produz uma string.

Sensor monitora uma propriedade (string) do runtime Java.

Op = operação:
- equals = igualdade
- prefix = prefixo

v = valor (ao qual aplica a operação)

ignoreCase = sensível a maiúsculas?

“Se modo AWT começa com “swing”, é desktop;
Se é “awt-low-res” é PDA;
Se é “none”, então é console texto.”

CtxManagerDemo – main() (1/3)

```
// Ponto de entrada da aplicação
public static void main( String[] args ) throws Exception {

    // Obtém referência para o serviço ContextManager
    ContextManager ctxman = (ContextManager)
            Exehda.getService( ContextManager.SERVICE_NAME );

    // Cria contextos (Context) baseados nos descritores XML. O serviço
    // ContextManager fica responsável por solicitar ao serviço Coletor
    // o registro dos sensores apropriados.
    Context ctx = ctxman.createContext( CTXDEF );      // contexto “load”
    Context ctx2 = ctxman.createContext( CTXDEF2 );  // contexto “gui”
```

API

CtxManagerDemo – main() (2/3)

API

```
// declara uma subclasse (anônima) de ContextListener,
// sobrescrevendo o seu método:
//   void contextChanged(ContextState p, ContextState c);
ContextListener l = new ContextListener() {
  public void contextChanged(ContextState prev, ContextState curr) {
    System.out.println("Context changed from "+prev+" to "+curr);
  }
};

// registra as instâncias dos contextos “load” e “gui” no
   ContextManager,
// associando, a estas instâncias, o objeto ContextListener criado acima.
ctxman.addContextListener( l, ctx );
ctxman.addContextListener( l, ctx2 );
```

Próximo slide

ContextManager.addContextListener()

- O método addContextListener() da interface ContextManager associa um objeto “escutador” (*listener*) a um elemento de contexto:

  Argumentos:

  - ContextListener: objeto que será notificado sobre mudanças no contexto;
  - Context: o contexto em questão.

API

```
public interface ContextManager {
  public void addContextListener(ContextListener l, Context c);
```

CtxManagerDemo – main() (3/3)

```
// aguarda 60 segundos para que seja possível mudar o
// contexto através da modificação dos valores dos sensores
// ( por exemplo, mudança na ocupação de CPU, ...)
System.out.println("Waiting for context events...");
Thread.currentThread().sleep(60000);

System.out.println("Cleaning up...");

// remove uma associação entre um Context e um ContextListener
// ( parâmetros: idem ao addContextListener() )
ctxman.removeContextListener( l, ctx );
ctxman.removeContextListener( l, ctx2 );

// libera o Context do "load"
ctxman.releaseContext( ctx );

} // fim do main() e do demo
```

API

Próximo slide...

ContextManager. releaseContext()

- O método releaseContext() da interface ContextManager notifica o ContextManager de que o contexto não é mais necessário:

  Recebe como argumento o Context que pode ser liberado;

  A liberação de um Context serve para possível liberação de recursos alocados por sensores;

API

```
public interface ContextManager {
  public void releaseContext(Context h);
```

EXEHDA - API

In a Nutshell

Exercícios

- **Descrever (pseudo-código) as seguintes aplicações, fazendo uso dos serviços do middleware**

1. Um agente de software que percebe a ocupação de processador do nodo no qual executa e migra para o próximo de uma lista (previamente conhecida) sempre que o nodo local atinge um determinado nível de carga

2. Um agente de software que interage com o usuário (i.e. apresenta uma pergunta que deve ser respondida pelo usuário) de formas diferentes da qualidade do display. Caso o dispositivo não tenha display gráfico, migra para o próximo dispositivo de uma lista (previamente conhecida)