Sensibilidade ao Contexto em Java

Daniel Coutinho e Marco Túlio de Oliveira Valente

Departamento de Ciência da Computação
Pontifícia Universidade Católica de Minas Gerais
danielcm@gmail.com,mtov@pucminas.br

Resumo. *Descreve-se neste artigo uma plataforma de middleware, chamada J2CS (Java 2 Context Service), a qual encapsula e abstrai diversos interesses inerentes ao desenvolvimento de aplicações sensíveis ao contexto em Java. A plataforma J2CS pode ser classificada como um middleware simples, flexível, extensível e baseado em componentes. O artigo descreve e discute a arquitetura e a interface de programação de J2CS. Apresentam-se também alguns exemplos de aplicações sensíveis ao contexto baseadas no middleware proposto.*

Abstract. *In this paper, we describe a middleware system, called J2CS (Java 2 Context Service), that provides support to most of the tasks involved in designing context-aware applications in Java. The J2CS system can be classified as a lightweighted, flexible, extensible and component oriented middleware. In this paper, we discuss the software architecture and the programming interface of J2CS. We also present some examples of context-aware applications based on the proposed middleware.*

1 Introdução

Atualmente, são cada vez mais comuns ambientes saturados de dispositivos computacionais, incluindo computadores de mesa, computadores de mão, computadores vestíveis (*wearable computers*), equipamentos eletrônicos, sensores, atuadores etc. Tais ambientes estão transformando em realidade o que Mark Weiser denominou há cerca de 15 anos de computação ubíqua [14, 15], também chamada de terceira onda da computação. Segundo esta classificação, a primeira onda foi marcada pelos computadores de grande porte (*mainframes*) e a segunda pelos computadores pessoais. Cada uma destas ondas favoreceu o surgimento de novos sistemas computacionais, os quais não existiam anteriormente. Assim, *mainframes* viabilizaram a automatização dos mais diversos sistemas de informação corporativos; da mesma forma, computadores de mesa popularizaram diversas aplicações de produtividade pessoal. No caso de computação ubíqua, espera-se que seja cada vez mais freqüente o desenvolvimento de *aplicações sensíveis ao contexto* (*context-aware applications*) [4, 10].

Uma aplicação é dita sensível ao contexto se a mesma utiliza informações sobre o contexto de entidades relevantes ao seu domínio para prover informações ou serviços a seus usuários. Tipicamente, informações de contexto incluem a localização, identidade ou estado de pessoas ou objetos físicos ou computacionais. Por exemplo, um serviço de impressão poderia fornecer a um usuário móvel a possibilidade de sempre imprimir por *default* na impressora que estivesse mais próxima dele. Um telefone celular poderia sempre passar para o modo silencioso caso o usuário estivesse em uma reunião (isto é, caso sensores de presença indicassem que o usuário se encontra na sala de reuniões de sua

empresa e caso sensores de som indicassem um determinado nível de ruído na mesma). Como último exemplo, em um sistema de correio eletrônico, poderia ser possível enviar uma mensagem apenas para usuários que se encontrassem fisicamente em um certo ambiente (por exemplo, em um laboratório de uma universidade).

O desenvolvimento de aplicações sensíveis ao contexto pode se beneficiar diretamente de uma infra-estrutura de *middleware* [4, 7, 8, 1, 6]. Fundamentalmente, esta infra-estrutura deve se responsabilizar pela captura, interpretação e disseminação de informações de contexto. Para capturar informações, a plataforma deve interagir com os mais diversos provedores de contexto, incluindo sensores, dispositivos e sistemas computacionais. A plataforma deve ser ainda capaz de interpretar e inter-relacionar as informações de contexto coletadas. Por exemplo, para detectar o andamento de uma reunião em uma sala, devem ser relacionadas informações providas por sensores de presença e de ruído. Por fim, a plataforma deve ser capaz de disseminar eventos inferidos a partir de informações capturadas no ambiente. O objetivo maior, como usual em plataformas de *middleware*, é simplificar e tornar mais produtivo o desenvolvimento de aplicações sensíveis ao contexto, delegando para a camada de *middleware* diversos interesses inerentes à construção destas aplicações.

Neste artigo, descreve-se o sistema J2CS (*Java 2 Context Service*), um sistema de *middleware* extensível para desenvolvimento e execução de aplicações sensíveis ao contexto em Java. O projeto de J2CS se inspirou no padrão arquitetural normalmente adotado por sistemas de *middleware* baseados em componentes, como EJB [12] e CCM [11]. A arquitetura destes sistemas é centrada na idéia de *containers*, os quais são responsáveis por gerenciar o ciclo de vida de componentes e disponibilizar serviços não-funcionais aos mesmos, tais como persistência, transações, segurança, tolerância a falhas etc. Outro conceito fundamental nestas arquiteturas são os chamados servidores de aplicação, aos quais cabe hospedar e executar *containers* e seus respectivos componentes.

J2CS pode ser classificado como um *middleware* simples, flexível, extensível, baseado em componentes e direcionado para o desenvolvimento de aplicações sensíveis ao contexto. Em J2CS, *containers* são responsáveis por executar aplicações denominadas *contextlets*, às quais cabe interpretar e disseminar informações de contexto. Além de hospedarem *contextlets*, *containers* são responsáveis por interagir de forma contínua com os mais diversos tipos de dispositivos computacionais encarregados de prover informações de contexto. Estas informações são repassadas para os *contextlets* assinantes das mesmas. Cabe aos *contextlets* interpretar as informações coletadas e notificar aplicações sensíveis ao contexto da ocorrência de eventos de seus interesses. O sistema J2CS é flexível e extensível porque permite que *contextlets* sejam dinamicamente instalados e removidos de seus *containers*, sem que estes tenham que ser reiniciados.

O restante deste artigo encontra-se organizado conforme descrito a seguir. Na Seção 2, apresentam-se os principais requisitos e serviços que devem ser providos por uma plataforma de *middleware* para desenvolvimento de aplicações sensíveis ao contexto. Na Seção 3, são descritos os principais componentes da plataforma J2CS, incluindo uma descrição do papel destes componentes no sistema e uma descrição da interface de programação dos mesmos. Na Seção 4, apresentam-se alguns exemplos de sistemas sensíveis ao contexto construídos usando-se J2CS. A Seção 5 descreve uma avaliação experimental do sistema. A Seção 6 discute trabalhos relacionados e a Seção 7 conclui o artigo.

2 Sistemas de Middleware para Sensibilidade ao Contexto

Plataformas de *middleware* para desenvolvimento de aplicações sensíveis ao contexto devem, em última análise, prover suporte às diversas tarefas relacionadas com o tratamento de contexto em ambientes de computação ubíqua. Considerando o ponto de vista de um desenvolvedor de aplicações sensíveis ao contexto, tais plataformas devem prover pelo menos os seguintes serviços [4, 7, 8]:

- Captura e interpretação de contexto: aplicações sensíveis ao contexto devem ser capazes de tratar informações de contexto sem ter que se preocupar com detalhes de baixo nível referentes a comunicação com sensores e outros dispositivos computacionais. Indo um pouco mais além, aplicações sensíveis ao contexto devem apenas reagir e interpretar eventos que sejam próprios de objetos de seu domínio funcional. Por exemplo, uma aplicação pode não estar interessada em ser notificada toda vez que houver uma mudança na localização de um usuário, mas apenas quando o mesmo estiver localizado em um determinado ambiente. Assim, a camada de *middleware* deve ser capaz de sintetizar eventos típicos do domínio da aplicação a partir da interpretação, agregação e associação de eventos de mais baixo nível, capturados de sensores e outros dispositivos computacionais. Para inferência destes eventos de mais alto nível, a plataforma pode se valer dos mais diversos mecanismos, desde programas em uma linguagem imperativa tradicional até programas baseados em técnicas de inteligência artificial.
- Comunicação: componentes responsáveis pela captura, interpretação e consumo de informações de contexto normalmente estão distribuídos em diferentes nodos de uma rede. Assim, uma plataforma de *middleware* para desenvolvimento de aplicações sensíveis ao contexto deve prover abstrações adequadas para que estas entidades possam se comunicar e coordenar. Como exemplo de tais abstrações, podemos citar chamadas remotas de métodos/procedimentos, espaços de tuplas, eventos etc.
- Reconfiguração dinâmica: uma plataforma de *middleware* para desenvolvimento de aplicações sensíveis ao contexto deve permitir que componentes responsáveis pela captura de contexto, componentes responsáveis pela interpretação de contexto e componentes consumidores de contexto possam ser dinamicamente adicionados e removidos do sistema. Mais especificamente, esta incorporação não deve requerer recompilação do núcleo do sistema, nem mesmo uma interrupção nos serviços providos pela plataforma.
- Serviço de localização: devido ao dinamismo que caracteriza um ambiente de computação ubíqua, os componentes de uma aplicação sensível ao contexto usualmente não conhecem a localização dos demais componentes com quais devem interagir. Por este motivo, plataformas de *middleware* para tais ambientes devem prover um serviço flexível de localização de recursos, que não exija que clientes conheçam em tempo de implantação a localização dos serviços dos quais fazem uso [13].
- Disponibilidade: uma plataforma de *middleware* responsável pela captura, interpretação e disseminação de contexto deve ser executada de forma contínua, pois não se sabe *a priori* quando uma aplicação sensível ao contexto será ativada e passará a demandar seus serviços.

3 J2CS: Arquitetura e Interface de Programação

Conforme mostra a Figura 1, os principais componentes do sistema J2CS são os seguintes:

- Monitores de Contexto: são aplicações que interagem com uma fonte responsável por prover informações de contexto. Uma fonte de contexto pode ser tanto um *hardware* (um sensor, por exemplo) como um sistema computacional (uma aplicação financeira, por exemplo). Cabe ao monitor de contexto publicar eventos toda vez que o estado da fonte de contexto sofrer uma alteração relevante. Por exemplo, um sensor de temperatura pode gerar um evento toda vez que a temperatura ambiente sofrer uma alteração.
- *Containers* J2CS: são aplicações que agem como intermediárias entre monitores de contexto e aplicações sensíveis ao contexto. Cabe a um *container* capturar eventos de mais baixo nível gerados por monitores de contexto e propagá-los para aplicações encarregadas de interpretar os mesmos, as quais são chamadas de *contextlets*. Servidores encarregados de prover um serviço de *container* são chamados de servidores J2CS.
- *Contextlets*: são componentes que encapsulam detalhes referentes à interpretação de contexto. Estes componentes transformam eventos de mais baixo nível detectados por monitores de contexto em eventos de mais alto nível, os quais são disseminados para aplicações sensíveis ao contexto.
- Aplicações sensíveis ao contexto: são aplicações que se beneficiam diretamente da arquitetura de *middleware* proposta. As mesmas devem apenas tratar eventos de mais alto nível.

Os componentes da arquitetura são descritos com mais detalhes nas subseções seguintes.

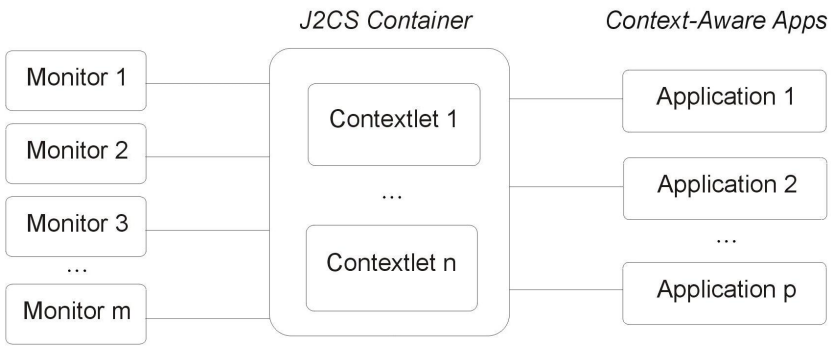

Figura 1. Principais Componentes da Arquitetura do Sistema J2CS

3.1 Monitores de Contexto

Monitores de contexto são associados a uma fonte de contexto e são responsáveis por encapsular detalhes de comunicação entre esta fonte e um *container* J2CS. São usados tanto para publicar assincronamente informações de contexto como para responder a requisições provenientes de *containers*.

Monitores de contexto possuem um conjunto de propriedades estáticas, as quais são usadas para identificar suas características principais. Por exemplo, um sensor de temperatura pode possuir as seguintes propriedades estáticas: seu identificador (obrigatório

em todo monitor), sua localização, seu modelo, de quanto em quanto tempo ele mede a temperatura, a unidade de temperatura usada etc. Além disso, monitores de contexto podem publicar propriedades dinâmicas do ambiente no qual estão instalados. Por exemplo, um sensor de temperatura pode publicar informações como a temperatura medida e a data da medição. Por fim, monitores de contexto podem responder a requisições provenientes de *containers*. Tais requisições são realizadas por um *container* quando este necessita obter informações sobre o estado (isto é, as propriedades dinâmicas) de um determinado monitor.

Registro e Publicação de Eventos: Monitores de contexto são objetos ativos capazes de responder às mensagens descritas a seguir. A mensagem `bind(StaticData)` é usada para registrar um monitor junto a um *container*, informando como parâmetro um objeto contendo as propriedades estáticas do monitor. A mensagem `unbind(String)` é usada para cancelar o registro de um monitor de contexto junto a um *container*. A mensagem `publish(DynamicData)` é usada para publicar uma alteração no contexto do monitor. Nesta mensagem, informa-se como parâmetro um objeto contendo as novas propriedades dinâmicas do contexto monitorado. Por fim, as mensagens `getState()` e `setState(DynamicData)` são usadas para recuperar e atualizar o estado dinâmico do contexto monitorado. A primeira mensagem é enviada, normalmente, por um *container* e a segunda por uma fonte de contexto.

3.2 ContextLets

Um *contextlet* é um objeto que é dinamicamente instalado em um servidor J2CS. Um *contextlet* recebe informações de contexto publicadas por monitores de contexto, interpreta estas informações e eventualmente gera eventos de mais alto nível para aplicações sensíveis ao contexto.

Instanciação e Instalação: *Contextlets* devem estender a classe abstrata `ContextLet`. Todo *contextlet* possui um nome, o qual é passado como parâmetro do construtor desta classe. Após serem instanciados em uma determinada estação da rede, *contextlets* são usualmente movidos para um *container*, usando para isso o seguinte método:

```
ContextLetHelper.install(ContextLet c,
                         ContextLetMonitorBinding[] m)
```

Este método recebe como parâmetro uma referência `c` para o *contextlet* que será instalado no *container* e uma especificação `m` dos monitores de contexto dos quais este *contextlet* deseja receber eventos. De forma semelhante a que ocorre com sistemas de agentes móveis, o estado de um *contextlet* é transferido junto com o mesmo. Já a execução de um *contextlet* em um *container* inicia-se pelo seu método `onArrival()`. Por fim, uma estação da rede pode requisitar a remoção de um *contextlet* de um *container*. Para isso, basta que seja invocado o método `ContextLetHelper.retract(String)`, fornecendo o nome do *contextlet* a ser removido como parâmetro.

Associação entre ContextLets e Monitores de Contexto: Um *contextlet* pode estar associado a um ou mais monitores de contexto. Neste caso, este *contextlet* será notificado de todos os eventos publicados pelos referidos monitores. Para associar um *contextlet* a um determinado monitor de contexto, basta especificar este monitor por meio de um

objeto do tipo `ContextLetMonitorBinding` e usar este objeto como argumento do método `ContextLetHelper.install`. Objetos deste tipo possuem dois atributos: o nome da classe que especifica as propriedades estáticas do monitor de contexto ao qual se deseja associar o *contextlet* e uma expressão booleana usada para selecionar instâncias desta classe. Esta seleção é feita com base nas propriedades estáticas destes monitores. Assim, os operandos usados nesta expressão podem ser atributos da classe que denota as propriedades estáticas de um monitor de contexto. Pode-se ainda usar os operadores lógicos e relacionais existentes em Java.

O conjunto de monitores de contexto associados a um *contextlet* é dinâmico. Em tempo de execução, monitores podem ser removidos deste conjunto caso o registro dos mesmos junto ao *container* seja cancelado (por meio do método `unbind`). Além disso, um novo monitor pode ser dinamicamente associado a um *contextlet* caso o mesmo venha a ser registrado posteriormente à instalação do *contextlet* no *container*. Evidentemente, esta associação somente ocorrerá caso o novo monitor atenda aos critérios definidos pelos objetos do tipo `ContextLetMonitorBinding` do referido *contextlet*.

Consultas a Monitores de Contexto: Um *contextlet* pode obter uma lista de referências para monitores de contexto por meio dos seguintes métodos providos pelo *container*:

```
ContextMonitor[] getMyContextMonitors();
ContextMonitor[] getContextMonitors(ContextLetMonitorBinding);
```

O primeiro método retorna uma lista de referências para os monitores de contexto associados ao *contextlet*. Já o segundo método retorna uma lista de referências para os monitores definidos pelo objeto `ContextLetMonitorBinding`, os quais não necessariamente precisam ser exatamente os mesmos monitores de contexto associados ao *contextlet* para fins de notificação de eventos. Em ambos os casos, as referências retornadas podem ser usadas para se consultar o estado dos monitores de contexto referenciados (por meio do método `getState`, conforme definido na Seção 3.1).

Notificação de Eventos: Suponha que um *contextlet* C esteja associado a um monitor de contexto M que publica eventos do tipo $E1$ e $E2$ (os quais devem ser subtipos de `DynamicData`, conforme definido na Seção 3.1). Este *contextlet* deverá então implementar dois métodos: `void update(E1 e1)` e `void update(E2 e2)`. Toda vez que M publicar uma informação coletada no seu contexto de execução, o *container* se encarregará de rotear esta informação até o respectivo método `update` do *contextlet* C. Portanto, J2CS utiliza uma linguagem para assinatura de eventos baseada em tipos [5].

Interpretação e Publicação de Eventos: Cabe aos métodos `update` de um *contextlet* interpretar, agregar e correlacionar informações de contexto, de forma a gerar eventos de mais alto nível do que aqueles recebidos de monitores de contexto. Para publicar um evento de mais alto nível para aplicações sensíveis ao contexto associadas a um determinado *contextlet*, deve-se chamar o método `publish`, passando como parâmetro um objeto de uma subclasse de `ContextLetData`.

3.3 Aplicações Sensíveis ao Contexto

Aplicações sensíveis ao contexto são beneficiárias diretas dos serviços de captura, interpretação e publicação de eventos providos pela plataforma J2CS. Em geral, estas aplicações estão interessadas em: serem notificadas de eventos de mais alto nível gerados por *contextlets* e, eventualmente, consultar o estado de um *contextlet*.

Assinatura de Eventos: Para assinar eventos gerados por um *contextlet*, uma aplicação deve usar o seguinte método:

```
subscribe(String,ContextLetEventHandler)
```

Este método recebe como parâmetro o nome do *contextlet* do qual se deseja receber eventos e um objeto do tipo `ContextLetEventHandler`, usado pelo *contextlet* para notificar a aplicação, via um *callback*. Para cancelar a assinatura de eventos gerados por um *contextlet*, a aplicação deve usar o método `unsubscribe(String)`, informando como parâmetro o nome do *contextlet*.

Notificação de Eventos: Como afirmado no item anterior, aplicações sensíveis ao contexto, ao assinarem um evento, informam um objeto que será usado pelo *container* para notificá-la. Este objeto implementa a seguinte interface:

```
interface ContextLetEventHandler {
  notify(Serializable EventObject)
}
```

Consultas a Contextlets: Uma aplicação sensível ao contexto pode obter uma referência para um *contextlet* em execução em um *container*. Esta referência pode então ser usada para chamar, de forma síncrona, métodos deste *contextlet*, a fim de obter informações sobre o seu estado. Para obter uma referência para um *contextlet*, deve-se usar o método `getContextLetbyName(String)`, informando o nome do *contextlet* como parâmetro.

4 Exemplos de ContextLets

Com o objetivo de ilustrar o uso do sistema J2CS, são mostrados a seguir dois exemplos de aplicações sensíveis ao contexto construídas sobre a plataforma. As aplicações mostradas consideram como cenário um departamento universitário cujos ambientes são equipados com diversos tipos de sensores. Os professores, funcionários e estudantes deste departamento possuem crachás, os quais são usados para garantir acesso a laboratórios, salas de reunião etc. Assume-se também que todas as estações de trabalho dos laboratórios possuem *tags* RFID, o que permite que sensores de leitura identifiquem sempre que um equipamento entrar ou sair de um laboratório.

Em cada exemplo, mostra-se o código dos contexlets, incluindo o código responsável pela migração e instalação dos mesmos.

Exemplo 1: Assume-se neste exemplo um *contextlet* responsável por gerar um evento toda vez que um equipamento entrar ou sair de um dos laboratórios do departamento. Quando isto ocorrer, assume- se que um monitor de contexto acoplado a um sensor RFID publica um evento do tipo `MovimentacaoEquipamento`, o qual é tratado pelo seguinte *contextlet*:

```
1: class ContextLetControleEquipamento extends ContextLet {
2:   int tipo;
3:
4:   public ContextLetControleEquipamento(String nome,int t){
5:     this.tipo= t; super(nome);
6:   }
7:   public void update(MovimentacaoEquipamento equip) {
```

```
 8:      if (equip.getTipo() == tipo)
 9:         publish(equip);
10:    }
11:  }
```

O construtor desta classe (linhas 4 a 6) recebe como argumento o nome do *contextlet* e um inteiro denotando o tipo do equipamento cuja movimentação deseja-se monitorar (impressoras, microcomputadores, projetores multimídia etc). O método `update` (linhas 7 a 10) é automaticamente invocado pelo *container* quando um equipamento entrar ou sair do laboratório (isto é, quando o monitor de contexto associado ao sensor de RFID instalado na porta de entrada do laboratório detectar a passagem de uma *tag* RFID). Se o tipo do equipamento detectado for igual àquele que está sendo monitorado pelo *contextlet*, publica-se um evento para a aplicação responsável por controlar a localização dos equipamentos no departamento (linha 9).

O trecho de código a seguir mostra a instanciação e instalação do referido *contextlet*.

```
1: ContextLet c=
2:   new ContextLetControleEquipamento("ctx_equip",tipo);
3: ContextLetMonitorBinding m=
4:   new ContextLetMonitorBinding("SensorRFIDLab",
5:   "(sala >= 200) && (sala < 300)");
6: ContextLetHelper.install(c, m);
```

Inicialmente, instancia-se o *contextlet* com o nome `ctx_equip` (linhas 1 e 2). Nas linha 3 a 5, instancia-se um objeto do tipo `ContextLetMonitorBinding`, o qual é usado para especificar que o *contextlet* será associado a monitores de contexto cujas propriedades estáticas são do tipo `SensorRFIDLab`. Mais especificamente, dentre estes monitores, a associação será realizada com aqueles localizados nos laboratórios do departamento (salas com números maiores ou iguais a 200 e menores do que 300). Por fim, o *contextlet* é migrado para um *container* (linha 5).

Exemplo 2: Assume-se neste segundo exemplo um *contextlet* responsável por notificar uma aplicação usada por um professor quando dois de seus orientandos estiverem presentes no LCD (Laboratório de Computação Distribuída) na data e horário de sua reunião semanal com os mesmos.

```
 1: class ContextLetReuniaoSemanal extends ContextLet {
 2:   int matr1, matr2;
 3:   boolean presente1= false, presente2= false;
 4:   RemoteService rs;
 5:
 6:   public ContextLetReuniaoSemanal(String nome, int matr1,
 7:                                    int matr2, RemoteService rs) {
 8:     this.matr1= matr1; this.matr2= matr2; this.rs= rs;
 9:     super(nome);
10:   }
11:   public void update(DadosEntradaUsuario entrada) {
12:     int matr= rs.fromBadgeIdToMatr(entrada.getBadgeId());
13:     if (matr == matr1) presente1= true;
```

```
14:     if (matr == matr2) presente2= true;
15:     if (presente1 && presente2 && "data/hora reuniao")
16:       publish(new String("Estudantes presentes no LCD"));
17:   }
18:   public void update(DadosSaidaUsuario saida) {
19:     int matr= rs.fromBadgeIdToMatr(saida.getBadgeId());
20:     if (matr == matr1) presente1= false;
21:     if (matr == matr2) presente2= false;
22:   }
23: }
```

O construtor desta classe recebe como argumento o nome do *contextlet*, as matrículas dos alunos que se deseja monitorar e uma referência para um serviço remoto (linhas 6-10). O primeiro método `update` (linhas 11 a 17) é automaticamente invocado pelo *container* quando um usuário entrar no referido laboratório (isto é, quando o monitor de contexto deste laboratório publicar um evento do tipo `DadosEntradaUsuario`). Este método utiliza um serviço remoto para converter o identificador do crachá do usuário para seu número de matrícula (linha 12). As variáveis de instância `presente1` e `presente2` são usadas para indicar a presença de cada um dos dois alunos no laboratório. Caso os dois alunos estejam presentes e seja a data e hora da reunião semanal, publica-se um evento para a aplicação sensível ao contexto usada pelo professor (linhas 15 e 16). O segundo método `update` (linhas 18 a 22) é usado para atualizar o estado do *contextlet* quando os alunos saírem do referido laboratório.

O próximo trecho de código mostra a instanciação e instalação do referido *contextlet*.

```
1: ContextLet c=
2:  new ContextLetReuniaoSemanal("ctx_reuniao",matr1,matr2,rs);
3: ContextLetMonitorBinding m=
4:  new ContextLetMonitorBinding("SensorEntrada","sala==210");
5: ContextLetHelper.install(c, m);
```

Inicialmente, instancia-se o *contextlet* com o nome `ctx_reuniao` (linhas 1 e 2). Nas linha 3 e 4, instancia-se um objeto do tipo `ContextLetMonitorBinding`, o qual é usado para especificar que o *contextlet* será associado a monitores de contexto cujas propriedades estáticas são do tipo `SensorEntrada`. Mais especificamente, dentre estes monitores, a associação será realizada com aquele localizado na sala 210 (número da sala do Laboratório de Computação Distribuída). Por fim, o *contextlet* é migrado para um *container* (linha 5).

Exemplo 3: Mostra-se neste exemplo um *contextlet* que monitora os usuários que estão presentes no Laboratório de Computação Distribuída. Portanto, em linhas gerais, este *contextlet* é semelhante ao mostrado no Exemplo 2. No entanto, o mesmo exporta um método `obterUsuariosLCD`, o qual pode ser chamado por uma aplicação sensível ao contexto para obter uma lista de usuários que estão presentes neste laboratório. Este *contextlet* ilustra, portanto, um modo de comunicação baseado em consulta (*poll*), onde a aplicação pró-ativamente consulta o estado do *contextlet*.

```
01: interface ConsultaUsuariosLCD extends Remote {
02:   String[] obterUsuariosLCD() throws RemoteException;
```

```
03: }
04:
05: class ContextLetUsuariosLCD extends ContextLet
06:                              implements ConsultaUsuariosLCD {
07:   String[] usuariosLCD;
08:   ......
09:   public void update(DadosEntradaUsuario entrada) {
10:     .....
11:   }
12:   public void update(DadosSaidaUsuario entrada) {
13:     .....
14:   }
15:   String[] obterUsuariosLCD() {
16:     return usuariosLCD;
17:   }
18: }
```

Por questões de economia de espaço, não foram mostrados os códigos dos métodos `update` uma vez que estes são parecidos com aqueles do exemplo anterior. Não será mostrado também o código responsável por instalar o *contextlet* no *container*. Mostra-se a seguir, no entanto, o trecho de código que uma aplicação sensível ao contexto poderia utilizar para consultar o estado do contextlet e obter uma lista de usuários presentes no LCD:

```
01: ConsultaUsuariosLCD ctxUsuLCD=
02:    J2CSContainer.getContextLetByName("ctx_usu_lcd");
03: String []usuariosLCD= ctxUsuLCD.obterUsuariosLCD();
```

Nas linhas 1 e 2, obtém-se uma referência para um *contexlet* previamente registrado com o nome `ctx_usu_lcd`. Na linha 3, chama-se o método `obterUsuariosLCD` deste *contextlet*.

5 Resultados Experimentais

Esta seção apresenta os resultados de um experimento utilizando um protótipo de J2CS. O principal propósito do experimento foi apresentar resultados sobre o desempenho do sistema. O experimento foi realizado em máquinas AMD Athlon 1.9 GHz, com 512 MB RAM, Microsoft Windows Service Pack 4 e JDK 5.0. Utilizou-se uma rede local Ethernet para conectar os computadores.

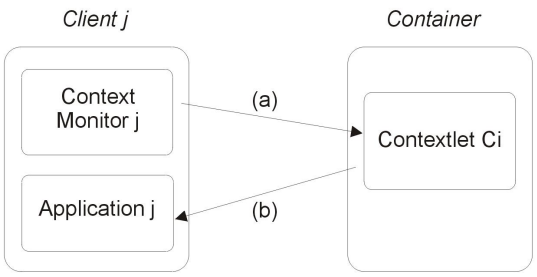

Figura 2. Configuração do experimento

A Figura 2 resume a configuração do experimento. Um dos computadores foi configurado como servidor J2CS, ou seja, um serviço de *container* foi iniciado nesta máquina. Também configurou-se algumas máquinas clientes. Cada cliente foi configurado com dois processos: um monitor de contexto e uma aplicação sensível ao contexto. Para a avaliação do desempenho em tempo de execução da arquitetura, o experimento foi executado 10 vezes, com o número de clientes conectados variando de 1 (na primeira execução) a 10 (na última execução). Na i-ésima execução ($1 <= i <= 10$), o seguinte contextlet foi instalado no serviço de container:

```
class Ci extends Contextlet {
  void update(A1 a1) { publish (a1); }
  void update(A2 a2) { publish (a2); }
  ...
  void update(Ai ai) { publish (ai); }
}
```

Em cada execução do experimento, cada monitor de contexto i foi programado para disparar 5000 eventos do tipo `Ai` em intervalos de 1 ms.

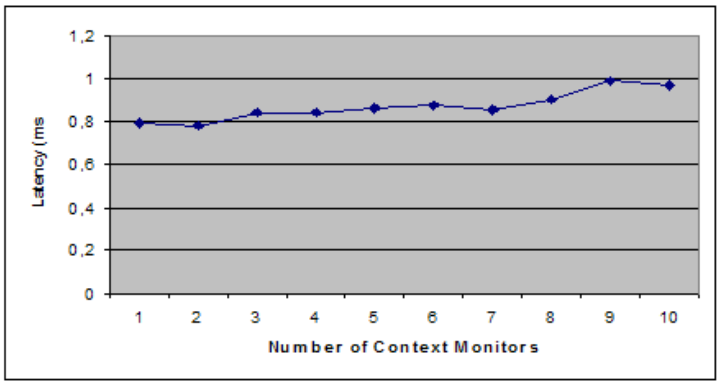

Figura 3. Avaliação de desempenho

Na i-ésima execução, foi calculada a latência média entre o envio de um evento `Aj` no monitor de contexto j e o recebimento do mesmo na aplicação associada, sendo $1 <= j <= i$. Em outras palavras, foi calculado o tempo médio da entrega dos eventos (a) e (b) na Figura 2. Posteriormente, a partir da latência calculada para cada par monitor/aplicação, calculou-se a latência média de entrega de eventos na i-ésima execução do experimento. A Figura 3 mostra os resultados finais obtidos. Como podemos ver através do gráfico, esses tempos variam de 0.8 ms, no caso de 1 cliente, até aproximadamente 1 ms, quando utilizamos todos os 10 clientes. Considera-se que este resultado mostra que para aplicações sensíveis ao contexto restritas a uma rede local a implementação de J2CS possui um desempenho aceitável.

6 Trabalhos Relacionados

Nesta seção, são descritos com mais detalhes três sistemas de *middleware*: Context Toolkit [4], Gaia [8] e JCAF [2]. Além de apresentar estes sistemas, procura-se também realizar uma comparação dos mesmos com J2CS. Por fim, são descritos outros sistemas de *middleware* para desenvolvimento de aplicações sensíveis ao contexto.

Context Toolkit: O projeto do sistema Context Toolkit foi provavelmente uma das primeiras iniciativas em propor um sistema de apoio ao desenvolvimento de aplicações sensíveis ao contexto. O sistema defende a tese de que uma aplicação sensível ao contexto deve ser estruturada em torno de componentes especializados, responsáveis pela captura, interpretação e agregação de contexto. No sistema Context Toolkit, *widgets* são componentes que encapsulam detalhes inerentes à comunicação com sensores. São, portanto, semelhantes aos monitores de contexto de J2CS. Interpretadores são responsáveis por aumentar o nível de abstração das informações coletadas pelos *widgets*. Por exemplo, em um sistema de localização, um interpretador pode transformar coordenadas geográficas em nomes de ruas, praças etc. Um agregador é responsável por relacionar e inferir informações de contexto geradas por componentes distintos. Por exemplo, um agregador pode deduzir que uma reunião está em andamento a partir de informações fornecidas por sensores de presença e de som. Em J2CS, *contextlets* desempenham o papel tanto de interpretadores como de agregadores. Por fim, o quarto componente do sistema Context Toolkit é chamado de localizador (*discover*) e tem a missão de manter um registro dos componentes disponíveis em uma instalação do sistema.

Em relação ao sistema Context Toolkit, a plataforma J2CS apresenta duas contribuições principais. Primeiro, o projeto de J2CS é inspirado no padrão arquitetural normalmente utilizado em sistemas de *middleware* que disponibilizam componentes para aplicações corporativas (como EJB [12] e CCM [11]). Esta característica incrementa o nível de disponibilidade e separação de interesses da infra-estrutura provida pelo sistema. Em segundo lugar, J2CS se vale dos recursos de reflexão computacional, comunicação remota e serialização de objetos de Java com o objetivo de prover uma plataforma de *middleware* dinamicamente reconfigurável.

Gaia: Em [8], Ranganathan e Campbell descrevem um sistema de *middleware* para desenvolvimento de aplicações sensíveis ao contexto na plataforma Gaia[1]. No *middleware* proposto, existem três componentes principais: provedores de contexto, sintetizadores de contexto e consumidores de contexto. Estes componentes desempenham papéis semelhantes, respectivamente, aos monitores de contexto, *contextlets* e aplicações sensíveis ao contexto em J2CS. Uma característica importante do *middleware* para a plataforma Gaia é o fato de o sistema utilizar predicados lógicos para representar contexto e, por conseqüência, prever o emprego de formalismos típicos de Inteligência Artificial, tais como lógica de predicados, lógica fuzzy, redes bayesianas, redes neurais etc. A idéia é que estes mecanismos sejam usados para interpretação e inferência de contexto. Por outro lado, em J2CS, optou-se por usar uma linguagem imperativa – no caso, Java – para codificar o tratamento e processamento de informações de contexto. Como mostram os exemplos da Seção 4, aplicações sensíveis ao contexto via de regra não demandam algoritmos sofisticados para tratamento de contexto. No entanto, nada impede que a partir de um código de um *contextlet* se chame, por exemplo, um sistema de inferência implementado em alguma linguagem lógica. Por fim, em Gaia, não existem regras para associação dinâmica entre provedores e sintetizadores de contexto, como em J2CS.

JCAF: O sistema JCAF (*Java Context-Awareness Framework*) é organizado em dois grandes módulos: uma infra-estrutura de execução e um arcabouço para programação

[1] Gaia é um ambiente operacional para cenários de computação ubíqua proposto pelo grupo de pesquisa dos referidos autores [9].

de aplicações sensíveis ao contexto. A infra-estrutura de execução de JCAF é formada por serviços de contexto (*context services*), aos quais cabe hospedar entidades. Estas são objetos Java que modelam e armazenam o contexto de objetos do mundo real. Observadores de entidades (*entity listeners*) são aplicações Java interessadas em serem notificadas quando houver determinadas alterações no estado de uma entidade. O sistema incorpora ainda os seguintes componentes: monitores de contexto (objetos que capturam contexto), atuadores (objetos que alteram o contexto) e transformadores (os quais têm o mesmo propósito dos interpretadores do sistema Context Toolkit). Por fim, JCAF inclui um componente que controla e autentica o acesso aos demais componentes da arquitetura.

A principal semelhança de projeto entre JCAF e J2CS é o fato de ambos os sistemas incluirem uma infra-estrutura para execução de aplicações encarregadas de capturar, interpretar e inferir contexto. Além disso, ambos os sistemas são fortemente integrados à biblioteca de classes de Java, fazendo uso de serviços como Java RMI, JDBC etc. No entanto, existe uma diferença importante no modelo conceitual proposto pelos dois sistemas para interpretação de contexto. JCAF utiliza extensivamente o conceito de entidades para representar objetos do mundo real. Já J2CS não parte do pressuposto de que toda entidade do mundo real deve ter uma representação interna no sistema. Nas experiências realizadas com ambos os sistemas, observou-se que o modelo proposto por JCAF incentiva a instanciação de diversos objetos, os quais freqüentemente possuem funcionalidades bastante simples. Por outro lado, J2CS incentiva um estilo de programação orientado por eventos, baseado em um menor número de objetos, os quais possuem maior granularidade. Além disso, J2CS utiliza Jini para localização de serviços remotos, o que proporciona um maior desacoplamento entre clientes e servidores. J2CS utiliza também regras que permitem associação dinâmica entre monitores de contexto e *contextlets*. Por outro lado, JCAF inclui um componente controlador de acesso, o qual ainda não existe em J2CS.

Outros Middleware Sensíveis ao Contexto: Carisma [3] é um *middleware* que usa os princípios de reflexão computacional para apoiar o desenvolvimento de aplicações sensíveis ao contexto para dispositivos móveis. Em Carisma, arquivos de configuração são usados para especificar políticas usadas para adaptar o comportamento de uma aplicação. O foco do sistema, no entanto, consiste na adaptação de uma aplicação a partir de mudanças de contexto que ela mesma é capaz de detectar. Diferentemente de J2CS, o sistema não inclui componentes para aquisição, análise e síntese de informações de contexto disponibilizadas em um ambiente de computação ubíqua.

QuO (Quality Objects) [16], desenvolvido pela BBN, é um exemplo conhecido de plataforma de *middleware* que permite a incorporação de atributos de qualidade de serviço em aplicações distribuídas baseadas no padrão CORBA. QuO propõe uma linguagem para especificação de contratos de qualidade de serviço, chamada CDL (*Contract Definition Language*). Porém, QuO foi projetado para apoiar o desenvolvimento de aplicações capazes de se adaptar a variações de QoS típicas de um ambiente de rede de longa distância. Por exemplo, QuO não oferece abstrações dedicadas a integrar diversas fontes/monitores de contexto espalhadas por um ambiente de computação ubíqua.

7 Discussão e Conclusões

Neste artigo, foram descritas a arquitetura e a interface de programação do sistema J2CS, um *middleware* voltado para o desenvolvimento de aplicações sensíveis ao contexto em

Java. Inspirado em sistemas tradicionais de *middleware* orientados por componentes, J2CS oferece uma infra-estrutura que encapsula e implementa de forma automática diversos detalhes inerentes ao processamento de contexto em ambientes de computação ubíqua. Em J2CS, *containers* são usados para interagir com monitores de contexto e para rotear informações de contexto para aplicações chamadas *contextlets*, às quais cabe inter-relacionar, associar e agregar as informações recebidas e então disseminar eventos de mais alto nível para aplicações sensíveis ao contexto. Estas, em última instância, são as beneficiárias diretas dos serviços providos pelo *middleware*.

Na plataforma J2CS, *contextlets* podem ser dinamicamente instalados e removidos de *containers*, o que incrementa a flexibilidade e extensibilidade do sistema. Em um certo sentido, *contextlets* podem ser considerados como agentes móveis, os quais se movem para *containers* para interpretar algum tipo de informação de contexto e então publicar a informação interpretada para aplicações sensíveis ao contexto. Em J2CS, utiliza-se ainda regras que viabilizam uma associação dinâmica entre monitores de contexto e *contextlets*. J2CS é plenamente integrado ao ambiente de programação de Java, usando diversas APIs deste ambiente. Dentre elas, podemos citar JDBC (para acesso a bancos de dados), RMI (para comunicação), Jini (para localização de serviços), serialização (para migração de *contextlets*) e reflexão (para implementação de assinatura de eventos baseada em tipos).

J2CS utiliza um mecanismo de comunicação baseado em eventos. Acredita-se que o desacoplamento no tempo, no espaço e de sincronização proporcionado por um sistema de eventos é bastante adequado para ambientes abertos e dinâmicos, como aqueles típicos de computação ubíqua [5]. Além disso, acredita-se que um modelo de programação centrado em eventos, condições e ações, como aquele adotado na programação de *contextlets*, oferece abstrações que são úteis na implementação de uma vasta gama de aplicações sensíveis ao contexto [6]. Por outro lado, em J2CS, aplicações sensíveis ao contexto podem se valer também de mecanismos síncronos de comunicação para, por exemplo, consultar o estado de *contextlets* ou de monitores de contexto.

Na atual implementação do sistema, o *container* J2CS é executado em um servidor centralizado. A princípio, poderia ser argumentado que este fato limita a escalabilidade da plataforma. No entanto, J2CS tem como alvo principal o desenvolvimento de aplicações sensíveis ao contexto restritas a um determinado domínio administrativo, como um departamento de uma universidade. Nestes cenários, o problema de escalabilidade não é crítico, conforme pode-se verificar através do experimento mostrado na Seção 5.

Na grande maioria das vezes, o acesso a informações de contexto não deve ser disponibilizado a qualquer cliente, inclusive por questões de privacidade. Além disso, muitas vezes é imprescindível o uso de um mecanismo de autenticação, de forma a identificar tanto a fonte como o destino de uma determinada informação. Pode ser também necessário estabelecer canais seguros de comunicação entre provedores e consumidores. Por fim, pode ser necessário quantificar o grau de confiança de uma determinada informação de contexto. A versão atual de J2CS não possui suporte a nenhum destes recursos e, portanto, planeja-se que os mesmos sejam objeto de investigações futuras. Pretende-se também implementar novos estudos de caso usando-se o sistema.

Referências

[1] Gregory D. Abowd. Software engineering issues for ubiquitous computing. In *21st International Conference on Software Engineering*, pages 75–84. IEEE Computer Society Press, 1999.

[2] Jakob E. Bardram. The Java Context Awareness Framework (JCAF). In *Third International Conference on Pervasive Computing*, volume 3468 of *Lecture Notes in Computer Science*, pages 98–115. Springer, 2005.

[3] Licia Capra, Wolfgang Emmerich, and Cecilia Mascolo. Carisma: Context-aware reflective middleware system for mobile applications. *IEEE Transactions Software Engineering*, 29(10):929–945, 2003.

[4] Anind K. Dey, Gregory D. Abowd, and Daniel Salber. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human-Computer Interaction*, 16(2–4):97–166, Dec. 2001.

[5] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, 2003.

[6] Sumi Helal. Programming pervasive spaces. *IEEE Pervasive Computing*, 4(1):84–87, 2005.

[7] Jason I. Honga and James A. Landay. An infrastructure approach to context-aware computing. *Human-Computer Interaction*, 16(2–4):287–303, Dec. 2001.

[8] Anand Ranganathan and Roy H. Campbell. A middleware for context-aware agents in ubiquitous computing environments. In *ACM/IFIP/USENIX International Middleware Conference*, volume 2672 of *Lecture Notes in Computer Science*, pages 143–161. Springer, 2003.

[9] Manuel Román, Christopher K. Hess, Renato Cerqueira, Anand Ranganathan, Roy H. Campbell, and Klara Nahrstedt. Gaia: A middleware infrastructure to enable active spaces. *IEEE Pervasive Computing*, pages 74–83, Oct-Dec 2002.

[10] M. Satyanarayanan. Pervasive computing: Vision and challenges. *IEEE Personal Communications*, pages 10–17, August 2001.

[11] Jon Siegel. *CORBA 3 Fundamentals and Programming*. John Wiley & Sons, 2nd edition, 2000.

[12] Sun Microsystem. Enterprise Java Beans specification (version 2.1), November 2003.

[13] Jim Waldo. The Jini architecture for network-centric computing. *Communications of the ACM*, 42(7):76–82, July 1999.

[14] Mark Weiser. The computer for the 21st century. *Scientific American*, 265(3):66–75, January 1991.

[15] Mark Weiser. Some computer science issues in ubiquitous computing. *Communications of the ACM*, 36(7):74–84, 1993.

[16] John A. Zinky, David E. Bakken, and Richard E. Schantz. Architectural support for quality of service for CORBA objects. *Theory and Practice of Object Systems*, 3(1):55–73, 1997.