A Flexible and Extensible Component-Oriented Middleware for Creating Context-Aware Applications

Daniel Coutinho de Miranda and Marco Tulio de Oliveira Valente
Computer Science Department
Catholic University of Minas Gerais, Brazil
danielcm@gmail.com, mtov@pucminas.br

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 the proposed platform, containers manage the life cycle of components called contextlets, which are used to infer high-level events from low-level sensed data. Containers are also in charge of handling communication details with sensors and other sources of context information. In this paper, we discuss the software architecture, the programming interface, the run-time performance and the implementation of J2CS. We also present some examples of context-aware applications based on the proposed middleware.

1 Introduction

Nowadays, environments saturated with computing devices are commercially available, including laptops, handhelds, wearable computers, electronic equipments, sensors, actuators and so on. Such environments are transforming in reality what Mark Weiser fifteen years ago named ubiquitous computing or third wave of computing [20, 21]. According to this classification, the first wave was dominated by mainframes and the second one by personal computers. Each of these waves have collaborated to the appearing of new computer systems, which did not exist previously. For example, mainframes made possible the implementation of several enterprise information systems; in the same way, desktops have popularised many personal productivity applications. In the case of ubiquitous computing, it is expected to be frequent the development of *context-aware applications* [5, 4, 7, 16].

A context-aware application is one that uses information about the context of relevant entities in its domain, either to provide information or services to users. Typically, context information includes location, identity or state of people, physical objects or computing devices. As an example, a print service can provide to a mobile user the possibility of printing by default in the nearest printer. A cell phone can set itself to silent mode if the user is in a meeting (in other words, if location sensors indicate that the user is in the meeting room of his company and sound sensors indicate that a predefined noise level was reached). As a last example, an electronic mail system can provide the possibility of sending a message only to users that are located in an specific environment (e.g. a laboratory).

The research described in this paper was motivated by the observation that the implementation, deployment and execution of context-aware applications can benefit from a middleware infrastructure [5, 9, 14, 1, 8]. Basically, this infrastructure should be responsible for capturing, interpreting and disseminating context information. In order to capture context information, the platform should interact with context providers, including sensors, devices and computer systems. In the same way, the middleware should be able to interpret and raise the level of sensed information. For example, in order to detect the start of a meeting in a room, information provided by location and sound sensors should be aggregated. Finally, the middleware platform should disseminate high-level events synthesized from information gathered in a pervasive environment. The ultimate purpose, as usual in middleware platforms, is to simplify and to make more productive the implementation of context-aware applications, by delegating to the middleware layer several interests concerned with the development of this kind of application.

In this paper, we describe the J2CS (Java 2 Context Service) system, a lightweighted, flexible, exten-

sible and component-oriented middleware designed to support the implementation and execution of context-aware applications in Java. The design of J2CS was inspired by the architectural pattern that is the base of modern component-based middleware systems, such as EJB [18] and CCM [17]. The architecture of such systems is centered in the abstraction of containers, which are responsible for managing the life cycle of components and to provide them with non-functional services, including persistence, transactions, security, fault-tolerance, etc.

In the proposed system, containers are responsible for executing applications, called *contextlets*, that are used to interpret and disseminate context information. In the platform, *containers* are also used to interact with computing devices that provide context information. At deployment time, contextlets can subscribe to context information published by particular context providers. Containers are also responsible for delivering the information generated by a sensor to its associated contextlets. Contextlets should interpret the sensed information and notify context-aware applications that a relevant event has occurred. J2CS is flexible and extensible since contextlets can be installed and removed from containers without the need of restarting the whole system.

The remaining of this paper is organized as follows. Section 2 presents the main services that a middleware platform designed to support the development of context-aware application should provide. Section 3 describes the building blocks of the J2CS system and its software architecture. Section 4 presents examples of context-aware applications designed using J2CS. In Section 5, we describe implementation issues. Section 6 describes an experimental evaluation of the system. Finally, Section 7 presents related work and Section 8 concludes the paper.

2 Middleware for Context-aware Applications

Middleware platforms targetting the implementation of context-aware applications should shield application developers from many tasks inherent to context handling in ubiquitous computing environments. Considering the point of view of a context-aware application engineer, such platforms should provide at least the following services [5, 9, 14]:

- Context interpretation: context-aware applications must be able to handle context information without having to implement low-level details concerning communication with sensors and other computing devices. Moreover, context-aware applications must only react to events associated to their functional requirements. For example, an application may not be interested in being notified each time the location of a user changes, but only when he is located in a particular environment (e.g. a laboratory or a meeting room). Thus, the middleware layer must be able to publish high-level events through the interpretation, aggregation and association of low-level events received from sensors and other computing devices.

- Communication: components responsible for collecting and reasoning about context information are often distributed over different nodes in the network. Thus, any middleware platform that aims to support the implementation of context-aware applications must provide appropriate abstractions that allow such entities to communicate and to coordinate themselves. As an example of such abstractions, we can mention remote method/procedure invocation, tuple spaces, events etc.

- Dynamic reconfiguration: middleware platforms targetting the development of context-aware applications must support dynamic updating of its internal components, including context providers, context interpreters and context consumers. More specifically, users should be able to change the internal components of the system without the need of recompiling its kernel or restarting its execution.

- Service discovery: since ubiquitous computing environments are usually very dynamic and open, the components of context-aware applications usually do not know the location of other components they need to interact with. For this reason, middleware platforms for such environments must provide a flexible lookup service, that does not require clients to know at deployment time the exact location of the services they use.

- Availability: middleware platforms targetting the development of context-aware applications must execute continuously, since we can not know a priori when a context-aware application will require the services provided by the platform.

3 J2CS Architecture

As illustrated by Figure 1, J2CS main components are the following:

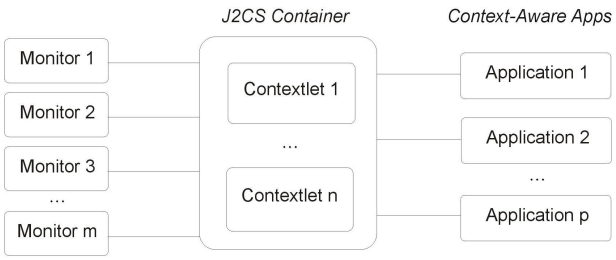

Figure 1. Main components of J2CS

- Context Monitors: applications that interact with a source that provides context information. A context source can be hardware (e.g. a sensor) or software (e.g. a financial application that provides stock information). Context monitors are responsible for publishing events each time the state of a context source changes. For example, a temperature sensor should produce an event each time the temperature changes.
- Containers: applications that act as mediators between context monitors and context-aware applications. Containers are responsible for capturing low-level events published by context monitors and propagate then to context interpreters, which are called contextlets in J2CS. Servers responsible for running a container service are called J2CS servers.
- Contextlets: components that encapsulate details related to context interpretation. Contextlets are Java objects that transform low-level events into high-level ones, which are propagated to context-aware applications.
- Context-aware applications: applications that directly benefit from the proposed middleware architecture. In J2CS, context-aware applications just need to handle high-level events generated by contextlets.

The components of the J2CS system are described with more details in the next subsections.

3.1 Context Monitors

Context monitors are components associated with a context source, such as a sensor or other computing device. They encapsulate communication details between this source and a J2CS container. Moreover, they are used to publish context information asynchronously and also to answer queries coming from contextlets.

Context monitors have a set of static properties, which are used to identify their main characteristics. For example, a temperature sensor can have the following static properties: its identifier (mandatory in all monitors), its location, model, how often the temperature is measured, the unit of measure etc. Besides, context monitors can publish dynamic properties about its monitored environment. For example, a temperature sensor can publish information such as the value of the temperature and the date of the measuring. Finally, context monitors can respond to queries coming from containers. Such queries are made by a container when it needs to get information about the state (that is, the dynamic properties) of a monitor.

Registering and publishing events: Context monitors are active objects that can respond to messages as follows. The message `bind(StaticData)` is used to register a context monitor in a container, having as input parameter an object that contains the static properties of the monitor. Message `unbind(String)` is used to cancel the registering of a context monitor in a container. The message `publish(DynamicData)` is used to publish a change in the context of the monitor. The input parameter of this message is an object containing the new dynamic properties of the context. Finally, messages `getState()` and `setState(DynamicData)` are used to get and update the dynamic state of the monitor. The first message is generally sent by a container, while the second one is sent by a context source.

3.2 Contextlets

A contextlet is an object that is dynamically installed in a J2CS server. A contextlet receives context information published by context monitors, interprets this information and eventually generates high-level events to context-aware applications.

Instantiation and installation: Contextlets must extend the class `Contextlet`. All contextlets have a unique name, which must be passed as a parameter to the constructor of this class. After they are instantiated in a specific node of the network, contextlets are usually moved to a container, using the following method:

```
ContextletHelper.install(Contextlet c,
    ContextletMonitorBinding[] m)
```

This method receives as parameter a reference `c` to the contextlet that we want to move to the container and an array `m` containing the context monitors this contextlet will be associated to. Similar

to mobile agents systems, the state of a contextlet is transferred with it. The execution of a contextlet in a container starts by its `onArrival()` method. Eventually, developers may need to remove a contextlet from a container. For this purpose, the method `ContextLetHelper.retract(String)` must be invoked, with the name of the contextlet to be removed as a parameter.

Association between contextlets and context monitors: A contextlet can be associated with one or more context monitors, which means that this contextlet will be notified about events published by such monitors. In order to associate a contextlet to a context monitor, we need to create an object of type `ContextletMonitorBinding`. Objects of this type have two attributes: the name of the class containing the static properties of the monitor we want to associate to the contextlet and a boolean expression used to select instances of this class. This selection relies on the static properties of the monitors. In this way, the operands of this expression must be constants or attributes of the class denoting the static properties of the context monitor. The selection expression can also use the built-in logical and relational operators of Java.

The set of context monitors associated with a contextlet is dynamic. At run-time, monitors can be removed from this set if they are unregistered from the system (using the `unbind` method). Moreover, if a new monitor is registered after the installation of the contextlet, this monitor can be dynamically associated to this contextlet. Of course, this late association only occurs if the monitor attends the requirements defined by the `ContextletMonitorBinding` object associated to the contextlet.

Queries: Contextlets can get references to the context monitors registered in the system using the following methods inherited from the `Contextlet` class:

```
ContextMonitor[] getMyContextMonitors()
ContextMonitor[] getContextMonitors(
   ContextletMonitorBinding)
```

The first method returns a list of references to the context monitors associated to the calling contextlet. The second method returns a list of references to the monitors specified by the `ContextletMonitorBinding` object, which do not necessarily need to be the same context monitors associated to the contextlet for the purpose of event notification. In both cases, the returned references can be used to query the state of the associated context monitors (using the method `getState`, as defined in Section 3.1).

Event notification: Suppose that a contextlet `C` is associated with a context monitor `M`, which publishes events of types `E1` and `E2` (both subtypes of `DynamicData`, as defined in Section 3.1). In order to handle such events, the contextlet `C` must implement two methods: `void update(E1 e1)` and `void update(E2 e2)`. Each time `M` publishes an event, the container takes the charge of routing this event to the respective `update` method of the contextlet `C`. In other words, J2CS uses a type-based schema for event subscription [6].

Interpretation and publication of high-level events: The `update` methods implemented by a contextlet must interpret, aggregate and correlate context information, possibly generating high-level events from those low-level events received from context monitors. In order to publish high-level events to context-aware client applications, contextlets must invoke the `publish` method inherited from the `Contextlet` class, passing as a parameter an object that implements the Java built in `java.io.Serializable` interface.

3.3 Context-Aware Applications

Context-aware applications directly benefit from the services provided by the J2CS platform. In general, such applications are notified of high-level events generated by contextlets and eventually they can also consult the state of a contextlet.

Event subscription: In order to subscribe to events generated by a given contextlet, a context-aware application must invoke the following method:

```
ContextletHelper.subscribe(String,
   ContextletEventHandler)
```

The first parameter of this method is the name of the contextlet from which the application wants to receive events; the second parameter is an `ContextletEventHandler` object, which is used by the container to callback the application when the specified contextlet publish an event. In order to cancel an event subscription, the application must call the `ContextletHelper.unsubscribe(String)` method, passing the name of the contextlet as parameter.

Queries: A context-aware application can get remote references to contextlets executing in a container. Such references can be used to call methods of the contextlet. In order to get a reference to a contextlet, the method `ContextletHelper.getContextletbyName(String)` must be called informing the name of the contextlet as a parameter.

4 Examples

In order to illustrate the use of the J2CS system, three examples of context-aware applications built on top of the platform are described in this section. The supposed scenario is an university department with different types of sensors in its physical environments. In this department, professors, administrative staff and students carry badges that are used to authorize access to the labs, class rooms, meeting rooms etc. It is also assumed that the workstations of the department have RFID tags. This allows reading sensors to identify each time when an equipment enters or leaves a laboratory.

In the following paragraphs, we present and discuss the contextlets used by some of the context-aware applications in use by this department.

Example 1: In this example, we describe a contextlet that generates an event each time an equipment enters or leaves in one of the labs of the department. When such event happens, a context monitor connected to a RFID sensor publishes an event of the type `MovingEquip`, which is handled by the following contextlet:

```
 1: class EquipmentTracking extends Contextlet {
 2:   int type;
 3:
 4:   public EquipmentTracking(String name,int t){
 5:     this.type= t; super(name);
 6:   }
 7:   public void update(MovingEquip equip){
 8:     if (equip.getType() == type)
 9:        publish(equip);
10:   }
11:  }
```

The constructor of class `EquipmentTracking` (lines 4-6) receives as argument the name of the contextlet and an integer indicating the type of device we would like to monitor (printers, desktops, multimedia projectors etc). The method `update` (lines 7-10) is automatically invoked by the container when a device enters or leaves the lab (that is, when the context monitor associated to the sensor installed at the entry of the lab detects a RFID tag). When this happens and the type of the detected device is the same as the type monitored by the contextlet, an event is published to the context-aware application that controls the location of the department computing equipments (line 9).

We show next the code that handles the instantiation and installation of a contextlet of type `EquipmentTracking`:

```
1: Contextlet c=
2:   new EquipmentTracking("ctx_equip",type);
3: ContextletMonitorBinding m= new
4:   ContextletMonitorBinding("RFIDLab",
5:   "(room >= 200) && (room <= 300)");
6: ContextletHelper.install(c,m);
```

First, the contextlet is created with name `ctx_equip` (lines 1 and 2). Lines 3 to 5 shows the creation of an object of the type `ContextletMonitorBinding`. This object associates the previous contextlet with context monitors whose static properties are of type `RFIDLab` and that are located in rooms whose number range from 200 to 300. Finally, the method `install` moves the contextlet to a container (line 6).

Example 2: In this second example, we present a contextlet that notifies an application when two of the students supervised by a given professor are in the DCL (Distributed Computing Laboratory) room. Moreover, this notification will only happen at the date and time of the weekly meeting among this professor and his students.

```
 1: class MeetingNotification extends Contextlet {
 2:   int id1, id2;
 3:   boolean presence1= false,
 4:   boolean presence2= false;
 5:   RemoteService rs;
 6:
 7:   public MeetingNotification (String name,
 8:     int id1, int id2, RemoteService rs){
 9:     this.id1= id1; this.id2= id2;
10:     this.rs= rs;
11:     super(name);
12:   }
13:   public void update(EntryData entry) {
14:     String msg= "Students in the lab";
15:     int aux= entry.getBadgeId();
16:     int id= rs.fromBadgeIdToCSId(aux);
17:     if (id == id1) presence1= true;
18:     if (id == id2) presence2= true;
19:     if (presence1 && presence2 &&
20:        "date/time == Friday,13:00 PM")
21:        publish(msg);
22:   }
23:   public void update(ExitData exit) {
24:     int aux= exit.getBadgeId();
25:     int id= rs.fromBadgeIdToCSId(aux);
26:     if (id == id1) presence1= false;
27:     if (id == id2) presence2= false;
28:   }
29: }
```

The constructor of this class receives as an argument the name of the contextlet, the identification number of

the students we want to monitor and a reference to a remote RMI object (lines 7-12). The first `update` method (lines 13-22) is automatically invoked by the container when a student enters in the lab (that is, when the context monitor located in the lab publishes an event of the type `EntryData`). The `update` method uses Java RMI to call a remote service that converts the internal identifier returned by the user's identification badge to his identification number in the Computer Science Department (line 16). The instance variables `presence1` and `presence2` are used to indicate the presence of each student in the lab. In case both students are in the lab at the date and time of the weekly meeting (Friday, 13:00 PM), an event is published to the context-aware application used by the professor (lines 19-21). The second `update` method (lines 23-28) is used to update the state of the contextlet when a student leaves the lab.

The following code handles the instantiation and installation of `MeetingNotification`:

```
1: Contextlet c= new MeetingNotification(
2:      "ctx_meeting",id1,id2,rs);
3: ContextletMonitorBinding m= new
4:   ContextletMonitorBinding("PresenceSensor",
5:   "room == 210");
6: ContextletHelper.install(c,m);
```

First, the contextlet is created with name `ctx_meeting` (lines 1 and 2). In lines 3 and 4, we show the creation of an object of the type `ContextletMonitorBinding`. This object associates the previous contextlet with context monitors whose static properties are of type `PresenceSensor` and that are located in the room 210 (room number of the DCL). Finally, the method `install` moves the contextlet to a container (line 6).

Example 3: Similar to the previous contextlet, this example shows a contextlet that monitors the users that are working in the Distributed Computing Lab. However, the contextlet presented next exports a `getUsers` method, which can be invoked by context-aware applications to get a list of the users that are in the lab. Thus, this contextlet illustrates a poll-based communication model, where a context-aware application proactively queries the state of the contextlet.

```
1: interface DCLUsers extends Remote {
2:   String[] getUsers()
3:            throws RemoteException;
4: }
5:
6: class DCLUsersContextlet extends Contextlet
7:    implements DCLUsers {
8:   String[] users;
9:   ......
10:  public void update(EntryData entry) {
11:    .....
12:  }
13:  public void update(ExitData exit) {
14:    .....
15:  }
16:  String[] getUsers() {
17:    return users;
18:  }
19: }
```

For the sake of clarity and simplicity, we do not show the code of the `update` methods. This code is similar to those in the last example. For the same reasons, we do not show the code that create, bind and move the contextlet to its container. On the other hand, we show next the code used by a context-aware application to query the contextlet and get a list of the users present in the DCL.

```
1: String n= "ctx_dcl_users";
2: DCLUsers c= (DCLUSers)
3:   ContextletHelper.getContexteletbyName(n);
4: String[] users= c.getUsers();
```

First, we get a reference to a contextlet previously registered with the name `ctx_dcl_users` (lines 1-3). Next, the method `getUsers` is invoked and returns a list of the users present in the DCL (line 4).

5 Implementation

J2CS was implemented in Java, using the JDK 1.5. As expected, the system is strongly integrated with the Java API. The components of the system rely on Java RMI for remote communication. Java object serialization API is used to migrate a contextlet to a container. JDBC is used to encapsulate the communication with the database that store static properties of context monitors. Finally, Jini [19] is used to support spontaneous localization of the container by context monitors, context-aware applications and any application that installs contextlets in containers.

Moreover, J2CS extensively uses the Java reflection API for two basic purposes. First, the introspection resources of the language are used to get the static properties of a context monitor when it is registered in the container. This simplifies this task , since we can use the same method to register contextlets of different types. Second, when a context monitor publishes an event of type `T1`, reflection is used to select the contextlets that have an `update` method with the following signature: `void update(T2)`, where `T2` is a supertype

of `T1`. In other words, reflection is a key resource in the implementation of the type-based event subscription feature of J2CS [6].

6 Performance Evaluation

This section presents results of a experiment performed with our prototype implementation of J2CS. The experiment main purpose was to present numbers about the performance of the system. We have run the experiment on 1.9 GHz AMD Athlon machines, with 512 KB RAM, Microsoft Windows Service Pack 4 and JDK 5.0. A dedicated fast ethernet network was used to connect the machines.

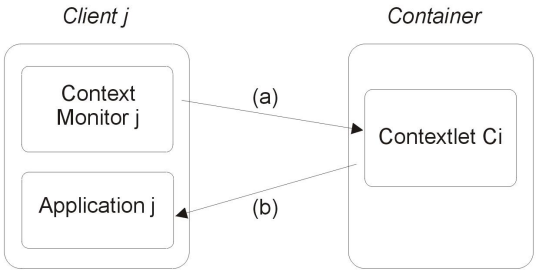

Figure 2. Experiment configuration

Figure 2 summarizes the configuration of the experiment. We have configured one of the machines as a J2CS server, i.e., a container service was started in this machine. We have also configured a number of client machines. Each client was configured with two processes: a context monitor and a context-aware application. In order to evaluate the run-time performance of our architecture, the experiment was executed 10 times, with the number of connected client machines ranging from 1 (in the first execution) to 10 (in the last execution). In the i-th execution ($1 <= i <= 10$), we have installed the following contextlet in the container service:

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

In each execution, each context monitor i was programmed to trigger 5000 events of type `Ai` in intervals of 1 ms.

In i-th execution, we first calculate the average latency between publishing an event `Aj` in context monitor j and receiving it in the associated application,

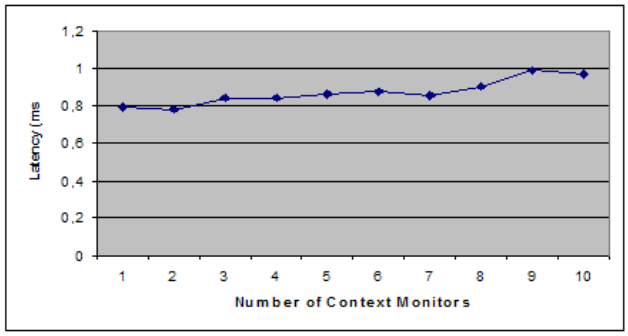

Figure 3. Performance evaluation

with $1 <= j <= i$. In other words, we have calculated the average time to deliver events (a) and (b) in Figure 2. Next, from the latencies calculated for each monitor/application pair in the previous step, we have calculated the average latency for delivering events in the i-th execution of the experiment. Figure 3 shows such final delivering times. As we can see in the graphic, delivering times range from 0.8 ms for one client to around 1 ms to for 10 clients. In our opinion, this result shows that for context-aware applications confined to local area networks J2CS implementation provides very acceptable performance.

7 Related Work

In this section, we describe three middleware systems targetting the implementation of context-aware systems: Context Toolkit [5], Gaia [14] and JCAF [2]. Furthermore, we compare such systems with J2CS. At the end of the section, we briefly present other middleware systems.

Context Toolkit: The Context Toolkit was a seminal research effort in providing a middleware platform for context-awareness. The system is structured around specialized components that shield application developers from context handling concerns. In the Context Toolkit, widgets are components that encapsulate details regarding communication with sensors. Therefore, they are similar to context monitors in J2CS. Interpreters are responsible for raising the level of abstraction of information provided by widgets. For example, in a location system, an interpreter can be used to transform geographical coordinates into street names. An aggregator gathers logically related information produced by different components. For example, an aggregator can deduce that a meeting is happening by relating information provided by location sensors and sound sensors. In J2CS, contextlets can

act both as interpreters and aggregators. Finally, the forth component of Context Toolkit – called discover – keeps a registry of the available components in a given context-aware application.

Regarding the Context Toolkit, J2CS presents two main contributions. First, the design of the system is based on the architectural pattern followed by middleware systems that host components for enterprise applications (such as EJB [18] and CCM [17]). We believe that the container based-approach borrowed from such systems increases the level of availability and separation of concerns provided by J2CS. Second, J2CS is closed integrated with Java and its API, including reflection, serialization, remote communication and service discovery. In our opinion, this integration has contributed to make the system more flexible, simple and dynamically reconfigurable than the Context Toolkit.

Gaia: Ragnathan and Campbell has designed a middleware system for the implementation of context-aware applications in Gaia [15] (Gaia is an infrastructure for running systems in pervasive spaces). The proposed middleware has three main components: context providers, context synthesizers and context consumers. These components play similar roles as the context monitors, contextlets and context-aware applications in J2CS. However, the middleware designed for Gaia relies on logical predicates to represent context. Therefore, different reasoning formalisms, such as first order logic, fuzzy logic, bayesian networks and neural networks, can be used for dealing with context. In J2CS, we have decided to handle context in Java, i.e., in a imperative and object-oriented language. As the examples in Section 4 show, we believe that most context-aware applications do not require complex formalisms to represent, interpret and reason about context information.

JCAF: The JCAF (Java Context-Awareness Framework) system has two main modules: a run-time infrastructure and a framework for programming context-aware applications. The run-time infrastructure consists of servers that host Java objects – called entities – that model and store the context of real world objects. Entity observers are Java applications interested in being notified when particular changes happen in the state of an entity. Moreover, the system supports the following remaining components: context monitors (objects that capture context), actuators (objects that change context) and transformers (objects with the same purpose as the interpreters in the Context Toolkit). Finally, JCAF includes a component that controls and authenticates access to others components in the architecture.

Both JCAF and J2CS include a run-time infrastructure that host applications that interpret and raise the level of context data. Besides, both systems are strongly integrated to the Java API, using services such as RMI and JDBC. However, there is a fundamental difference concerning the object model proposed by both systems for context representation and interpretation. JCAF extensively uses the concept of entities for representing real world objects. Differently, J2CS does not assume that all entities in the real world should have an internal representation in the system (as we can figure out by the examples in Section 4). In the experiments performed with both systems, we observed that the model proposed by JCAF results in the creation of a great number of lightweighted objects. On the other hand, J2CS fosters an event-oriented programming model, based on a smaller number of coarse-grained objects. Besides, J2CS uses Jini for service lookup, which provides a loosely couple between clients and servers. J2CS also supports rules that provide lazy and dynamic association between context monitors and contextlets. On the other hand, JCAF includes a control access component, which does not exist in J2CS.

Other context-aware middleware: RCSM (Reconfigurable Context-Sensitive Middleware) proposes a Context-Aware IDL (CA-IDL) to determine which context to monitor and which action to trigger when a particular condition is valid [23]. CA-IDL compilers are used to build context-aware applications called adaptive object containers (ADC). In many concerns, we consider RCSM more similar to the middleware for Gaia described before than to J2CS.

Carisma [3] is a middleware that relies on reflection to support the development of context-aware applications for mobile devices. In the system, configuration files are used to specify policies used to adapt the behavior of an application. Thus, the system focus on adaptation from changes that the application itself is able to detect and forecast. MiddleWhere [13] is a middleware that fuses and resolves conflicting information provided by different location technologies (GPS, badges, login information etc).

8 Conclusions

Ubiquitous spaces saturated with sensors, actuators, microcontrollers, wireless networks and portable computing devices are each day more present in univer-

sities, enterprises, public agencies, home networks etc. In order to program and integrate such spaces, software engineers will certainly need the support of middleware infrastructures. More specifically, we believe that middleware platforms are a powerful tool to leverage the design, implementation and execution of context-aware applications, i.e., applications that can take advantage of the variety of information provided by ubiquitous computing spaces. Traditional middleware platforms, such as CORBA [11] and Java RMI [22], were not designed to support the implementation of applications in such environments. It is a general belief that such platforms lack suitable abstractions to collect, interpret, synthesize and disseminate context information.

In this article, we have discussed the architecture, the programming interface and the implementation of J2CS, a middleware that supports the implementation of context-aware applications in Java. Inspired by traditional component-based middleware systems, J2CS provides an infrastructure that encapsulates several details inherent to context handling in ubiquitous computing environments. In the described system, containers are used to handle communication with context monitors and to route context information to small Java applications named contextlets. Such applications are implemented and deployed by middleware users in order to store, interpret, aggregate and disseminate context information required by the particular context-aware applications they are designing.

Contextlets can be dynamically installed and removed from containers, increasing the flexibility and extensibility of the system. In other words, contextlets can be considered mobile agents that move to containers in order to interpret and publish context information. Moreover, J2CS relies on lazy rules to associate context monitors to contextlets. The system is fully integrated with the Java programming environment, using several of its APIs. Among them, we can mention JDBC (for database access), RMI (for remote communication), Jini (for service location), serialization (for contextlets migration) and reflection (for supporting type-based event subscription). The system relies on standard Java objects to model and store context information. As the examples in Section 4 show, an object oriented language can support the implementation of several patterns common in context-aware applications.

J2CS programming is founded on an event-based (or publish/subscribe) communication mechanism. We believe that the decoupling in time, space and synchronization provided by event systems are suitable for open and dynamic environments as those typical in ubiquitous computing [6]. Besides, we believe that a programming model centered on events, conditions and actions (ECA) [12, 8], as common when programming contextlets, has expressive power to deliver simple and efficient implementations for most context-aware applications. On the other hand, context-aware applications in J2CS can use synchronous communication mechanisms, for example to query the state of contextlets or context monitors.

In the current implementation of the system, the J2CS container is executed in a centralized server. It could be argued that this fact limits the scalability of the platform. However, the system was designed to support the execution of context-aware applications restricted to a specific administrative domain, such as an university department. As the experiment of Section 6 shows, we believe that scalability concerns are not critical in such environments. However, we have plans to work on a version of the system for use in multiple administrative domains. Basically, this version will be based on the notion of federated lookup services of Jini [19]. This notion allows the creation of interlinked communities mirroring working domains. Each community will have its own container (and its own Jini lookup service).

In ubiquitous settings, most context information should be protected from unauthorized access [10]. Moreover, it is important to handle privacy concerns and ambiguity in context information. Thus, middleware for context-aware applications should provide services such as authentication, access control, safe communication channels and sensor fusion mechanisms. Since our first objective was to provide a toolkit for the rapid prototyping of context-aware systems, the current J2CS implementation does not support any of these services and hence we intend to incorporate them in the near future. Finally, we intend to transform contextlets in persistent components such as CMP (Container-Managed Persistence) and BMP (Bean-Managed Persistence) beans in EJB [18].

Acknowledgments: This research was supported by a grant from The State of Minas Gerais Research Foundation (FAPEMIG).

References

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

[2] J. E. Bardram. The Java Context Awareness Framework (JCAF). In *Third International Con-*

ference on Pervasive Computing, volume 3468 of *Lecture Notes in Computer Science*, pages 98–115. Springer, 2005.

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

[4] A. K. Dey and G. D. Abowd. Towards a better understanding of context and contextawareness. In *Workshop on the What, Who, Where, When and How of Context-Awareness, affiliated with the CHI 2000*. ACM Press, 2000.

[5] A. K. Dey, G. D. Abowd, and D. 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.

[6] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, 2003.

[7] A. Harter, A. Hopper, P. Steggles, A. Ward, and P. Webster. The anatomy of a context-aware application. *Wireless Networks*, 8(2-3):187–197, 2002.

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

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

[10] M. Langheinrich. Privacy by design - principles of privacy-aware ubiquitous systems. In *Third International Conference on Ubiquitous Computing*, Lecture Notes in Computer Science, pages 273–291. Springer, 2001.

[11] Object Management Group. The common object request broker: Architecture and specification revision 3.0.2, Dec. 2002.

[12] N. W. Paton, F. Schneider, and D. Gries, editors. *Active Rules in Database Systems*. Springer-Verlag New York, Inc., 1998.

[13] A. Ranganathan, J. Al-Muhtadi, S. Chetan, R. H. Campbell, and M. D. Mickunas. Middlewhere: A middleware for location awareness in ubiquitous computing applications. In *ACM/IFIP/USENIX International Middleware Conference*, volume 3231 of *Lecture Notes in Computer Science*, pages 397–416. Springer, 2004.

[14] A. Ranganathan and R. 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.

[15] M. Román, C. K. Hess, R. Cerqueira, A. Ranganathan, R. H. Campbell, and K. Nahrstedt. Gaia: A middleware infrastructure to enable active spaces. *IEEE Pervasive Computing*, pages 74–83, Oct-Dec 2002.

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

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

[18] Sun Microsystem. Enterprise Java Beans specification (version 2.1), Nov. 2003.

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

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

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

[22] A. Wollrath, R. Riggs, and J. Waldo. A distributed object model for the Java system. In *2nd Conference on Object-Oriented Technologies & Systems*, pages 219–232. USENIX, 1996.

[23] S. S. Yau, F. Karim, Y. Wang, B. Wang, and S. K. Gupta. Reconfigurable Context-Sensitive Middleware for Pervasive Computing. *IEEE Pervasive Computing*, 1(3):33–40, July–September 2002.