Service-Oriented Autonomic Pervasive Context

Colin Aygalinc, Eva Gerbert-Gaillard, German Vega, Philippe Lalanda

Service-oriented autonomic pervasive context

Colin Aygalinc, Eva Gerbert-Gaillard, German Vega, and Philippe Lalanda

Grenoble Alpes University
first-name.name@imag.fr

Abstract. Pervasive computing promotes environments where smart, communication-enabled devices cooperate to provide services to people. Due to their inherent complexity, many pervasive applications are built on top of service-oriented platforms, providing a set of facilities simplifying their development and execution. In this paper, we present such a platform, iCasa, extended with an autonomic, service-oriented context module. This module is programmed with a domain-specific service-oriented language built on top of iPOJO, the Apache service-oriented component model. It is validated on smart home applications developed with the Orange Labs.

Keywords: pervasive computing, context, service-oriented components

1 Introduction

A growing number of smart, communication-enabled devices are integrated in our living environments. This is essentially due to major advances in hardware and networking technologies, which make sensors more powerful, cheaper and smaller in size. Such digitalized environments, said to be pervasive or smart, are increasingly accepted in all places where social or professional activities take place. They support the creation of new added-value services that are delivered anytime and in a non-obstructive way. This new form of computing is raising huge economical and societal expectations in domains like manufacturing, buildings and homes, energy, commerce, and even healthcare. Applications integrated in such smart, pervasive environments are context-aware by essence. They are able to adapt their behaviors and the provided services according to environmental conditions [1]. The notion of environment should be taken in its broad sense here. It includes any information that can be of interest for an application like, for instance, physical quantities, locations and expectations of human beings (implied or not in the functions provided by the application), the software itself, and even remote digital resources. Initially, context was essentially limited to location-awareness [2] and to information collection. Since then, it has evolved towards more elaborated models. As a consequence, developing and evolving context information for pervasive applications is still very challenging. This explains why research into context-awareness is a firmly grounded activity in computer science and will continue to expand with the recent emergence of Internet of Things, IoT [3].

Context evolutions, in terms of models and supporting technologies, actually follow architectural evolutions of pervasive applications. Indeed, these applications are today very distributed from devices to cloud facilities, going through multiple gateways [4]. Contextual information is used at every level but with different requirements. For instance, at the cloud level, focus is on large scale and completeness whereas, at the fog level [4], concerns are more about reactivity and security. This leads to different ways to represent, query, build, and update contextual information. In our work, we focus on the fog level where contextual information is usually manipulated for two different purposes. First, it is used to implement local actions with stringent real-time requirements. Then, part or all of the gathered information is aggregated, synthesized and sent to the cloud level for longer-term analysis. In this paper, we propose a novel approach based on service-oriented and autonomic computing in order to better manage context at the fog level. Our solution relies on an architecture where context is a programming module, connected to the physical environment and publishing information as a dynamic set of services. Context is dynamic in order to reflect the changing nature of the execution environment but also to deal with applications evolving needs. Context also includes event-based facilities to make consumer applications aware of contextual evolutions. Finally, context is autonomic in the sense that it can self-adapt to those evolutions. Our solution is seamlessly integrated in a pervasive platform, called iCasa, used by our industrial partners (Orange and Schneider Electric). It heavily uses the facilities of the iPOJO service-oriented component model [5]. In order to ease context development, we have also developed a Domain-Specific service-oriented Language (DSL) allowing the straightforward definition of a context module and of its autonomic capabilities.

The structure of this paper is the following. The coming section provides background about our application domain and about the notion of context in pervasive computing. Section 3 presents the overall approach defended in this paper. Then, the following section gives details about the way it is implemented in iCasa. Section 5 is about the evaluation of the proposed approach. It has been experimented on smart home use cases, defined with the Orange Labs. Finally, related work is developed in section 6, and section 7 concludes this paper.

2 Background

Our research deals with pervasive applications in intelligent environments like smart homes, smart buildings or smart manufacturing (industry 4.0). These applications are now widely distributed, from the sensors up to cloud facilities. Some code is executed at the edge of the network, in an Internet gateway for instance, while other code is run in computing farms. More and more, code is also executed in intermediary machines in order to prune the volume of data to be transmitted in the cloud, and perform mediation operations. Depending on the code location, various forms of context are needed. They are based on different formalisms, different real-time constraints, and different interaction pat-

terns between context providers and context consumers. Current architectures are illustrated here after by Fig. 1.

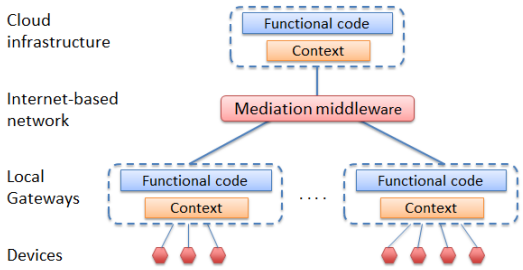

Fig. 1. Pervasive applications architecture.

To make these various needs more concrete; let us focus on an actimetrics application that we have been investigating for years with the Orange Labs. Two major functions can be distinguished for actimetrics: the first one is about early diagnosis of degenerative diseases like Alzheimer whereas the second one is concerned with real-time supervision of people at home. The first function, concerned with identification of degenerative diseases, deals with long-term evolutions spanning several months. It requires complex time-series and event correlation analysis, and is based on a rich, slowly evolving context that is explicitly accessed and browsed by the analysis algorithms. The second function is about real-time supervision. It deals, for instance, with fall detections or automated alerting in case of unusual events like prolonged inactivity or irregular sleep hours. This second function may use the same environmental data as the first one, but it is has to deal with stringent real-time constraints: new information should be made available very rapidly to the application. This is made particularly difficult by the inherent dynamicity of pervasive environments.

Let us now define more precisely what we mean by context. Context is traditionally presented as a synchronized description of concepts and relationships between them pertaining to the execution environment. Precisely, contextual information comes from computing environment (memory, network, etc.), user environment (location, needs, preference, etc.) and physical environment (temperature, noise, etc.). It can be the description of a fact, a physical object, a physical value, or an event.

There are several architectural approaches to build context-aware applications. A popular solution is to separate context and applications into different programming modules (see Fig. 2). Context-specific tasks like information gathering (context acquisition), information processing through inferences or explicit code-based operations, information storing, and information presentation lie outside the application boundaries. This pattern where context and applications are clearly separated improves code readability, debugging and evolution. It also allows context sharing between pervasive applications.

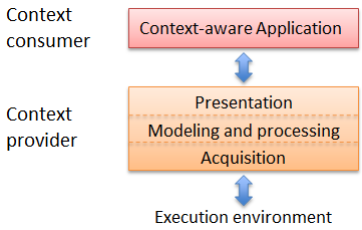

Fig. 2. Separation of context and applications.

We subscribe to this architectural approach but we readily acknowledge that building a context module is complex. Multiple design trade-off decisions have to be made and implemented regarding data access, synchronization mechanisms, knowledge representation, reasoning, and presentation. A common solution to alleviate these difficulties is to use a context management framework. Its purpose is to deal with a number of generic features like information synchronization or publication. In our work, we concentrate on the fog level, where reactivity is essential. The supporting framework has to deal with dynamic environments, where resources can appear and disappear without notice, and dynamic applications that can be launched or de-activated anytime. Specifically, we have identified the following requirements:

- Coupling between applications and context middleware must be as loose as possible. Applications should not be aware of information sources. They have to be aware of the availability (or not) of the information and its quality.
- The context middleware must present the information in a format understandable by the application and in the expected quality (level of security or preciseness for instance) that can change overtime.
- The context middleware must be able to adapt dynamically in order to provide the best contextual information depending on the application needs without interruption of service.
- The context middleware must adapt autonomically since, in pervasive settings, no administrator is available or skilled enough to perform management operation. For instance, the middleware must be able to adjust synchronization frequencies depending on the application needs.
- The context middleware must be able to activate or deactivate the sources of information, in particular to save energy. Sensors that are not connected to the mains power source should be activated only when necessary.
- The context middleware must be able to alert running applications when contextual information of interest has changed since physical environment can change anytime without notice.

We believe that service-orientation coupled with autonomic features is well suited to meet these requirements. Our interest in services is twofold. We aim to present context as a service to pervasive applications. We also aim to implement the context module with service-oriented components to enhance self-adaptation of the context.

3 Overall approach

We have integrated an autonomic, service-oriented context module in our pervasive platform, named iCasa [6][7]. This platform builds upon the Apache service-oriented component model, iPOJO. Context appears as a dynamic set of services. Depending on the sources availability and the platform needs, different services are published in and withdrawn from the platform service registry. They are then opportunistically used by the pervasive applications, coded in iPOJO. As illustrated, the context module receives goals from the platform that are used for context self-adaptation. The context module tracks any contextual modifications and sends an event to alert consumer applications. The overall approach is illustrated by Fig. 3. In our solution, services are made available in the iPOJO registry.

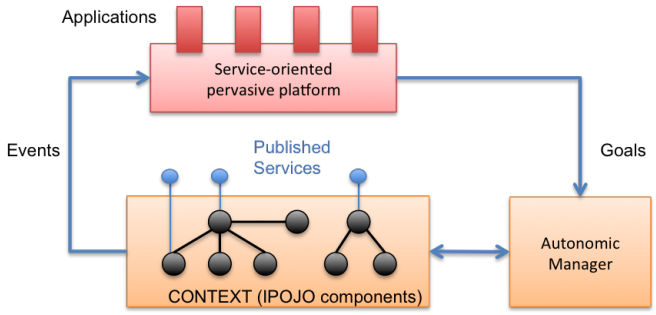

Fig. 3. Overall approach.

A major driver of our approach is not to go beyond what is necessary in order to achieve the applications expectations. This is the whole purpose of the autonomic management. It should only create and keep up to date the services of interest for the current applications, depending on the available sources. Our approach also enforces application development through dynamic composition of context services. An application specifies its required context services and, at run time, it is bounded to an appropriate service and kept informed of every evolution.

Regarding implementation, the context module is also based on the iPOJO service-oriented component model. Context is modeled as components representing concepts and relationships. Some of them can publish services. Others are used to compute information and are not proposed as external services. IPOJO containers include some autonomic features in the dynamic selection of services. Also, iPOJO includes touchpoints in the containers that can be used by more global autonomic manager. There are then several autonomic loops in the context module. It appears however that implementing the context and the associated autonomic manager is still rather complex, especially regarding the

timing aspects. We have then developed a Domain Specific Language (DSL) to cope with identified context specific concerns.

4 Context Management Domain Specific Language

The purpose of the proposed DSL is to enforce the architectural pattern described in the previous section, and to help developers with the most common tasks found in context management. The DSL is an extension of the iPOJO Component Model.

4.1 Service Oriented Component

An iPOJO component is implemented as a plain Java class decorated with specific annotations to specify non-functional concerns. The set of Java annotations can be regarded as the concrete syntax of the DSL. The base iPOJO annotations support the dynamic service interaction pattern. To illustrate iPOJO core, the following code shows a simplified Light-follow-me using two context services: lights and presence.

An application component is implemented by a Java class decorated with the annotation `@Component`. Component dependencies are specified using the `@Requires` annotation. A dependency is specified in terms of a service specification defined by a Java interface (`BinaryLight` and `PresenceService` in the example). Each dependency is associated with a field of the class (`binaryLights` and `presenceServices`), which will be bound at runtime to the selected service. The field can then be used to transparently access the required service. Dependencies can specify filters to match the available service providers.

Java code for service-oriented components

```
@Component(name="LightFollowMeApplication")
@Instantiate
public class LightFollowMeApplication {

  @Requires(id="lights", optional = true,specification = BinaryLight.class,
 filter = "(!(locatedobject.object.zone=LOCATION_UNKNOWN))")
  private List<BinaryLight> binaryLights;

  @Requires(id="presence", optional = false,specification = PresenceService.class)
  private List<PresenceService> presenceServices; }
```

4.2 Context Service Description

Application development relies only on service descriptions to reduce coupling with the context. The example here after illustrates a simplified presence detection service. The service is specified as a Java interface, annotated with the `@ContextService` marker. In this case, the interface proposes a single method (`presenceInZone`) to get the current detection status, in a zone associated with the context provider.

Java code for context service description

```
public @ContextService interface PresenceService {

  public @State static final String PRESENCE_SENSED = "presence.sensed";
  public @State static final String ATTACHED_ZONE = "zone.attached";

  public PresenceSensing presenceInZone();
  public enum PresenceSensing {YES,NO,NOT_MEASURED}}
```

As mentioned earlier, context management services also require event notifications. To do so, we have extended service specification with declaration of context service states. Changes in states are notified to consumers with events. In the example, we have declared (using the annotation `@State`) two states associated with the presence service: the presence status (`PRESENCE_SENSED`) and the detection zone (`ATTACHED_ZONE`). Note that the context service specification is a contract between the context provider and the requiring component. It is the responsibility of the context provider to generate appropriate events when state changes.

4.3 Context Service Usage

Applications use iPOJO facilities to react to context evolution. Two kinds of context events are of interest to applications: availability/unavailability of context providers, and changes in context service state. The following code example shows how the Light Follow Me application is programmed to react to changes in the context.

Java code for dynamic context service usage

```
@Component(name="LightFollowMeApplication")
public class LightFollowMeApplication {

  @Bind(id="presence")
  public void bindPresence(PresenceService presenceService){
    managelight(presenceService);}

  @Unbind(id="presence")
  public void unbindPresence(PresenceService presenceService){
    Set<BinaryLight> lightInZone = getLight-InZone(presenceService);
    lightInZone.stream().forEach((light) ->light.turnOff());
  }

  @Modified(id="presence")
  public void modifiedPresence(PresenceService presenceService){
    managelight(presenceService); }

  private void managelight(PresenceService presenceService){
    Set<BinaryLight> lightInZone = getLight-InZone(presenceService);
    if (presenceService.presenceInZone().equals(YES)){
      lightInZone.stream().forEach((light) ->light.turnOn());
    }else {
      lightInZone.stream().forEach((light) ->light.turnOff());
    }}}
```

Notification of providers availability is declared using the `@Bind` and `@Unbind` annotations. The annotated bind method is invoked each time a new matching provider is registered in the service registry (respectively, unbind methods are invoked when the provider is unregistered). In the simplified scenario for instance, when a new presence service provider is added to the context, the application simply turns on/off lights in the zone according to the current sensed presence status. Interest in changes of the state of the context service is declared using the `@Modified` annotation. The annotated method is invoked each time a state change event is triggered by the provider. The specified callback receives a reference to the source of the event and, optionally, the state that was modified.

Note again that the developer only declares interest in a given context event. All the code concerning callback registration and invocation is handled by the iCasa platform runtime. Notice also how, inside the event callbacks, the developer can directly use the injected dependency fields to transparently access the context services. The declarative nature of the DSL greatly reduces common errors and simplifies the programing model, as discussed in the evaluation section.

4.4 Context Entity

Developers implementing a context service must meet specific requirements. In general, they need to interact with physical devices; this interaction requires error-prone code for synchronization and event handling. We have developed specific DSL extensions to cope with this need. To illustrate this, the code of a component implementing the `BinaryLight` context service using the ZigBee protocol is presented. We assume that the code dealing with ZigBee is encapsulated a `ZigbeeDriver` component. As previously said, a context provider is implemented as an iPOJO component, using the `@ContextEntity` annotation to declare the provided context services (a single component may implement several context services). The component has to implement all methods and states declared in the context service declaration.

Java code for context entity implementation

```
@ContextEntity(services = {BinaryLight.class})
public class ZigbeeBinaryLight implements BinaryLight, ZigbeeDeviceTracker {

  @ContextEntity.State.Field(service = BinaryLight.class,
state = BinaryLight.POWER_STATUS, value = "false")
  private boolean powerStatus;

  @Requires
  private ZigbeeDriver driver;

  @Override
  public boolean getPowerStatus() {return powerStatus;}

  @Override
  public void turnOn() {  powerStatus = true;}

  @Override
  public void turnOff() {  powerStatus = false;}
```

```
@ContextEntity.State.Apply(service = BinaryLight.class,state = POWER_STATUS)
Consumer<Boolean> setPowerStatus = newPowerStatus -> {
  if (newPowerStatus) {
    driver.setData(moduleAddress, "1");
  } else {
    driver.setData(moduleAddress, "0");
  }
};

public void deviceDataChanged(String address,Data oldData, Data newData) {
  if(address.compareTo(this.moduleAddress) == 0){
    pushPowerStatus(newData.getData());
  }
}

@ContextEntity.State.Push(service = BinaryLight.class,state = POWER_STATUS)
public boolean pushPowerStatus(String data){
  return data.compareTo("1")==0? true : false;
}

@ContextEntity.Relation.Field(owner = LocatedObject.class)
@Requires(id="zone", specification=Zone.class, optional=true)
private Zone zoneAttached;}
```

A common implementation pattern is to maintain within the component an in-memory representation of the current state of the environment, and keep this representation synchronized. Our DSL supports this pattern using state fields. In the code example, the component implementation class `ZigbeeBinaryLight` declares `powerStatus` to keep the current lamp state (on/off). State fields are marked with the `@ContextEntity.State.Field` annotation, but otherwise behave as normal Java fields. For instance, the service method `getPowerStatus` simply returns the current field value, and methods `turnOn` and `turnOff` directly modify it. Any modification to the in-memory field must be reflected on the environment, using the corresponding ZigBee actuator. The code performing synchronization is specified with the `@ContextEntity.State.Apply` annotation that provides a function which is invoked each time the associated field is modified. In the example, every time `powerStatus` is modified (using the `turnOn` and `turnOff` methods) the `setPowerStatus` function is invoked, which will in turn delegate to the ZigBee driver to do the actual action.

In general, environment synchronization is bi-directional. So, sensed changes in the environment must equally be reflected in the in-memory state representation. In our example, lights can be physically turned on/off using a mechanical button; this is detected by a ZigBee sensor associated with the lamp. Depending on the capabilities of the sensor and device protocol, environment information can be gathered synchronously or asynchronously.

The example shows an asynchronous update. Here, the protocol driver notifies the context entity (by invoking method `deviceDataChanged`) of changes in the status of the lamp, and from the raw data the component calculates the new state value by using the method `pushPowerStatus`. This method is declared using the annotation `@ContextEntity.State.Push` that associates it with a corresponding context state field. Each time the method is invoked, the

return value is used to update the in-memory state representation. Synchronous updates declared with `@ContextEntity.State.Pull` are similar, except that the declaration specifies the periodicity used to calculate the new state value.

Using this declarative approach has two main advantages: it reduces the programming complexity and it allows autonomic adaptations at runtime. For instance, if a context entity is not used by any application at a given moment, the iCasa autonomic context manager can decide that updating the in-memory fields is not needed, avoiding unnecessary polling. A context entity can be related to other context entities. This is naturally supported in iPOJO as a service requirement. In the code example, the `ZigbeeBinaryLight` entity is related to a `Zone` context service, using the `zoneAttached` dependency field. This field is marked with `@ContextEntity.Relation.Field` to express that it can be handled by the context autonomic manager.

4.5 Context service provisioning

Context service provisioning is the process of deploying, instantiating, and relating context entities. This process is guided by external events: a device joining the network, a new application deployed in the platform, or an explicit demand by the platform administrator.

Part of this process can be automated using approaches like RoSE [8] and MUSIC [9] which provide a pattern to modularize and maintain the discovery of external events at runtime, but no specific support is provided for dynamic instantiation of context service providers. iPOJO runtime supports this behavior, but its establishment remains highly technical, tightly coupled to the iPOJO model and de facto become less feasible for developers.

Our middleware provides autonomic facilities regarding this issue, without cluttering the discovery code. As shown here after, discovery code emits now instantiation requests (previously it was direct instantiation) and the middleware choose to process or stock the requests according to the application contextual service requirements.

Java code for a context service dynamic provisionning

```
private @Creator.Field Creator.Entity<ZigbeeBinaryLight>
    binaryLightFactory;

public void zigbeeDiscoveryEvent(Map<Parameter> param){
  String id = ...;
...
  binaryLightFactory.create(id,param);
}
```

5 Runtime support

In this work, we have extended the iPOJO runtime. Precisely, we implemented two additional handlers (see Fig. 4):

– *A Synchronization Handler* deals with state synchronization of entity components. It keeps the states up-to-date by managing the synchronization functions. Different strategies can be specified to do so. For example, the handler can periodically call pull functions or just wait for push callbacks to keep the state up-to-date. Additionally, the handler is in charge of publishing states as service properties. This publication has two main interests: it allows the processing of more advanced filters and state updates can be reported to the application without the burden of an Observer pattern, by relying on iPOJO notification mechanism.
– *A Relation Handler* is in charge of the dynamic service binding of relations.

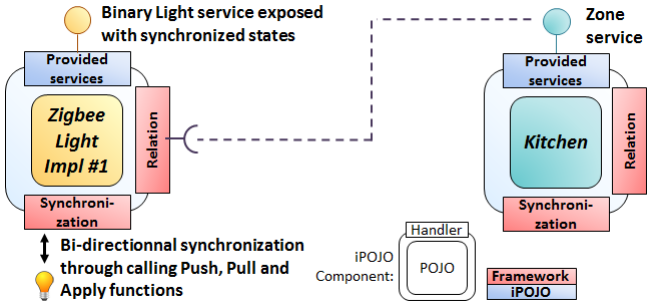

Fig. 4. Service-oriented component view of context model.

To cope with changing runtime condition, we implemented autonomic behavior at two levels. First, local autonomic loops are executed in the component container. Their goal is to locally modify the topology of the context graph by using service substitutability and late binding. This behavior is interesting for abstract context entities. It can help to increase measurement relevance by correlating new low level context sources. The global autonomic loop is implemented in the context manager. Its goal is to dynamically adapt the global topology of the context graph and the configuration parameter exposed by our DSL in order to satisfy, in a best effort way, the application provisioning. To do so, we assume that applications are developed following the iPOJO model. It involves that applications specify their needs in terms of context service dependency. The context manager can adapt the running context graph to fit applications needs. Currently, it is possible to dynamically realize the following changes:

– Enable or disable context entity provisioning;
– Modify specific synchronization parameters;
– Replace context providers.

All of this adaption logic is hard coded in our context manager. We are currently investigating integration with dynamic deployment to provide finer grained context management.

6 Validation

For evaluation purposes, we defined two scenarios that have been implemented with traditional SOCM and with our DSL. The evaluation focused on design time activities. Specifically, we used the following metrics: number of lines of code, cyclomatic complexity, and technical debt (evaluation of the effort needed to fix all issues). These metrics are computed and provided by the SonarQube [10] management tool. The first scenario deals with the iCasa platform and its associated simulator. Precisely, the following functionalities are provided:

- A set of abstraction for devices, location, user, and their implementations;
- A web interface acting as a dashboard;
- A script language allowing to dynamically instantiate simulated device, location, and user.

In the reference development, contextual information was computed in an *ad hoc* way. It was coded without any specific strategy in a non-modular way (to be fair, we reused an existing implementation). It turned out to be difficult to extend and evolve. We then redesigned the code, using our DSL, and compared the two versions (see figure see Fig. 5).

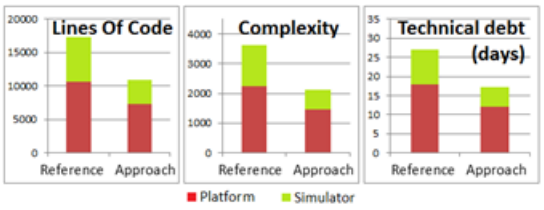

Fig. 5. Evaluation on iCasa platform and simulator.

Thanks to annotations, the number of lines of code decreased. By clearly identifying synchronization functions and limiting their number, cyclomatic complexity has been reduced too. We also noticed that the restructured implementation presents a high percentage of duplicated lines (approach 7%, reference 3%) due to iPOJO technical limitations: it doesnt support inheritance. The number of lines could therefore be reduced more. Our approach notably lightens the context layer development. It offers non-functional technical facilities. The context is modularized, extensible, and autonomic. The whole software is more consistent, testable, and maintainable.

The second evaluation compares two versions of the Light Follow Me application build upon the reference and the new platform (Fig. 6). This application is simple, yet it encompasses all requirements presented before. It is a typical home pervasive application that does not need complex reasoning algorithm but has to face the dynamism of the environment and directly influences the user

environment. In the reference implementation, the application processes information like the presence per zone by directly reasoning over the sensors and their location. In the new implementation, we decided to externalize this processing with a dedicated presence per zone context service (blue part on Fig. 6). This presence-per-zone service can be shared between applications and evolves independently of the business code of applications.

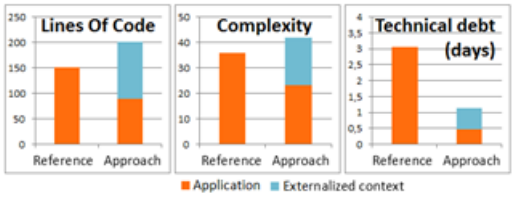

Fig. 6. Evaluation on Light Follow Me application.

Modularizing the presence service brings an overhead in terms of lines of code and complexity. This is due to the fact that service provisioning must be implemented in our solution. However, this overhead can be shared between several applications. So, if we analyze only the application business code (orange part on Fig. 6), it is approximately divided by 2 and de facto becomes easy to test, maintain, and evolve.

As a conclusion, externalizing context requires an additional development task and the resulting architecture is more complicated, but this cost can be mutualized and shared among several applications. Moreover, new applications can be develop on top of more abstract services, which facilitates their implementation.

7 Related Work

Many surveys [1][3][11][12][13] about context middleware have been proposed. We compare our approach to those whose architecture can be adapted to a fog computing environment and its specific constraints.

The Context Toolkit [14] promotes code-reuse through the composition of distinct artifact called widgets to build the context. These widgets are used to hide the complexity of sensors and abstract context information. These reusable blocks are explicitly linked at design time, each block deciding which blocks to use. Our approach is similar in the sense that we divide the context in individual small pieces. However, we delegate composition at runtime with more variability expressed at design time. Moreover, Entity-Relation-like model offers more flexibility to design complex contexts.

COSMOS, COntext entitieS coMpositiOn and Sharing [15], is a component based context middleware. Each pieces of context is reified as a component called

Context Node organized in a hierarchical structure. This approach provides separation of concerns by offering several built-in mechanisms like push/pull notifications. However, the strictly hierarchical approach of COSMOS context makes it difficult to model horizontal relations. Moreover, component specifications are strictly defined at design time, so runtime extensibility is hard to achieve.

DiaSuite [16] is a component based tool suite using a DSL, DiaSpec. DiaSuite, following the Sense/Compute/Control pattern, defines three primitive types of component: resource, context, and controller. DiaSpec is used to describe the structure of each component, and through an additional build time step, to generate the component skeleton. DiaSuite is similar to our approach in the sense that its main goal is to help developers. However, many things remain on developer side like programming of runtime component binding, and there is no support for synchronization. We believe that this behavior must be specified and not programmed in order to ease runtime reconfiguration. Also, java based annotations seem to be better accepted by developers.

ACoMS, Autonomic Context Management System [17] promotes a work dealing with fault-tolerance as regards to the dynamism of context provisioning by using autonomic behavior. Applications describe their needs in terms of context fact, and ACoMS can autonomously configure and reconfigure its context acquisition and pre-processing functionality. ACoMS promotes autonomous behavior, but context sources are at a sensor level and no clear guidelines are provided to construct more abstract concepts. Moreover, we think that by infusing autonomic touch points at a finer granularity, more advanced autonomic behavior can be brought to context management.

[18] work deals with proactive adaptation and context management based on a SOCM architecture. It underlines the fact that context interactivity is not just about providing the most powerful modeling and reasoning engine. Indeed, applications also can deal with context in a proactive manner, with the ability to change the context through actuators. Our approach, in this sense, is very similar. To do so, a specific query language is provided, with the issue of a steep learning curve.

8 Conclusion

In this paper, we have presented a service-based architecture to design context-aware applications. We have also described a Domain-Specific Language facilitating the development of a context module in iPOJO. These facilities are seamlessly integrated in the iCasa platform and tested on real-size applications with the Orange Labs.

We are now working in two complementary directions. First, we are trying to model more complex contextual entities. In particular, we believe it will be soon necessary to include IoT and other pervasive platforms in the context since these artifacts will be more and more present in smart environments. Second, we are seeking to better formalize the application possible adaptations as a function of the available contextual information and associated quality.

References

1. Baldauf, M., Dustdar, S., Rosenberg, F.: A survey on context-aware systems. IJAHUC **2**(4) (2007) 263–277
2. Bauer, M., Becker, C., Rothermel, K.: Location models from the perspective of context-aware applications and mobile ad hoc networks. Personal and Ubiquitous Computing **6**(5/6) (2002) 322–328
3. Perera, C., Liu, C.H., Jayawardena, S., Chen, M.: A survey on internet of things from industrial market perspective. IEEE Access **2** (2014) 1660–1679
4. Bonomi, F., Milito, R.A., Zhu, J., Addepalli, S.: Fog computing and its role in the internet of things. In Gerla, M., Huang, D., eds.: Proceedings of the first edition of the MCC workshop on Mobile cloud computing, MCC@SIGCOMM,, Helsinki, Finland, ACM (August 2012) 13–16
5. Escoffier, C., Hall, R.S., Lalanda, P.: ipojo: an extensible service-oriented component framework. In: 2007 IEEE International Conference on Services Computing, Salt Lake City, IEEE Computer Society (2007) 474–481
6. Escoffier, C., Chollet, S., Lalanda, P.: Lessons learned in building pervasive platforms. In: 11th IEEE Consumer Communications and Networking Conference, CCNC 2014,, Las Vegas (January 2014) 7–12
7. iCasa: platform and simulator releases
8. Bardin, J., Lalanda, P., Escoffier, C.: Towards an automatic integration of heterogeneous services and devices. In: 5th IEEE Asia-Pacific Services Computing Conference, APSCC 2010,, Hangzhou, IEEE Computer Society (2010) 171–178
9. Rouvoy, R., Barone, P., Ding, Y., Eliassen, F., Hallsteinsen, S.O., Lorenzo, J., Mamelli, A., Scholz, U.: MUSIC: middleware support for self-adaptation in ubiquitous and service-oriented environments. In Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J., eds.: Software Engineering for Self-Adaptive Systems. Volume 5525 of Lecture Notes in Computer Science., Springer (2009) 164–182
10. SonarQube: an open platform to manage code quality
11. Bettini, C., Brdiczka, O., Henricksen, K., Indulska, J., Nicklas, D., Ranganathan, A., Riboni, D.: A survey of context modelling and reasoning techniques. Pervasive and Mobile Computing **6**(2) (2010) 161–180
12. Bellavista, P., Corradi, A., Fanelli, M., Foschini, L.: A survey of context data distribution for mobile ubiquitous systems. ACM Comput. Surv. **44**(4) (2012) 24
13. Ibarra, U.A., Augusto, J.C., Clark, T.: Engineering context-aware systems and applications: A survey. Journal of Systems and Software **117** (2016) 55–83
14. Dey, A.K.: Understanding and using context. Personal and Ubiquitous Computing **5**(1) (2001) 4–7
15. Conan, D., Rouvoy, R., Seinturier, L.: Scalable processing of context information with COSMOS. In: Distributed Applications and Interoperable Systems, 7th IFIP WG 6.1 International Conference, DAIS 2007, Paphos (June 2007) 210–224
16. Bertran, B., Bruneau, J., Cassou, D., Loriant, N., Balland, E., Consel, C.: Diasuite: A tool suite to develop sense/compute/control applications. Sci. Comput. Program. **79** (2014) 39–51
17. Hu, P., Indulska, J., Robinson, R.: An autonomic context management system for pervasive computing. In: Sixth Annual IEEE International Conference on Pervasive Computing and Communications (PerCom). (2008) 213–223
18. VanSyckel, S., Schiele, G., Becker, C.: Extending context management for proactive adaptation in pervasive environments. In: Ubiquitous Information Technologies and Applications. Springer Netherlands (2013) 823–831