Context-Aware Exception Handling in Mobile Agent Systems: The MoCA Case

Karla Damasceno[1] Nelio Cacho[2] Alessandro Garcia[2]
Alexander Romanovsky[3] Carlos Lucena[1]

[1]Computer Science Department, Pontifical Catholic University of Rio de Janeiro – PUC-Rio, Brazil
[2]Computing Department, Lancaster University, United Kingdom
[3]Computer Science School, University of Newcastle upon Tyne, United Kingdom

{karla,lucena}@inf.puc-rio.br, {n.cacho,a.garcia}@lancaster.ac.uk
alexander.romanovsky@newcastle.ac.uk

ABSTRACT

Handling erroneous conditions in context-aware mobile agent systems is challenging due to their intrinsic characteristics: openness, lack of structuring, mobility, asynchrony, and increased unpredictability. Even though several context-aware middleware systems support now the development of mobile agent-based applications, they rarely provide explicit and adequate features for context-aware exception handling. This paper reports our experience in implementing error handling strategies in some prototype context-aware collaborative applications built with the MoCA (Mobile Collaboration Architecture) system. MoCA is a publish-subscribe middleware supporting the development of collaborative mobile applications by incorporating explicit services to empower software agents with context-awareness. We propose a novel context-aware exception handling mechanism and discuss some lessons learned during its integration in the MoCA infrastructure. The discussions include how to use other emerging implementation techniques, such as aspect-oriented programming, to address the limitations of classical publish-subscribe mechanisms identified in our study.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging – *Error handling and recovery*; I.2.11 [**Distributed Artificial Intelligence**]:Multiagent systems

General Terms: Algorithms, Reliability, Languages

Keywords: Exception handling, mobile agents, context-awareness, middleware, fault tolerance, aspect-oriented programming.

1. INTRODUCTION

There is a growing popularity of pervasive agent-based applications that allow mobile users to seamlessly exploit the computing resources and collaboration opportunities while moving across distinct physical regions. Typically mobile collaborative applications need to be made context aware to allow autonomous adaptation of the agent functionalities. In particular, they need to deal with frequent variations in the system execution contexts, such as fluctuating network bandwidth, temperature changes, decreasing battery power, changes in location or device capabilities, degree of proximity to other users, and so forth. However, the development of robust context-aware mobile systems is not a trivial task due to their intrinsic characteristics of openness, "unstructureness", asynchrony, and increased unpredictability [6, 18].

These system features seems to indicate that the handling of exceptional situations in mobile applications is more challenging, which in turn makes it impossible the direct application of conventional exception handling mechanisms [13, 14]. First, error propagation needs to be context aware since it needs to take into consideration the dynamic system boundaries and changing collaborative agents. Second, both the execution of error recovery activities and determination of exception handling strategies often need to be selected according to user contexts. Third, the characterization of an exception itself may depend on the context, i.e. a system state may be considered an erroneous condition in a given context, but it may be not in others.

Several middleware systems [6,9,19] are nowadays available to support the construction of mobile agent-based applications. Their underlying architecture rely on different coordination techniques, such as tuplespaces [6], publish-subscribe mechanisms [9], and computational reflection [19]. However, such the middleware systems rarely provide explicit support for *context-aware exception handling*. Often the existing solutions (e.g. [12,18,20]) are too general and not specific for the characteristics of the coordination technique used. Typically they are not scaleable because they do not support clear system structuring using exception handling contexts. Our analysis shows that understanding the interplay between context awareness and exception handling in mobile agent systems is still an open issue. As a result, in order to deal with the complexity of context-aware exceptions, application programmers need to directly rely on existing middleware mechanisms, such as interest subscriptions or regular tuple propagation. The situation is complicated even further when they need to express exceptional control flows in the presence of mobility.

We have implemented error handling features in several prototype context-aware collaborative applications built with the MoCA (Mobile Collaboration Architecture) system [9]. MoCA is a publish-subscribe middleware that supports the development of collaborative mobile applications by incorporating explicit services empowering software agents with context-awareness. This paper presents the lessons learned while developing exception handling in

SELMAS'06, May 22–23, 2006, Shanghai, China.

MoCA applications. We have identified a number of exception handling issues that are neither satisfied by the regular use of the exception mechanisms of programming languages nor addressed by conventional mechanisms of the existing context-aware middleware systems, such as MoCA. We have also analysed how complementary techniques, such as aspect-oriented programming, can be used to tackle some limitations identified in the use of those conventional mechanisms.

The main contributions of this paper are as follows. First, we present a case study helping us to identify the requirements for the context-aware exception handling mechanism. The system is a typical ambient intelligence (AmI) application developed with the MoCA middleware. Secondly, using these requirements we formulate a proposal for a context-aware exception handling model. Thirdly, we describe a prototype implementation of the model in the MoCA middleware; it consists of an extension of the client and server APIs and new middleware services, such as management of *exceptional cont*exts, context-sensitive error propagation and execution of context-aware exception handlers.

The plan of the paper is as follows. Section 2 presents the basic concepts of context awareness and introduces the fundamental exception handling concepts. Section 3 describes the case study in which we have identified challenging exception handling issues for the development of robust context-aware agent applications. Section 4 discusses an implementation of the proposed mechanism in MoCA. Section 5 overviews the related work. Section 6 concludes the paper by discussing directions of future work.

2. BACKGROUND

This section discusses the background of our work. Section 2.1 introduces the terminology and a categorization of the context-aware middleware. Section 2.2 overviews the MoCA system. Section 2.3 introduces the exception handling concepts used in this paper.

2.1 Context and Context-Aware Middleware

The concepts of context and context-aware systems have been defined in a number of ways (e.g. [2, 3, 4]). According to Dey and Abowd [1], context is any information that can be used to characterize the situation or an entity. A system is context-aware if it uses context to provide relevant information and/or services to the user. Thus, one entity can be represented by an agent or a person with a mobile device and the context-aware system can provide information about location, identity, time and activity for these entities.

Before the context can be used it is necessary to acquire data from sensors, conduct context recognition and some other tasks [5]. These tasks are usually implemented by context-aware middleware, which hides the heterogeneity and distributed nature of devices processing the contextual information. The following session describes MoCA, a context-aware publish/subscribe middleware which has been used in our first experiment to incorporate exception handling strategies in context-aware mobile agent systems. Such an architecture was selected because of the growing number of context-aware middleware systems based on the publish-subscribe model [7,8,9].

Publish/Subscribe (pub/sub) architectures rely on an asynchronous messaging paradigm that allows loose coupling between publishers and subscribers. Publishers are the agents that send information to a central component, while subscribers express their interest in receiving messages. Broker [7] or Dispatcher [8] is the central component of a pub/sub system, and is responsible for recording all subscriptions, matching publications against all subscriptions, and notifying the corresponding subscribers.

2.2 MoCA: Mobile Collaboration Architecture

MoCa [9] is a middleware system supporting development and execution of the context-aware collaborative applications which work with mobile users. Figure 1 shows the three elements that compose the MoCa application: a server, a proxy, and clients. The first two are executed on the nodes of the wired network, while the clients run on mobile devices. A proxy intermediates all communication between the application server and one or more of its clients on mobile hosts. The server and the client of a collaborative application are implemented using the MoCA APIs, which hide from the application developer most of the details concerning the use of the services provided by the architecture.

To support context-aware applications, MoCA supplies three services: Context Information Service (CIS), Symbolic Region Manager (SRM) and Location Inference Service (LIS). The CIS component receives and processes state information sent by the clients. It also receives notification requests for from the application Proxies, and generates and delivers events to a proxy whenever a change in a client's state is of interest to this proxy. To provide transparency, CIS takes decisions on behalf of the publish/subscribe mechanism; which is implemented using built-in mechanisms that cater for the basic functionalities rather than deal with the high levels of heterogeneity and dynamicity intrinsic to mobile environments, such as the problem of late delivery [7].

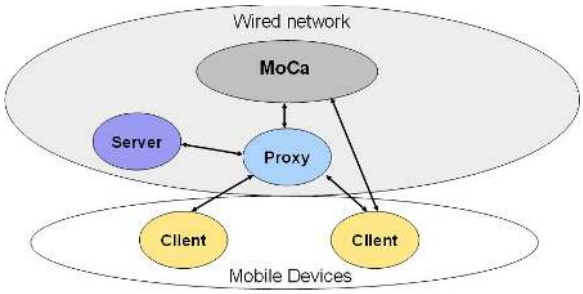

Figure 1. MoCA application

SRM provides an interface to define and request information about hierarchies of symbolic regions, which are names assigned to well-defined physical regions (i.e. rooms, halls, buildings) that may be of interest to location-aware applications [9]. Based on SRM information, LIS infers the approximate location of a mobile device from the *raw* context information collected by CIS of this device. It does this by comparing the current pattern of radio frequency (RF) signals with the signal patterns previously measured at the pre-defined *Reference Points* of the physical region. Therefore, to make any inference, the LIS database has to be populated with RF signal probes (device pointing in several directions) at each reference point, and with the inference parameters that are chosen according to the specific characteristics of the region.

2.3 Exception Handling

Agent activity, as the activity of any software component, can be divided into two parts [21]: *normal activity* and *exceptional activity*. The normal activity implements the agent's normal services while the exceptional activity provides measures that cope with

exceptions. Each agent (and other system components) should have *exception handlers*, which constitute its exceptional activity. Handlers are attached to a particular region of the normal code which is called *protected region* or *handling scope*. Whenever an agent cannot handle an exception it raises, the exception will be signaled and *propagated* to other handling scopes defined in the higher-level components of the system. After the exception is handled, the system returns to its normal activity.

Developers of dependable systems often refer to errors as exceptions because they manifest themselves rarely during the agent's normal activity. Exceptions can be classified into two types [14]: (i) *user-defined*, and (ii) *pre-defined*. The user-defined exceptions are defined and detected at the application level. The predefined exceptions are declared implicitly and are associated with the erroneous conditions detected by the run-time support, the middleware or hardware .

Exception handling mechanisms [13, 14] developed for many high-level programming languages allow software developers to define exceptions and to structure the exceptional activity of software component. An exception handling mechanism introduces the specific way in of *exception propagation* and of changing the normal control flow to the *exceptional control flow* when an exception is raised. It is also responsible for supporting different exceptional flow strategies and search for the appropriate handlers after an exception occurs. Exception mechanisms are either built as an inherent part of the language with its own syntax, or as a feature of the middleware systems coping with the intricacies of the different application domains and architecture styles.

3. Context-Aware Exception Handling in Mobile Agent Systems

This section describes a typical context-aware agent-based application, for which we have implemented a prototype system with the MoCA architecture, and identified a number of difficulties in incorporating error handling. Section 3.1 describes the case study, while Section 3.2 presents the identified requirements for a mechanism smoothly supporting context-aware exception handling.

3.1 AmI: a Case Study

The case study is an ambient intelligence (AmI) [11] application, which is composed of numerous sensors, devices and control units interconnected to effectively form a machine [10]. A wide range of sensors and controllers could be utilized, such as: fire alarm, energy control, heating control, ventilation control, climate, surveillance, lightning, power, and automatic door and window. Figure 2 depicts an AmI scenario where each office contains sensors and output devices, which are monitored and controlled locally by software agents. All these agents are connected together via a network, forming a decentralized architecture that enables building-wide collaboration.

Each piece of equipment has an associated device controlling its activation. All users have a smartcard that operates as a mobile device supplying the current position and employee ID. Immediately after entered the office, the system needs to identify the user preferences and starts the procedures for dealing with the temperature, ventilation, illumination, and climate adaptation for the specific user preferences. In order to achieve the system robustness, a number of environmental, hardware, and software-related exceptions that need to be effectively handled. Such exceptional circumstances include fire or excessive number of users in a given building region, or the occurrence of problems in the diverse primary and/or secondary mobile heating systems distributed over the building, or even in the central heating system. The handling of such exceptional conditions depend on the combination of changing contextual information, such as the location and type of the heating systems, the physical regions where the different system administrators are, and so on.

In the following, we discuss problems relative to the incorporation and implementation of error handling scenarios in such a context-aware mobile agent-based application. First, we explain the problems found in the context of our case study. Second, we explain why they cannot be addressed while using the underlying mechanisms of the MoCA architecture. The shortcomings here vary from exception declaration to exception handlers and error propagation issues.

3.2 Specification of "Exceptional Contexts"

During design of the AmI application we have identified a number of user-defined "exceptional contexts" that depend on a multitude of contextual information and also on user preferences, which in turn are typically application-specific. For us, exceptional contexts mean one or more conditions associated with the context types, which together denote a environmental, hardware, or software fault. For example, the exceptional contexts can be characterized by the situations when the temperature of an office or public room in the building occasionally exceeds the maximum limit according to user preferences, which can indicate a serious problem in the heating system (not detected by the associated controlling system). Handling of such situations requires an exceptional control flow different from the normal one, consisting of regular notification-based reactions. The seriousness of this context requires propagation of such exceptional context information to the proper administrators, which also may vary depending on their physical location. It may also require involvement of several people.

The specification of contextual conditions of interest in publish-subscribe systems, such as MoCA, requires explicit subscriptions based on regular expressions. The subscription is usually carried out by the code in the devices or proxy servers, which will be receiving notifications when those contextual conditions are matched according to the changing circumstances. However, the specification of an exceptional context situation inherently has a different semantics and, as such, needs to encompass different elements in its specification, including the handling scope, alternative "default handlers", types of contextual information which *should* and *should not* propagated together with the exception occurrence, and so on. This is why in MoCA normal contextual subscriptions need to be different from the exceptional subscriptions.

3.3 Lack of Exception Handling Scoping

There are several situations in the AmI case study (Section 3.1) when handling exceptions requires several software agents and users to be involved depending on the physical regions and other types of contextual information. For example, as discussed in Section 3.2, the proper handling of some exceptional conditions in the mobile heaters requires exceptions to be propagated to a set of devices belonging to the staff responsible for heater maintenance. However, the propagation needs to be context sensitive in the sense it should take into account the suitable maintainers for the specific heater type that is closest to the region where the faulty heater is located. The

contextual exception needs to be systematically propagated to broader scopes until the appropriate handlers are found. Moreover, if some fire exception is detected, it needs to be propagated to all the building regions and group of mobile users. Hence the physical regions or a group of devices (such as, those ones with the maintenance people) are examples of contextual handling scopes that should be supported by the underlying middleware. In this way, the proper exception handlers could be activated in all the relevant devices according to different user preferences. However, the MoCA middleware does not support such scopes for context-aware error handling, which hinders the modularity of the system on the presence of exceptional contexts.

3.4 Need for Context-Aware Handlers

There are also some cases where the selection of proper exception handlers depends on the contextual conditions associated with devices involved in the coordinated error handling. For the same exception, we need to create handlers tailored to different contextual conditions, and make sure that they are correctly executed. For instance, we need to associate contextual information about the heater physical location to the handlers dealing with the faulty heaters. Some handlers can be only selected if the mobile heater is in the context of a specific department. Again, we have to implement such a control of context-aware handlers as part of the application since there is no MoCA facility for that purpose.

4. Exception Handling in MoCA

This section presents our context-aware exception handling model and its MoCA implementation, which deal with the problems discussed in Sections 3.2 – 3.4. Our current approach basically supports the notion of *exceptional context* (Section 3.2) and *different levels of handling scopes* (Section 3.3) to treat the limitations of conventional APIs and mechanisms provided by existing context-aware middleware systems (Section 2.1).

Exceptional contexts. The goal of exceptional contexts is to facilitate the definition of exceptional situations in applications that have a great number of devices and sensors that collect information for a specific purpose. An exceptional context corresponds to undesirable or dangerous set of conditions pertaining to different contexts. They can be associated with one specific user, application's agents, or mobile devices.

Scope nesting: protected device(s), regions, or groups. In order to support a modular context-aware approach for error propagation, exceptions can be caught by scopes at four different levels: a device, a group (of devices), a proxy server, and a region. In our MoCA implementation, the central MoCA server, where the CIS and LIS services (Section 2.2) are located, is also treated as an exception handling scope. To illustrate these types of handling scopes, Figure 2 depicts how scopes of different type covers different elements of our AmI case study (Section 3.1). Device and server scopes comprise basic operational units of the context-aware system, which allows the exception handler functionality to be encapsulated into the scope of its own unit (device or server).

Group-based scopes. Group scope encompasses a set of devices which are defined by the application to support mobile cooperative handling of an exception amongst the device's agents pertaining to that group. This kind of scope is not directly related to spatial relationship, and it makes it possible to insert or remove elements from the scope according to the application necessity. Thus software agents can autonomously join and leave a group. For example, the agents acting on behalf of heat maintainers (Section 3.1) can form a specific group, as when heating-related exceptions are raised all of them may be notified.

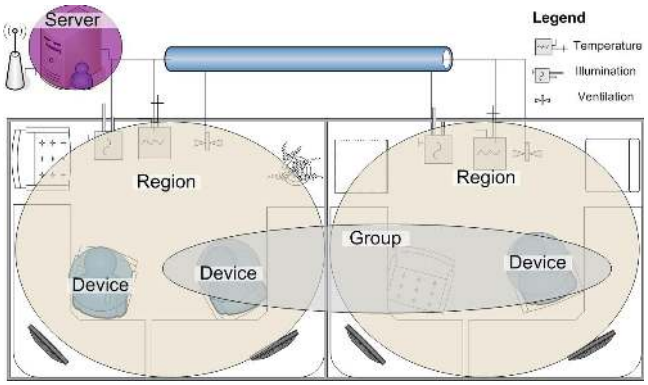

Figure 2. Different scope levels: Device, Region, Group, Server

Region-based scopes. Differently from the three first ones, a region scope has a more dynamic behavior to identify t-he devices that is part of the scope. In our implementation, this scope is strongly related to the MoCa LIS service (Section 2.2) as it provides a reference mechanism that allows a device be aware of its neighbors. In addition, it is possible to control when a device enters and goes out from one region scope. A device movement characterizes the scope change; in other words, whenever a device moves from one a physical region X to Y, it automatically moves from the region-based handling scope X to Y. Hence this movement encompasses the context-sensitive change of the exceptional conditions that the device can handle.

Regular contextual subscription. Figure 3 shows the server code responsible for defining a user preference for a specific office. Due to the user movement, it is necessary to subscribe a listener in the LIS service to notify whenever a new device enters in the office. When this occurs, server gets user preferences and starts the appropriate services to suit the user requirements. To avoid long connections between server and devices, once each device can take long time to finish the process, servers break the connection after invoking the start method.

An exceptional context example. As discussed in Section 3, a device's agent suddenly detects that the temperature is exceeding the maximum limit, and then, it infers that there is a fault in the mobile heating system. In this situation, the device should throw and propagate an exception to inform the other mobile users and the heater support about the problem. This step is done by the code shown in Figure 4. The *UnableToHeat* exception is created and thrown by the heating device.

Context-aware error propagation. It also specifies that it needs to be propagated to a set of scopes. To define the sequence of exception propagation, the user should use the *propagateTo* method. This method receives as parameter the scope reference, the sequence number, and also the condition constant. The scope reference determines for which scope the exceptional context will be propagated, the sequence number, and in which order. Scope constant defines the propagation policy in which the error propagation will occur; it determines when the exception needs to propagated: whether when none (NONE value), one (ONE value) or all handlers (ALL value) were found and successfully executed to

deal with this exception. For instance, in Figure 4, the *UnableToHeat* will firstly propagate to the region and group. If none of them handles this exception, it is delivered to the server scope.

```
RegionListener listen = new RegionListen();
Lis_service.subscribe(UserDevice, listen);
...
private class RegionListen
              implements RegionListener {
 public void onDeviceEntered(String regionID,
                             String deviceID)
   HashMap userpref = getUserPref(deviceId,
                                  regionID);
   Heat.start(userpref.get("Temperature"));
   LightCtrl.start(userpref.get("Light"));
                . . .
}}
```

Figure 3. User preference definition.

Contextual information propagation and context-aware selection of handlers. In addition, the *UnableToHeat* exceptional context can carry information related to the exceptional occurrence. This may include the operation status of the thermostat, the operation status of the tip over the safety mechanism, the heater type, and also the heater brand. This contextual information is carried with the propagated exception to allow the exception mechanism to select an appropriate context-aware handler for a specific exception. Note that, after defining the exception content, this exception is thrown in the last line of the Figure 4.

```
UnableToHeat unaheat = new UnableToHeat();
Unaheat. propagateTo((RegionScope.getInstance(),1,
Scope.NONE);
Unaheat.propagateTo((RegionScope.getInstance("HeatM
aintainer"),1,Scope.NONE);
Unaheat.propagateTo((ServerScope.getInstance("MainS
erver"),2);
Unaheat.getContext().
    setStringProperty("Thermostat","noanswer");
              . . .
EHMechanism.throw(unaheat);
```

Figure 4. Device code: throwing exception.

Context-sensitive handlers. In order to handle the *UnAbleHeat* exception, four handlers are defined in different scopes. The first one deals with the university maintenance group; it is defined by each maintainer device and informs each one whether the heaters related to the offices have a problem. Figure 5 describes the *BrandSupport* handler definition and also its association with the maintainer group of users. To define a handler in our mechanism it is necessary to extend the *Handler* abstract class. This class requires the implementation of *verifyContextCondition* and *execute* abstract methods defined in the API of our mechanism. The first method performs verification if the exceptional context is really appropriate for this handler and the second one executes the handler functionality. For instance, in our case study, each maintainer employee is responsible for a specific university region and also for a specific type of heating. For this reason, there is a handler for each employee with appropriate conditions. Therefore, when an exception is caught, the mechanism executes the *verifyContextCondition* for each handler defined in that scope. Whether this method returns *true,* the mechanism invokes *execute*, but if not, the mechanism follows to the next defined handler. The purpose of this approach is to promote extra flexibility that supports the definition of context-aware handlers. After the handler definition, Figure 5 depicts *UniversityHeaterFail* and the scope group definition. The exceptional context *UniversityHeaterFail* catch all *UnableToHeat* exception occurrences that come from *University* region and has one of the two brands (A or B). To deal with this exception, each maintainer device gets an instance of the *HeatMaintainer* scope group and adds itself to this scope. Thus, whenever *UnableToHeat* was propagated to the *HeatMaintainer*, each device can carry out the exceptional context through its context-aware handlers.

```
public class BrandSupport extends Handler {

public boolean verifyContextCondition(){
SimpleContext simple =getException().getContext().
     find("Computing Dep",
            "HeaterType = 'Ceramic'");
  if (simple != null) return true;
  else return false;
 }
  public boolean execute(){
     makeAppointment();
  }
}
                  . . .
UniversityHeaterFail branduni =
    new UniversityHeaterFail(CompositeContext.OR);
branduni.addContext("University",
         "UnableToHeat.Brand='HeaterCompanyA'")
branduni.addContext("University",
         "UnableToHeat.Brand='HeaterCompanyB'")
BrandSupport suppCeramic = BrandSupport(branduni);
DeviceGroupScope groupScope = DeviceGroupScope.
          getInstance("HeatMaintainer");
groupScope.addDeviceList(this.getMyDefice());
groupScope.attachHandler(suppCeramic);
            . . .
```

Figure 5. Group scope definition.

To be aware of what is happening in the office, the user device need to define the exception shown in Figure 6. This exception represents all exception occurrences that come from its own current region. It is and associated with a handler that informs the user about the current problem.

```
BeAwareException awarex = new BeAwareException ();
Unaheat.addContext(device.getRegion());
NotifyUsers ntusers = new NotifyUsers(awarex);
RegionScope regionScope =
                RegionScope.getInstance();
regionScope.attachHandler(ntusers);
```

Figure 6. Region scope definition.

As we can see in Figure 4, if none of the devices that are part of the group scope do not handle the *UnableToHeat* exception, it will be propagated to the server scope. To deal with this exception, Figure 7 illustrates the exception and server scope definition. The *MakExternal* handler creates an external request to fix the problem that no internal maintainer is able to satisfy.

5. Discussions and Related Work

This section provides discussions and some lessons learned from our experience, and compares our pub/sub-based approach with other existing exception handling techniques for mobile systems (Section

5.1). It also analyzes how aspect-oriented programming approaches (Section 5.2) can be used to address the modularity-related problems identified in our case study (Section 3).

```
UnableToHeat uncatch = new UnableToHeat();
MakExternal makexternal =
                  new MakExternal(uncatch);
ServerScope serverScope = ServerScope.
                        getInstance("MainServer");
serverScope.attachHandler(makexternal);
```

Figure 7. Server scope definition.

5.1 Comparison with Existing Techniques

Although our current implementation (Section 4) supports a heterogeneous set of handling scopes, their granularity may not be always appropriate. To this end we are planning to adopt role-like abstractions, as supported by the CAMA tuplespace-based middleware [6], in order to allow handlers to be attached the specific agent actions or plans. Furthermore we plan to extend our mechanism to support code mobility in addition to physical mobility.

Developing advanced exception handling mechanisms suitable for multi agent systems is an area that needs serious efforts from the research community even though there have been a number of interesting results. A scheme in [12] supports exception handling in systems consisting of agents that cooperate by sending asynchronous messages. This scheme allows handlers to be associated with services, agents and roles, and supports concurrent exception resolution. Paper [18] identifies several typical failure cases in building context-based collaborative applications and proposes an exception handling mechanism for dealing with them.

An approach in [20] is based on defining a specialized service fully responsible for coordinating all exception handling activities in multi agent systems. Although this approach does not scale well, it supports separation of the normal system behavior from the abnormal one as the service curries all fault tolerance activities: it detects errors, finds the most appropriate recovery actions using a set of heuristics and executes them. As opposed to the last three schemes above which do not explicitly introduces the concept of the exception handling context (scope), the CAMA framework [6] (introduced in Section 2.1) supports the concept of (nested) scopes, which confine the errors and to which exception handlers are attached. However, CAMA and the other mechanisms mentioned above do not support a fully context-aware exception handling, as supported by our approach (Section 4). In particular, they do not implement context-aware selection of handlers, proactive exception handling, and the definition of exceptional contexts.

5.2 Aspectising Contextual Error Handling

The fundamental driving principle underlying the conception of exception handling mechanisms has been the support for improved modularity in the presence of errors [22, 23]. However, achieving modular exception handling is hard as error detection and treatment are widely-recognized as crosscutting concerns by their very nature [16, 24]. Hence modular error handling in its own is still a challenge nowadays [16, 25, 26]; context-aware exception handling just complicates even further the scenario as it needs to be tightly coupled to the context services in the supporting middleware.

As we can see in Section 4, to define context-aware exception handling is necessary to spread code out all over the application in a fashion that relates the application context specification, exception detection, and exception handling. In our experience, the implementation of error handling strategies has interfered and crosscut different levels of abstractions in the context-aware mobile system at hand: (1) the application level was affected by detection of context-aware exceptions and dynamic handler binding (Section 4), (2) the MoCA API was affected by inserting exception-related interfaces for context defintions, and (3) the core structure in the middleware level was modified to include components responsible for managing the context-aware exceptional control flow.

In this context, aspect-oriented programming (AOP) [27] seems a promising approach to support the implementation of context-aware exception handling. For example, a specialized pointcut language could be defined to quantify and explicitly specify context-aware detections of exceptions. In a recent work, AOP has been used [15] to modularize several parts of a context-aware application, such as context acquisition, location/proximity context and application specific context. This approach allows to create aspects capable to performing context fusion/fission and adapt modeled context into non-modeled contextual information that may be required by application to perform some function. However, this work does not take context-aware exceptions into account.

In order to exploit AOP to deal with context-aware error handling issues, we are current working in two main directions: (i) identifying scenarios where AspectJ mechanisms work (and not work) to provide modular implementations of exception handling code; and (ii) defining a pointcut language to support exception detection according to contextual conditions. As part of (i), we are currently investigating the use of AspectJ [28] to support the explicit separation of exception handling code and normal application code [16]. However, the general-purpose pointcut language and inter-type declaration mechanism of AspectJ does not to work well in a wide range of scenarios [16], which are very common in context-aware applications. First, it is not trivial and even possible to modularize application-specific exception detection with the AspectJ mechanisms. Second, softening exceptions may lead to the behavior of the system being unintentionally altered when the handlers are extracted to aspects.

6. Final Remarks

Error handling in mobile agent-based applications needs to be context sensitive. This paper discussed our experience in incorporating exception handling in several prototype MoCA applications. This allowed us to elicit a set of requirements and define a novel context-aware exception handling model, which consists of: (i) explicit support for specifying "exceptional contexts"; (ii) context-sensitive search for exception handlers; (iii) multi-level handling scopes that meet new abstractions (such as groups), and abstractions in the underlying context-aware middleware, such as devices, regions, and proxy servers, (iv) context-aware error propagation, (v) contextual exception handlers, and (vi) proactive exception handling. We have also presented an implementation of this mechanism in the MoCA architecture, and illustrated its use in an AmI agent-based application.

Acknowledgements. Karla and Carlos are supported by the ESSMA Project under grant 552068/02-0. Nelio and Alessandro are supported by the European Commission as part of the grant IST-2-004349 for the AOSD-Europe Project. Alexander is also supported by the European Commission as part of the IST RODIN Project.

7. REFERENCES

[1] A. K. Dey and G. D. Abowd. Towards a better understanding of context and context-awareness. *In Proc. of the 2000 Conference on Human Factors in Computing Systems*, The Hague, The Netherlands, April 2000.

[2] G. Abowd et al. 1999. Towards a Better Understanding of Context and Context-Awareness. In *Proc. of the 1st Intl. Symp. on Handheld and Ubiquitous Computing* (Karlsruhe, September 1999, LNCS 1707. Springer, 304-307.

[3] A. Dey. Understanding and Using Context. *Personal Ubiquitous Comput.* 5, 1 (Jan. 2001), 4-7.

[4] B. N. Schilit, R. Adams and R. Want. Context-aware computing applications. *In Proc. Workshop on Mobile Computing Systems and Applications.* IEEE, December 1994.

[5] O. Davidyuk et al. Context-aware middleware for mobile multimedia applications. In *Proc. of the 3rd international Conference on Mobile and Ubiquitous Multimedia* (College Park, Maryland, October 27 - 29, 2004). vol. 83. ACM Press, New York, NY, 213-220.

[6] A. Iliasov and A. Romanovsky. CAMA: Structured Communication Space and Exception Propagation Mechanism for Mobile Agents. *ECOOP-EHWS 2005*, 19 July 2005, Glasgow.

[7] V. Muthusamy et al. Publisher Mobility in Distributed Publish/Subscribe Systems, *Fourth International Workshop on Distributed Event-Based Systems* (DEBS) (ICDCSW'05), 2005.

[8] G. Cugola and J. E. M. Cote. On Introducing Location Awareness in Publish-Subscribe Middleware. *Fourth International Workshop on Distributed Event-Based Systems* (DEBS) (ICDCSW'05), 2005.

[9] V. Sacramento et al, MoCA: A Middleware for Developing Collaborative Applications for Mobile Users. *IEEE Distributed Systems Online*, vol. 5, no. 10, 2004.

[10] S. Sharples, V. Callaghan and G. Clarke, A Multi-Agent Architecture for Intelligent Building Sensing and Control. *International Sensor Review Journal*, May 1999.

[11] N. Shadbolt, Ambient intelligence. *IEEE Trans. Intell. Transp. Syst.*, vol. 18, no. 4, pp. 2-3, Jul. – Aug. 2003.

[12] F. Souchon et al. Improving exception handling in multi-agent systems. In C. Lucena et al (Eds), *Software Engineering for Multi-Agent Systems II*, number 2940. Feb. 2004.

[13] A.F. Garcia, C. M. F. Rubira, A. Romanovsky, J. Xu. A Comparative Study of Exception Handling Mechanisms for Building Dependable Object Oriented Software: *Journal of Systems and Software. 59(2001), 197-222.*

[14] J. B. Goodenough. Exception handling: issues and a proposed notation. Commun. ACM 18, 12 (Dec. 1975), 683-696.

[15] N. Loughran et al. A domain analysis of key concerns – known and new candidates, Katholieke Universiteit Leuven, Leuven, AOSD-Europe Deliverable D43, AOSD-Europe-KUL-6, 27 February 2006, pp 87-130. Available at www.aosd-europe.net

[16] F. C. Filho, C. M. F. Rubira and A. Garcia. A Quantitative Study on the Aspectization of Exception Handling. In *ECOOP'2005 Workshop on Exception Handling in Object-Oriented Systems*, Glasgow, UK, 2005.

[17] R. Laddad. AspectJ in Actions. Manning, 2003.

[18] A. Tripathi, D. Kulkarni and T. Ahmed. Exception Handling Issues in Context Aware Collaboration Systems for Pervasive Computing. In *Developing Systems that Handle Exceptions*. Proc. ECOOP 2005 Workshop on Exception Handling in Object Oriented Systems. TR 05-050. LIRMM. Montpellier-II University. 2005. July. France.

[19] L. Capra, W. Emmerich and C. Mascolo. CARISMA: Context-Aware Reflective Middleware System for Mobile Applications. *IEEE Transactions on Software Engineering* 29(10): pp. 929--944, Oct 2003.

[20] M. Klein and C. Dellarocas. Exception Handling in Agent Systems. Proc. of the 3rd Int. *Conference on Autonomous Agents*, Seattle, WA, May 1-5, 1999. Pp. 62-6.

[21] T. Anderson and P. A. Lee. *Fault Tolerance: Principles andPractice*. Prentice-Hall, 2nd edition, 1990.

[22] D. Parnas and H. Wurges. Response to Undesired Events in Software Systems. *Proc.2nd ICSE*, S.Francisco, USA, 437-446, 1976.

[23] F. Cristian. A Recovery Mechanism for Modular Software. *Proc. 4th ICSE*, Munich, Germany, pp. 42 – 50, 1979.

[24] M. Lippert and C. Lopes. A Study on Exception Detection and Handling using Aspect-Oriented Programming. *Proc. 22nd ICSE*, Limerick, Ireland, pp. 418 – 427, 2000.

[25] M. Robillard and G. Murphy. Designing Robust Java Programs with Exceptions. *Proc. 8th FSE*, San Diego, USA, 2 – 10, 2000.

[26] M. Robillard and G. Murphy. Static analysis to support the evolution of exception structure in object-oriented systems. *ACM Trans. Softw. Eng. Methodol*., 12(2): 191-221 (2003).

[27] G. Kiczales et al. Aspect-Oriented Programming. *Proc. ECOOP'97*, pp. 220-242.

[28] G. Kiczales et al. An Overview of AspectJ. *Proc. ECOOP'01*, pp. 327-353.