Handling Exceptional Conditions in Mobile Collaborative Applications: An Exploratory Case Study

Nelio Cacho[1] Karla Damasceno[2] Alessandro Garcia[1]
Thais Batista[3] Frederico Lopes[3] Carlos Lucena[2]

[1]*Computing Department — Lancaster University • United Kingdom*
{n.cacho, a.garcia}@lancaster.ac.uk
[2]*Computer Science Department — Pontifical Catholic University of Rio de Janeiro • Brazil*
{karla, lucena}@inf.puc-rio.br
[3]*Computer Science Department, Federal University of Rio Grande do Norte – UFRN• Brazil*
{thais, fred}@ufrnet.br

Abstract

The incorporation of exception handling strategies in mobile collaborative applications bring several challenges to middleware designers due to their intrinsic characteristics of openness, context-awareness, lack of structuring, asynchrony, and increased unpredictability. Publish-subscribe middleware systems are often referred as one of the most common solutions to support the construction of mobile collaborative applications. However, there is no systematic study dedicated to investigate to what extent publish-subscribe mechanisms provide proper support for introducing exception handling strategies into such mobile systems. This paper presents a case study where we have analyzed: (i) the problems emerging from the incorporation of application-specific error handling in a prototype context-aware mobile system from the health care domain, and (ii) the feasibility of a typical publish-subscribe middleware system, called MoCA, to implement the features of an exception handling mechanism tailored to specific requirements of mobile collaborative applications. This paper also discusses the suitability of existing mechanisms recently proposed in the literature to address the identified shortcomings.

1. Introduction

Error handling is a main tenet on the development of resilient distributed systems, which traditionally rely on disciplined system structuring and rigid exception propagation policies [10]. However, treatment of exceptional conditions in mobile collaborative applications offers a number of challenges due their stringent requirements of openness, asynchrony, context-awareness, and increased unpredictability [5, 6]. The dynamically changing collaboration contexts seem to account for a more flexible exception handling approach. Devices working collaboratively may fail to perform specific actions, which may in turn affect other participants in the collaboration. In addition, exception interfaces, search for handlers, and error propagation policies need to adapt according to the frequent variations in the collaboration contexts.

Exception handling is widely recognized as the main scheme to achieve resilience and robustness in modern applications [10]. However, the error handling strategies that were ill designed can make an application even more unreliable. Although there are several middleware systems available nowadays for the development of collaborative mobile applications (e.g. [1, 5]), little attention have been paid for understanding the interplay of error handling and mobile collaborative applications. To the best of our knowledge, there is no systematic study that investigates: (i) the intricacies of collaborative error handling in the presence of physical mobility, and (ii) how mainstream coordination techniques, such as pub-sub mechanisms, are appropriate to implement robust, mobile collaborative applications.

In this context, this paper presents an exploratory case study that identifies some problems emerging from the incorporation of application-specific error handling in a prototype mobile system from the health care domain. In order to implement the collaborative exception handling strategies, we have used a typical pub-sub middleware - MoCA [1]. It supports the development of mobile applications by offering explicit services that empower applications with context-awareness [7]. We also discuss about the adequacy of existing exception mechanisms [2-5] to address the identified shortcomings in collaborative exception handling in mobile environments.

The remaining part of this paper is as follows. Section 2 presents the MoCA architecture and the health care application Section 3 identifies the challenging exception handling issues we have experienced both in the development of the health care mobile application and in the use of MoCA publish-subscribe mechanisms. Section 4 provides discussions and compares our experience with related work. Section 5 presents the concluding remarks of this paper.

2. The Case Study

This section presents both the middleware system (Sec. 2.1) and the mobile collaborative application (Sec. 2.2) we have used in our case study.

2.1 MoCA: Mobile Collaboration Architecture

MoCA [1] is a publish-subscribe middleware [8,9] that supports the development and execution of context-aware collaborative applications. The elements that compose the MoCa application are: a server, a proxy, and clients. While the first two elements execute on a wired network, the clients run on mobile devices. A proxy mediates communication between the application server and its clients. The internal MoCa infrastructure is illustrated in Figure 1.

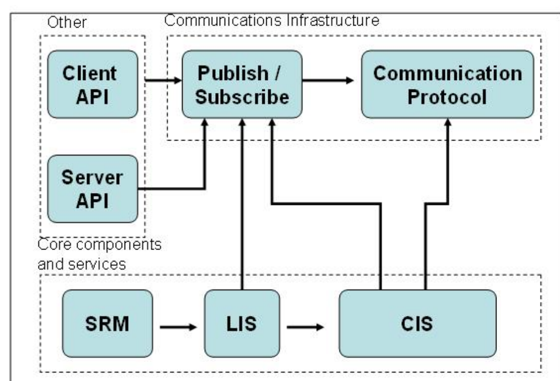

Figure 1. MoCA internal infrastructure

MoCa provides three main services: Context Information Service (CIS), Symbolic Region Manager (SRM) and Location Inference Service (LIS). CIS receives and processes contextual information sent by clients. It also receives notification requests from the application proxies and delivers events to whenever a change in a client's state is of interest to this proxy. SRM offers an interface to define and request information about hierarchies of symbolic regions that are meaningful to location-aware applications. Based on SRM information, LIS infers the location of a mobile device from the raw context information collected by CIS of this device. The communication substrate consists of the pub-sub mechanism and a communication protocol. The former provides the basic functionality to the CIS, once the context recognition is done by the definition of subscriptions that specify a set of user-defined context information.

2.2 A Health Care Mobile Application

In this study, we have implemented a health care mobile (HCM) application using MoCA. It supports the monitoring of mobile home care patients with any type of cardiovascular disease. The cardiac activity of these patients must be continuously monitored. Sensors are used to capture the vital signs of a patient such as heart rate, blood pressure, body temperature, and breathing frequency. Three sensors are attached to the patient. First, a *pulse oximetry* monitors the oxygen saturation of hemoglobin in arterial blood and pulse rate. The oxygen saturation should always be above 95%. The normal pulse rate for healthy adults ranges from 60 to 100 beats per minute (BPM). Second, a *blood-pressure monitor* measures the patient's blood pressure. Normal blood diastolic pressure ranges from 100 to 120 mmHg and systolic pressure ranges from 60 to 80 mmHg. Third, a *smart shirt* monitors the EKG, respiration, and temperature of the patient. Figure 2 illustrates the code used for publishing an event with some cardiovascular information of the patient received from the sensors.

```
ECIClient client= new ECIClient(
         ECIConstants.TCP,
         eciServerlAddress, eciLocalAddress);
EventProperties eventp = new
                         EventProperties();
eventp.setIntProperty("BPM", 30);
eventp.
setIntProperty("SystolicBloodPressure",210);
eventp.
setIntProperty("DiastolicBloodPressure",120);
client.publish("HeartInformation",
                eventp, null);
```

Figure 2: Publishing an Event

This application encompasses a number of both predicted and opportunistic mobile collaboration scenarios. They may include elements in different contexts, such as in the victim region, and specialized entities, such as ambulances and the Emergency support group. Several exceptional conditions may also occur in this application. For instance, the monitoring activity of sensors can diagnose an abnormal or dangerous heart rhythm that represents a risk for the patient. This event requires an exceptional control flow that immediately notifies the components in charge of handling that problem. Other examples of abnormal situations include a pulse rate out of range or low oxygen saturation, which are detected by one of the sensors. An event is triggered to immediately alert a relative, the family doctor, or the hospital, depending on the degree of alert detected. They are all considered exceptions because they both represent some environmental, software, or

hardware fault, and require immediate, synchronous event delivery and handling.

Figure 3 contains the code used by the users interested in the exceptional conditions mentioned in the previous paragraph. The code implements a subscription to an exceptional condition signaling a heart attack. All involved elements of the health care system (relatives, doctor, and ambulances) use a similar code to subscribe their interest in this exception. The occurrence of a heart attack event may then trigger the engagement of these elements in different "exceptional collaboration" scenarios. For example, the doctor entering in the hospital will receive the exceptional notification and may decide to join in patient handling. This is a typical situation where such usual subscriptions properly work to implement an exception notification. Section 3 discusses the scenarios where the pub-sub mechanism supported by MoCA has presented some shortcomings.

```
ECIClient device = new ECIClient();
EventListener listener = new EventListener();
Topic topic =
device.subscribe("HeartInformation",
 "(BPM<40 and SystolicBloodPressure > 180 and
DiastolicBloodPressure > 100");
device.addListener(listener, topic);
...
```

Figure 3: Subscribing an Event

3. Exception Handling in Mobile Collaboration

This section presents a number of difficulties that we have faced while incorporating exception handling in the HCM application and using MoCA. Each of the subsections below discusses: (i) specific problems we have identified for implementing proper error handling strategies in our case study, (ii) the illustration of those problems using concrete HCM mobile collaboration scenarios involved in the identification of (i), (iii) our MoCA-based implementation to address the identified shortcoming, and (iv) the limitations of the pub-sub mechanism of MoCA in the implementation of (iii).

3.1 Context-Aware Exception Handling

As in a previous case study [6], the implementation of the HCM application required the need for specifying "exceptional contexts". Exceptional contexts mean one or more conditions associated with the context types, which together denote an environmental, hardware, or software fault. For example in Figure 4, the exceptional contexts can be characterized by the situations when the cardiac variables (BPM and Blood Pressure) collected by the body sensors occasionally exceeds the maximum and minimum limit according to the medical definition. They can indicate a serious problem in the cardiovascular system of the human body. Exceptional context-related subscription also differ from normal event subscription in the sense it requires an exceptional control flow and involve, for example, several entities in the handling process. As illustrated in figure 4, it may require references to a reflective exception mechanism, such as, access to the reflective variable *ThisException.Handled.scopes*, which means a list of scopes that has already handled the exception

Context-Sensitive Exceptional Flow. The handling of such exceptional contexts in the HCM application required an exceptional control flow different from the normal ones, which typically consist of regular notification-based reactions. The seriousness of this exceptional context requires propagation of such exceptional context information to the proper urgent attendance, which also may vary depending on their physical location. It may also require involvement and collaboration of several entities, such as relatives and Emergence Support group. However, the MoCA pub-sub mechanism did not provide direct, effective support for implementing such a context-sensitive exceptional control flow. The pub-sub mechanism assumes that all the involved parties in a cooperative exception handling explicitly registers interest in the exceptional event.

Scoping in Collaborative Exception Handling. In order to support such "exceptional collaboration", we have basically relied on three main MoCA abstractions to implement exception handling scoping: a device, a proxy server, and a region. However, these scoping abstractions were not enough; some exceptional contexts required the handling involving a group of devices not associated with a region. We needed a more flexible application-specific mechanism to support collaborative exception handling. Hence we also had to implement the notion of group (of devices) as an additional kind of scope, which was not supported by MoCA. Each of these scopes provided a collaborative space within which eligible entities in the HCM application could interact to deal with the exceptional context occurrences. Typical examples of region and group scopes included respectively victim region and the Emergency Support team.

Multi-level Scopes. A group scope encompasses a set of devices that is defined by the application. The goal is to support mobile cooperative handling of an exception amongst the device's agents pertaining to that group. Differently from region scopes, this kind of scope is not directly related to spatial relationship. It makes it possible to insert or remove elements from the scope according to the application necessity. A region scope differs from the other scope types in the sense it has a more dynamic characteristic in terms of the participating devices. In our implementation, this scope is strongly related to the MoCA LIS service (Section

2.1) as it provides a reference mechanism that allows for a device being aware of its neighbours.

Volatile Exception Interface. In the HMC application, devices can enter and leave a region scope (e.g. the hospital). A device movement also characterizes a change in the exception handling scope. 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. The mobile devices encountered in the new region can also influence the list of possible exceptions being raised in collaborations. It implies that exception interfaces in mobile collaborative applications may vary according to new contexts and collaboration opportunities. Such volatile exception interfaces motivates the need for some source of proactive exception handling (Sect 3.2). In addition, it is a kind of requirement not commonly detected in traditional distributed systems, where the exception interface associated with each collaborative component is well-known in advance. MoCA did not provide any support for implementing the management of exception interfaces. This feature seems also not be straightforwardly implemented in pub-sub mechanisms.

Error Propagation. The multi-level scope of error handling allows for errors being propagated through the channel of nested scopes. The implementation of error propagation in HCM required a service to define the sequence of exception propagation. The user should use the *propagateTo* method as illustrated in Figure 4.

```
HeartAttack hattack = new HeartAttack(
 "(BPM<40 and SystolicBloodPressure > 180
       and DiastolicBloodPressure > 100)");

hattack.propagateTo(
  RegionScope.getInstance(),1, "!("+
  RegionScope.getInstance() +
 "exist_in ThisException. Handled.scopes)");
hattack.propagateTo(GroupScope.getInstance("Eme
rgencySupport"),1, "!("+GroupScope.
 getInstance("EmergencySupport")+ "exist_in
           ThisException. Handled.scopes)");
hattack.propagateTo(DeviceScope.getInstance("My
Doctor"),2);
```

Figure 4: Context-Aware EH definition

This method receives as parameter the scope reference, the sequence number, and also the condition. The scope reference determines the scope to which the exceptional context will be propagated, the sequence number, and the order. Scope condition defines the propagation policy in which the error propagation will occur; it specifies when the exception needs to be propagated. For instance, in Figure 4, the *HeartAttack* will firstly propagate to the victim region and the Emergency Support group. If none of them handles this exception, it is delivered to the doctor device. This collaborative activity is very important to the exception handling domain once it supports the correct exception propagation instead of the exception treatment. MoCa does not provide this kind of information, however we included it in our implementation.

Context-Sensitive Exception Catching. To deal with the exceptional context of Figure 4, we needed to define a context-sensitive handler that allows the definition of context-related conditions before it is executed.

```
public class FirstHelp extends Handler {
public boolean
 verifyContextCondition(Exception ex,
                              Context context){
    if (context.check("BPM < 20"))
        return false;
    else   return true;}
 public boolean execute(){
     inform_sequence_of_first_help();}}
FirstHelp fhelp = new FirstHelp(hattack);
RegionScope regionScope =
               RegionScope.getInstance();
regionScope.attachHandler(fhelp);
```

Figure 5: Context-sensitive handlers

For instance, Fig. 5 describes the *FirstHelp* handler definition and also its association with the region scope. The definition of a handler requires the extension of the *Handler* abstract class. This class requires the implementation of *verifyContextCondition* and *executes* abstract methods defined in the API of our mechanism. The first method verifies if the current execution context is really appropriate for the execution of such a handler, and the second one executes the handler functionality. In HCM, this handler is associated with the patient region (in this case its house), once every people who live together with he/she can help in an exceptional situation. Thus, when an exception is caught, the mechanism executes the *verifyContextCondition* for each handler defined in that scope. If 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. In MoCA, an exceptional event can be caught by a listener definition; however this approach uses just the event data to decide what to do with this event. In our implementation, as shown in Fig. 5, the handler can check the current situation of the application context to decide if it handles or not a specific exception. This allows the definition of a context-sensitive handler to deal with context exceptional conditions.

3.2 Proactive Exception Handling

In an open mobile system, like the HCM system, we could not wait that all the devices, in which software was developed by different designers, would be able to foresee all the exceptional contexts. In the health care

case, for example, during the occurrence of one heart attack, if there is nobody at home besides the victim, one of the neighbors could receive this exception notification; however, it may not be able to treat it. As a result, there was a need for exploiting the MoCA-enabled mobile collaboration infrastructure when an exceptional context is detected by one of the peers.

Proactive Exception Notification. Severe exceptions should be notified to other mobile devices even when they have not registered interest in that specific exceptional context. In other words, the contextual exception should be proactively raised in other mobile collaborative agents and/or mobile devices pertaining to the same region. Thus robust context-aware mobile systems require more intelligent, proactive exception handling due to their features of openness, asynchrony, and increased unpredictability. Again, this requirement does not suit well the pub-sub paradigm as it implies on the explicit registration of each involved element.

Proactive Handling and Propagation. In order to handle the unforeseen exception, the receiver should start a collaborative activity to search for an adequate handler for this received exception. In this situation, the receiver is going to collaborate with the victim device to deal with exception and, for instance, perform the first urgent help while the ambulance does not arrive. Figure 6 depicts a proactive exception definition.

```
FirstHelp handler = new FirstHelp();
PulseOximetryError oximetre = new
                      PulseOximetryError ();
oximetre.propagateTo(
  RegionScope.getInstance(),1, "!("+
  RegionScope.getInstance() +
 "exist_in ThisException. Handled.scopes)" );
oximetre.propagateTo(GroupScope.getInstance("Ma
intainerSupport"),1,null);
oximetre.propagateAsProactive(
 RegionScope.getInstance("neighborhood"),
                          2,null);
oximetre.throw();
```

Figure 6: Proactive exception definition

The main difference from Figure 3 is the *propagateAsProactive and throw* methods. The first one propagates (using *throw),* the *PulseOximetryError* exception to all devices available in the neighborhood region whether nobody has dealt with this exception before. This propagation allows an expansion into a new collaboration group and also supports a dynamic mechanism to discovery new handlers. For instance, the next destination of the exception - the neighborhood group - does not know how to handle it. In order to deal with it, the neighborhood group starts a collaborative activity to discovery and to execute possible handlers.

3.3 Concurrent Exception Resolution

In our case study, we have also noticed that several concurrently thrown exceptions can occur, which actually mean the occurrence of a more serious abnormal situation. A common example is the simultaneous detection of a heart attack exception (Fig. 4) and an attendance inability by the ambulance system. Thus, a central control system should be able to collect all those concurrent exception occurrences and resolve them so that the proper action can be triggered. Note that it is no longer satisfactory the activation of several handlers for each individual exceptional condition. MoCA did not provide any support for such concurrent resolution of events. We defined a concurrent exception resolution that infers a single exception from the multiple abnormal concurrent situations.

In order to illustrate the concurrent exception definition (Figure 7), we use the aforementioned situation where two exceptions *HeartAttack* and *AttendanceInability* are caught and resolved as the composite exception *GlobalException* that represents the simultaneous occurrence of both exceptions. To define such concurrent exceptions, our approach supports the following operators: *simultaneous* - || (two exceptions occurring exactly at the same time); *simultaneous with a time frame* - |(X)| (where, for example, X = 1 means one minute time frame); *follow* to define a sequential occurrence within a time frame – FOLLOWED(X) (where, X = 5 means the occurrence of the second event until 5 minutes after the first event); *interaction* occurrences like Oximetry FOLLOWED (INTERACTION(CardioSound,3)).

```
GlobalException glbexe = new
GlobalException("Exception('HeartAttack')|(10)|
Exception('AttendanceInability')");
glbexe.propagateTo(
  ServerScope.getInstance("MainServer"),
                             1,null);
```

Figure 7: Concurrent exception definition

4. Discussion and Related Work

As we have discussed in Sect. 2, the specification of contextual conditions of interest in pub-sub systems, such as MoCA, require explicit subscriptions based on regular expressions. The subscription is usually carried out by the code in the devices or proxy servers, which will receive notifications when those contextual conditions are matched. However, Sect. 3.1 illustrates that 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, types of contextual information which *should* and *should not* propagated together with the exception occurrence, and so on.

Most of existing solutions in terms of conventional pub-sub mechanisms are too general and do not offer a flexible support for handling a broad range of exceptional situations considering application-specific

context information. The first problem is that in pub-sub systems exceptions are continuously propagated to the users interested in such type of event. Second, the propagation mechanism sends the event to all elements that registered subscriptions to a specific type of event. As discussed in the exception of our case study, it must be propagated initially to the relatives of the patient. Thus, exception propagation must be selective and taking into account the different types of users and the context information such as the exception severity. This problem is also related to the fact that the *subscription* process does not support the definition of users' levels. In our example at least two different levels of users need to be identified: the relatives and the maintenance staff. Finally, in general, pub-sub middleware does not provide facilities for handling composition of exceptional events. In the HCM application two exceptions can be raised from a given patient and these two exceptions must be combined in order to provide meaningful information for the handlers.

A scheme in [2] supports exception handling in systems consisting of agents that cooperate by sending asynchronous messages. It allows handlers to be associated with services, agents and roles, and supports concurrent exception resolution. Some exception handling issues in a context-aware application is discussed in [3]. The paper discusses issues that arise due to security and coordination particularities. Klein et al. [4] defines 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 [5] supports (nested) scopes, which confine the errors and to which exception handlers are attached. However, CAMA and the other previously mentioned mechanisms do not support a fully context-aware, collaborative exception handling, as our approach does. They do not address context-aware selection of handlers, proactive exception handling, concurrent resolution and the definition of exceptional contexts.

In a previous case study [6], we have analyzed the impact of context-awareness in the implementation of exception handling mechanisms for a different domain. However, we have not focused on the interplay of exceptions and mobile collaborative applications.

5. Final Remarks

This paper has presented an exploratory case study that integrates error handling in a mobile collaborative application from the health care domain. We have analyzed the use of a pub-sub middleware, its advantages, and shortcomings to deal with exceptional conditions in mobile collaborations. We have also presented how we have implemented the collaborative error handling strategies in the HCM application with the MoCA architecture. The analysis allowed us to obtain a set of central requirements on the definition of an exception handling mechanism for mobile collaborative systems, 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) volatile exception interface and contextual exception handlers, (vi) proactive exception handling, and (vii) concurrent resolution of exceptional conditions.

6. References

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

[2] 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.

[3] A. Tripathi, et al. "Exception Handling Issues in Context Aware Collaboration Systems for Pervasive Computing". In Romanovsky et al (Eds.) Proc. ECOOP 2005 Workshop on Exception Handling in Object Oriented Systems. 2005.France.

[4] 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

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

[6] K. Damasceno et al. "Context-Aware Exception Handling in Mobile Agent Systems: The MoCA Case". *Proceedings of the 5rd International Workshop on Software Engineering for Large-Scale Multi-Agent Systems (SELMAS 2006)* at ICSE 2006, Shanghai, China, May 2006.

[7] A. Dey and G. 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.

[8] G. Cugola, et al. The JEDI event-based infrastructure and its application to the development of the opss wfms." IEEE Transactions on Software Engineering, 2001.

[9] P. Eugster et al. "The Many Faces of Publish/Subscribe". ACM Computing Surveys, 35(2):114{131, June 2003.

[10] A. Garcia, C. 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.