# On Using the CAMA Framework for Developing Open Mobile Fault Tolerant Agent Systems

Budi Arief, Alexei Iliasov and Alexander Romanovsky
School of Computing Science
University of Newcastle upon Tyne
Newcastle upon Tyne NE1 7RU, England
{L.B.Arief, Alexei.Iliasov, Alexander.Romanovsky}@newcastle.ac.uk

## ABSTRACT

The paper introduces the CAMA (*C*ontext-*A*ware *M*obile *A*gents) framework intended for developing large-scale mobile applications using the agent paradigm. CAMA provides a powerful set of abstractions, a supporting middleware and an adaptation layer allowing developers to address the main characteristics of the mobile applications: openness, asynchronous and anonymous communication, fault tolerance, device mobility. It ensures recursive system structuring using location, scope, agent and role abstractions. CAMA supports system fault tolerance through exception handling and structured agent coordination. The applicability of the framework is demonstrated using an ambient lecture scenario – the first part of an ongoing work on a series of ambient campus applications.

## 1. INTRODUCTION

Although the mobile agent paradigm supports structuring systems using decentralised and distributed entities cooperating to achieve their individual aims and promotes system openness, flexibility and scalability, the existing frameworks for development of such systems do not provide adequate means for achieving fault tolerance. The main difficulties here are caused by agent mobility, autonomy and asynchronous communication, system openness and dynamicity, which create new challenges for ensuring system fault tolerance.

In this work, we are focusing on *coordination mobile environments*, which have become very popular in developing mobile agent applications. These environments rely on the Linda approach to coordination of distributed processes. Linda [6] provides a set of language-independent coordination primitives that can be used for communication-between and coordination-of several independent pieces of software. Linda is now becoming the core component of many mobile software systems because it fits in nicely with the main characteristics of mobile systems. Linda coordination primitives support effective inter-process coordination by allowing processes to put *tuples* in a tuple space shared by these processes, get *tuples* out if they match the requested types, and test for them. A tuple is a vector of typed data values, some of which can be empty, in which case they match any value of a given type. Certain operations, like *get* (or `in`) and *test* (or `inp`), can be blocking.

One of the most developed and widely-used examples of Linda-based mobile coordination environments is Lime [12]. It supports both *physical mobility*, such as a device with a running application travelling along with its user across network boundaries, and *logical mobility*, when a software application changes its platform and resumes execution in a new one. To do that, Lime employs a distributed tuple space. Each agent has its own persistent tuple space that physically or logically moves with it. When an agent is in a location where there are other agents or where there is a network connectivity to other Lime hosts, a new shared tuple space can be created, thus allowing agents to communicate. If connection is lost or some agents leave, parts of the shared tuple space became inaccessible. Lime middleware – implemented in Java – hides all the details and complexities of the distributed tuple space control and allows agents to treat it as normal tuple space using conventional Linda operations.

Exception handling [3] is widely accepted to be the most general approach to ensuring fault tolerance of complex applications facing a broad range of faults. It provides a sophisticated set of features for developing effective fault tolerance using handlers specially tailored for the specific exception and system state in which the error is detected. It ensures nested system structuring and separates normal system behaviour from the abnormal one. Our analysis [10] shows that the existing Linda-based mobile environments do not provide sufficient support for development of fault tolerant mobile agent systems. The real challenge here is to develop general mechanisms that smoothly combine Linda-based mobility with exception handling. The two key features of mobile agents are asynchronous communication and agent anonymity. This is what makes mobile agents such a flexible and powerful software development paradigm. However, traditional fault tolerance and exception handling schemes are not directly applicable in such environments.

In this paper, we discuss a novel framework for disciplined development of open fault tolerant mobile agent systems and show how it is being applied in developing an ambient campus application. This framework offers a set of powerful abstractions to help developers by supporting exception handling, system structuring and openness. These abstractions are supported by an effective and easy-to-use middleware

which ensures high system scalability and agent compatibility. The plan of the paper is as follows. In the next section we introduce our CAMA framework in detail by describing the main abstractions offered to system developers, a novel exception handling mechanism and our current work on CAMA implementation. This is followed by a section discussing our experience in applying CAMA in the development an ambient lecture scenario as a part of our ongoing work on ambient campus applications. The last section of the paper outlines our plans for the future work.

## 2. CONTEXT-AWARE MOBILE AGENTS

We have developed a framework called CAMA *(Context-Aware Mobile Agents)*, which encourages disciplined development of open fault tolerant mobile agent applications by supporting a set of abstractions ensuring exception handling, system structuring and openness. These abstractions are backed by an effective and easy-to-use middleware allowing high system scalability and guaranteeing agent compatibility.

### 2.1 CAMA Abstractions

Any CAMA system consists of a set of *locations*. A location is a container for *scopes*. A scope provides a coordination space within which compatible agents can interact using the scoping mechanism described below. *Agents* are the active entities of the system. An agent is a piece of software that conforms to some formal *specification*. Each agent is executed on a *platform*; several agents may reside on a single platform. A platform provides an execution environment for agents as well as an interface to the location middleware. Figure 1 shows how these abstractions are linked.

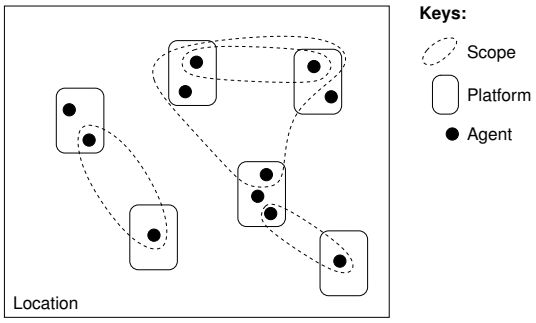


**Figure 1: Location, scopes, platforms and agents in Cama**

An agent is built using one or more *roles*. A role is a specification of one specific functionality of an agent. A composition of all agent roles forms its specification.

Location can be associated with a particular physical location (such as lecture theatre, warehouse or meeting room) and can have certain restrictions on the types of supported scopes. Location is the core part of the system as it provides means of communication and coordination among agents. We assume that each location has a unique name. This roughly corresponds to the IP address of the host in a network (which are usually unique) on which it resides. A location must keep track of the agents present and their properties in order to be able to automatically create new scopes and restrict access to the existing ones. Locations may provide additional services that can vary from one instance to another. These are made available to agents within what appears to be a normal scope where some of the roles are implemented by the location system software. As with all the scopes, agents are required to implement specific roles in order to connect to a location-provided scope. Few examples of such services include printing on a local printer, accessing the internet, making a backup to a location storage, and migrating to another location.

Agent *context* represents the circumstances in which an agent find itself [14]. Generally speaking, a context includes all information from an agent environment which is relevant to its activity. The context of an agent in CAMA consists of the following parts: the state connections to the engaged locations; the names, types and states of all the visible scopes in the engaged locations; and the state of scopes in which the agent is currently participating, including the tuples contained in these scopes. A set of all locations defines global structuring of the agent context. This context changes when an agent migrates from one location to another.

Agents represent the basic structuring unit in CAMA applications. To deal with various functionalities that any individual agent provides, CAMA introduces agent role as a finer unit of code structuring. A role is a structuring unit of an agent, and being an important part of the scoping mechanism, it allows dynamic composition of multi-agent applications, as well as being used to ensure agent interoperability and isolation.

Scope structures the activity of several agents in a specific location by dynamically encapsulating roles of these agents. Scope also provides an isolation of several communicating agents thus structuring the communication space.

A set of agents playing different roles can dynamically instantiate a multi-agent application. A simple example is a client-server model where a distributed application is constructed when agents playing two roles meet and collaborate. An agent can have several roles and use them in different scopes. A server agent can provide the same service in many similar scopes. In addition it can also implement a client role and act as a client in some other scopes.

Supporting system openness is one of the top design objectives of CAMA. Openness is understood here as the ability to create distributed applications composed of agents developed independently. To this end CAMA provide powerful abstractions that help to dynamically compose applications from individual agents, an agent isolation mechanism and service discovery based on the scoping mechanism.

#### *Scoping mechanism*

The CAMA agents can cooperate only when they participating in the same scopes. This abstraction is supported by a special construct of coordination space called *scope*. Scoping is a means to *structure* agent activity by arranging agents into groups according to their intentions. Scoping also allows agent communication to be configured to meet to the requirements of the individual groups. Reconfigurations happen automatically, thus allowing agents (and their developers) to focus solely on collaboration with other agents participating in the same scope. There are several benefits of agent system structuring using scopes:

- scopes provide higher-level abstractions of communication structuring;

- they reduce the risk of creating ad hoc structures that maybe incorrect, malfunctioning or cyclic;
- this structuring enforces strong relationship among agents supporting interoperability and exception handling;
- scopes support simple semantics thus facilitating formal development;
- scopes become units of fault tolerant system ensuring error confinement and supporting error recovery at the scope level.

A scope is a dynamic data container that provides an *isolated* coordination space for *compatible* agents. This is done by restricting visibility of tuples contained in the scope only to these agents. we say that a set of agents is compatible if there is a composition of their roles that forms an instance of an abstract scope model.

Agents can issue a request to create a scope, and when all the preconditions are satisfied, a scope is atomically instantiated by the hosting location. The scope creation request includes a scope identifier (a string) and a scope requirement structure. The request returns the name of a newly created scope. The agent creating the scope can use it to join the scope, to make it public (visible to other agents), to leave it and to remove it.

Scope has a number of attributes divided into two categories: scope *requirements* and scope *state*. Scope requirements essentially define the type of a scope, or, in other words, the kind of activities supported by it. Scope requirements are derived from a formal model of a scope activity and, together with agent roles, form an instance of the abstract scope model. State attributes characterise a unique scope instance. In addition to these attributes, scope contains *data* represented as *tuples* in the coordination space. Along with these data, there may be *subscopes* which define *nested activities* that may happen inside of the scope.

Nested scopes are used to structure large multi-agent applications into smaller parts which do not require participation of all agents. Such structuring has a number of benefits. It isolates agents into groups, thus enhancing security. It also links coordination space structuring with activity structuring, which supports localised error recovery and scalability. There is no hard rule when to use nested scopes. However, for reasons stated above, any application incorporating different modes of communication or different types of activities should use subscopes. Online shop is an example of such application. A seller publicly communicate with buyers while the latter are looking around for some products. However, payment must be a private activity involving only the seller and the buyer. In addition to obvious security benefits, a dedicated payment subscope helps to determine which agents must be involved into recovery should a failure happen during payment.

Restrictions on roles dictate the roles that are available in the scope, and how many agents are allowed for any given role. The latter is defined by two numbers: the minimum number of agents required for a given role and the maximum number of agents allowed for a given role. A *scope-state* tracks the number of currently-taken roles and determines whether the scope is ready for agent collaboration or whether more agents are allowed to join.

The existing scoping mechanisms (e.g. [15, 11]) are not explicitly developed to support data and behaviour encapsulation or isolation crucial for error confining and recovery. None of them is directly applicable for dealing with mobile agents interacting using coordination spaces (see our analysis in [10]). Also, these schemes do not support the set of abstractions which we have identified as crucial for CAMA.

### *Basic Operations in* CAMA

In CAMA, all the communication within a location happens through a single shared tuple space. This leads to asymmetrical design of the middleware where the tuple space operations are implemented in a *location middleware* while agents only carry a lightweight *adaptation layer*. On top of the coordination primitives derived from Linda, the CAMA middleware provides the following operations:

- `engage(id)` - issues a new location-wide name that is unique and unforgeable for agent `id`. This name is used as agent identifier in all other role operations.
- `disengage(a)` - makes issued name `a` invalid.
- `create(a, n, R)@s` (`n` $\notin$ `l.s`) - agent `a` creates a new subscope within scope `s` called `n` with given scope requirements `R` at location `l`. The created scope becomes a private scope of agent `a`.
- `delete(a, n)@l.s` (`n` $\in$ `l.s` $\wedge$ `a` *is owner of* `l.s.n`) - agent `a` deletes a subscope called `n` contained in scope `s`. This operation always succeeds if the requesting agent is the owner of the scope. If the scope is not in the pending state then all the scope participants shall receive `CamaExceptionNotInScope` exception notifying the scope's closure. This procedure is executed recursively for all the subscopes contained in the scope.
- `join(a, n, r)@s` (`n` $\in$ `l.s`$\wedge$`r` $\in$ `n`$\wedge$`n` *is pending or expanding*) - adds agent `a` into scope `n` contained in `l.s` with role `r`. This operation succeeds if scope `l.s.n` exists and agent `a` is allowed to take the specified role in the scope. This operation may cause the scope to change state.
- `leave(a, n, r)@s` (`a` *is in* `l.s.n` *with role(s)* `r`) - removes agent `a` with roles `r` from scope `l.s.n`. The calling agent must be already participating in the scope. This operation may also change the state of the scope.
- `put(a, n)@s` - agent `a` advertises scope `n` contained in scope `s`, thus making it a public scope. A public scope is visible and accessible by other agents.
- `get(a, r)@s`: enquires the names of the scopes contained in scope `l.s` and supporting role(s) `r`.

An agent always starts its execution by looking for available locations nearby. Once it engages a location it can join a scope or create a new one. An agent needs to know the name of the scope it intends to join. It can be the name of an existing scope or the name of a new scope created by this agent. When joining a scope, an agent specifies its role in the scope. In the current implementation of the middleware, an agent can choose a role in a scope from one of the roles it implements. The `join` operation returns a handle for a scope, which can be used by an agent to collaborate with other agents through Linda coordination primitives. To create a scope, an agent must specify the name of the scope and the scope requirements, which define the possible roles within the scope and their restrictions.

### *Physical and Logical Mobility*

Physical mobility allows devices carrying the agent code to move between locations. Logical mobility allows agent code and state to be moved from one location to another.

Physical mobility in CAMA is implemented using connectivity of the devices to the locations. When such a connectivity is established, the agent running on the device receives special event notifying it about discovery of the new location. CAMA allows any agent to access the list of active locations it is connected to at any time. An agent receives a predefined disconnection exception when the connectivity is lost. To support this functionality, the location middleware periodically sends hard beats messages in the proximity.

The CAMA middleware does not support logical mobility as the first class concept since the CAMA architecture does not allow locations to see each other. Nevertheless, agent migration can be provided through the standard inter-agent communication. Data can be moved between locations in CAMA by agents working at both locations at the same time, or by an agent physically migrating between two locations or by using some other capability supporting data transfer between locations. In particular, we have implemented a simple proof-of-concept support ensuring weak code mobility. In this implementation, a dedicated agent provides a service of data transfer between locations using internet or LAN networking. Using this service, any agent can transfer itself or another agent to another location.

## 2.2 Fault Tolerance

The CAMA framework supports application-level fault tolerance by providing a set of abstractions and a supporting middleware that allow developers to design effective error detection and recovery mechanisms. The main means for implementing fault tolerance in CAMA is a novel exception handling mechanism which associates scopes with the exception contexts. Scope nesting provides recursive system structuring and error confinement. In addition to this, the CAMA middleware supports a number of predefined exceptions (e.g. the connection and disconnection ones, violation of the scope constraints, etc.).

In developing exception handling support for CAMA, we relied on our previous work reported in [10], in which we proposed and evaluated a novel exception handling scheme developed for coordination-based agent applications. The main novelty of the CAMA mechanism is that it explicitly links nested scopes with the exception contexts.

Exception handling in CAMA allows fast and effective application recovery by supporting flexible choice of the handling scope and of the exception propagation policy. The mechanism of the exception propagation is complimentary to the application-level exception handling. All the recovery actions are implemented by application-specific handlers attached to agents. The ultimate task of the propagation mechanism is to transfer exceptions between agents in a reliable and secure way. However, the freedom of agent behaviour in agent-based systems does not allow any guarantees of reliable exception propagation to be given in a general case. In particular, the situations can be clearly identified when exceptions may be lost or not delivered within a predictable time period. This is the case for CAMA as well. To alleviate this, for example, in a mobile agent application requiring cooperative exception handling involving several agents, agents behaviour must be constrained in some way to prevent any unexpected migrations or disconnections. In our ongoing work we are developing techniques supporting formal analysis of exception handling behaviour of the multi agent systems.

There are three basic operations available to the CAMA agents for catching and raising inter-agent exceptions. These functionalities are complementary and orthogonal to the application-level mechanism used for programming internal agent behaviour.

The `raise` operation propagates an exception to an agent or a scope. There are two variants of this operation:

- `raise(m, e)` - raises exception `e` as a reaction to message `m`. The message is used to trace the producer and to deliver an exception to it. The operation fails if the destination agent has already left the scope in which the message was produced.
- `raise(s, e)` - raises exception `e` in all participants of scope `s`.

The crucial requirement for the propagation mechanism is to preserve all the essential properties of agent systems such as anonymity, dynamicity and openness. The exception propagation mechanism does not violate the concept of anonymity since we prevent disclosure of agent names at any stage of the propagation process. Note that the `raise` operation does not deal with names or addresses of agents. Moreover, we guarantee that our propagation method cannot be used to learn the names of other agents.

Two other operations, `check` and `wait` are used to explicitly poll and wait for inter-agent exceptions.

- `check` - raises exception `E(e)` if there are any pending exceptions for the calling agent. `E(e)` is a local envelop for the inter-agent exception $e$.
- `wait` - waits until any inter-agent exception appears for the agent and raises it in the same way as the previous operation.

Systematic use of exception handling should allow developers to design mobile agent applications that can tolerate a broad range of faults, including: disconnections, agent mismatches, malicious or unanticipated agent activity, violations of system properties, potentially harmful changes in the system environment, and reduced amount of resource available.

Unfortunately, there has not been much work carried out in this area. Paper [16] introduces a guardian model in which each agent has a dedicated guardian responsible for handling all agent exception. This model is general enough to be applied in many types of mobile systems but it does not directly address the specific characteristics of the coordination paradigm. Another relevant work is on exception handling in concurrent object-oriented language Oz [17]. In this system, exceptions can be propagated between the mobile callee and caller objects. The approach proposed is not applicable to the coordination- based mobile systems. Moreover, the main intention behind this work is not to support the development of open dynamic agent applications.

## 2.3 CAMA Implementation

In the current version of the CAMA system, the location middleware is implemented in C (we call it $_c$CAMA). This allows us to achieve the best possible performance of the coordination space and to effectively implement numerous extension, such as the scoping mechanism. The location middleware implementation is quite compact - it consists of approximately 6000 lines of C code and should run on most Unix platforms. We have so far tested it on Linux FC2 and Solaris 10. The full implementation of the location middleware is available at SourceForge [8].

In order to use the location middleware mentioned above, we have developed a CAMA adaptation layer in Java[1] called $_j$CAMA. This adaptation layer defines several classes for representing – among others – the abstract notions of Location, Scope and Linda coordination primitives. $_j$CAMA provides an interface through which mobile agents or applications can be developed easily.

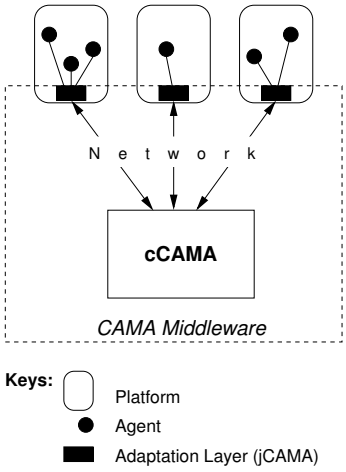


**Figure 2: Cama architecture**

A diagrammatic representation of the CAMA-based system architecture can be seen in Figure 2. Each platform carries a copy of $_j$CAMA. Agents residing on a platform uses the features provided by $_j$CAMA to connect over the wireless network to the $_c$CAMA location middleware.

It is possible to construct other adaptation layers for different platforms and languages. For now, the $_j$CAMA Java adaptation layer outlined above permits agent development for PocketPC-based PDAs. It has a very small footprint (˜60Kb) and can be used with both standard Java and J2ME. In the future we plan to develop adaptation layers for other languages such as Python and Visual Basic, as well as versions compatible for smartphone devices.

# 3. AMBIENT LECTURE APPLICATION

This case study provides a demonstration on how the CAMA framework can be used in developing open, dynamic and pervasive systems involving people carrying hand held devices (e.g. PDAs) to help them in their daily activities.

## 3.1 Introduction

We focus on the activities performed by students and teachers during a lecture (the *ambient lecture* scenario) and consider a set of requirements that define this scenario. This set will be extended to cover more general *ambient campus* scenarios (i.e. location-aware activities that can be performed on campus) such as interactive/smart map, events announcer, library application and students organiser.

There are several other projects aiming to integrate software systems – including mobile applications – into education or campus domain. The ActiveCampus project [7] aims to provide location-based services such as *Map* service (showing outdoor and indoor map of the user's vicinity along with activities happening there) and *Buddies* service (showing colleagues and their locations, as well as sending messages to them). The ActiveCampus system is implemented as a web server using PHP and MySQL. ActiveClass [13] is a client-server application for encouraging in-class participation using PDAs allowing students to ask questions regarding the lecture in anonymous manner, hence overcoming the problem of shyness among many students.

Gay et. al. carried out an experiment investigating the effects of wireless computing in classroom environment [5]. Students were given laptop computers with wireless or wired connection to the internet, allowing them to use any existing tools and services such as web browsers, word processors, instant messaging software – as well as any additional software they wish to install. The results suggest that the introduction of wireless computing in learning environments can potentially affect the development, maintenance and transformation of learning communities, but not every teaching activity or learning community can or should successfully integrate mobile computing applications.

Classtalk [4] is a classroom communication system that allows teacher to present questions for small group work, collect the answers and display the histograms showing how the class answered those questions. Up to four students can be in one group, sharing one input device (a palmtop), which is wired to the central computer controlled by the teacher.

Similar to Classtalk, our system allows students to be grouped together in order to carry out some task given by the teacher. The novelty of our approach lies in the communication channel (wireless instead of wired connection) as well as in using the framework for supporting scoping and fault tolerance (the mechanisms described in Section 2).

## 3.2 Traceable Requirements

We started work on the scenario by producing a requirements document [2], which consists of an explanatory text, diagrams, and requirements definitions. The requirements definitions are arranged using a specially-developed taxonomy which allows us to structure them according to various views on system behaviour, including: environment (`EN`), agent states (`ST`), service requirements and restrictions (`SV`), security (`SE`) and fault tolerance (`FT`). Each requirement is given a number within the group, for example:

```
EN 1: The scenario is composed of users,
locations and ambient computing environment (ACE)
```

```
ST 1: The agents' top-level states are lecture,
free, migrating, outside and emergency
```

```
SV 12: Teacher distributes lecture material
```

```
FT 14: Migration activity must tolerate
wireless disconnection and loss of ACE support
```

At the high level, the system consists of users (people participating in the scenario, i.e. teachers and students), locations (rooms with wireless connectivity) and *ambient computing environment* (ACE). ACE is composed of wireless hotspots, software agents and computing platforms (desktop computers or PDAs) on which the agents are run.

The interactions among users are done through agents. Each location provides a CAMA location middleware through which agents exchange information. Agents connect to the location middleware using the wireless hotspot available in each room.

[1]We use Java for developing the applications for PDAs.

Each teacher and student has an agent associated with him/her and assisting his/her participation in the lecture. During a lecture, teachers and students can be engaged in the following activities: lecture initiation, material dissemination, organisation of students into groups, individual or group student work, and questions and answers session.

## 3.3 Design

The ambient lecture system is being designed to meet the requirements in [2]. In this design, each classroom is a location with a wireless support, in which a lecture is conducted. An agent can take one of the two roles: teacher or student. The teacher agent runs on a desktop computer available in the classroom, while student agents are executed on PDAs (each student is given a PDA).

We use scoping mechanism described in Section 2.1 to structure the system. The teacher agent creates the outer scope constituting the lecture which student agents join. A lecture starts when there is one teacher agent and a predefined number of student agents joining this scope.

To support better system structuring, data and behaviour encapsulation, as well as fault tolerance, all major activities during the lecture are conducted within subscopes (nested scopes). The group work is one of the activities performed as a nested scope. Teacher – through his/her agent – arranges students into groups, so that only students belonging to the same group can communicate with each other through their agent. Each group is then given a task to solve (could be the same task for all groups). Students within the same group work together on the solution and present their answer at the end of the group work stage.

At the beginning of any lecture, all agents (teacher and students alike) are placed in the main scope. The teacher agent keeps a list of all students joining the lecture, and through the application's graphical user interface (GUI), the teacher can select which students to be placed within each group. Each group is given a unique name and the groups are mutually exclusive, i.e. a student cannot belong to more than one group. The teacher agent creates a subscope for each group and issues a *StartGroup* tuple to the student agents involved so that they automatically join the subscope they are assigned to. This is achieved by executing the CAMA `JoinScope` operation that uses the group name as a parameter. This structuring guarantees that while within a group, a student can only send messages to other students belonging to the same group, but he/she will also receive any message sent in the main lecture scope. To achieve this, the CAMA middleware creates a separate thread for each role inside a subscope.

Once a group is created, it is represented as a button (containing the names of the students assigned to this group) on the teacher agent's GUI. Clicking this button ungroups the students and issues a *EndGroup* tuple to the relevant student agents, making them invoke the `LeaveScope` command.

Following the fault tolerance requirements, the agents handle a number of potentially erroneous conditions. Some of them are detected by the agents themselves, others are detected by the middleware which raises predefined exceptions declared in the signatures of the CAMA operations. One example of these exceptions is the `CamaExceptionNoRights` exception, indicating that the agent concerned has no right to be in a particular scope, hence it cannot send or receive messages from the tuple space.

```
try {
  // Connect to the location middleware
  Connection connection = new Connection("Teacher",
    server, portNo);
  Scope lambda = connection.lambda();

  // Create a lecture scope that allows 1 Teacher
  // agent and up to 10 Student agents.
  ScopeDescr sd = new ScopeDescr(2, "lectureScope").
    add(new RoleRest("Teacher", 1,1)).
    add(new RoleRest("Student", 0,10));
  workScope = lambda.CreateScope("lectureScope", sd);

  // Join the scope and make the scope public
  workScope = workScope.JoinScope("Teacher");
  workScope.PutScope();
}
catch(CamaExceptionInvalidReqs e) { ... }
catch(CamaExceptionNoRoles e) { ... }
...
```

**Figure 3: Sample code: scope creation by Teacher agent**

## 3.4 Implementation

We developed an application for the group work activity described in section 3.3. There are two sets of agent software: `Teacher` and `Student`. Commands and data are passed as tuples through the tuple space provided by the location middleware.

Each agent runs at least two threads of execution: one thread handles the GUI and provides a means for sending tuples to the tuple space; another thread polls tuples from the tuple space and interprets the command contained in them. More threads are created when subscoping is used, so that an agent can also poll tuples from within the subscopes.

Figure 3 shows a snippet of the code for the `Teacher` agent, demonstrating how the lecture scope is initiated. Agents can join as a `Teacher` or a `Student`. In this example, only one `Teacher` agent is allowed, along with up to ten `Student` agents. An exception will be raised if this restriction is violated.

Figure 4 shows an example interaction among agents in the ambient lecture scenario. There is one `Teacher` agent, shown on the top of Figure 4. There are three `Student` agents: "Alice" (shown on the bottom left, this agent is run from a desktop computer), "Bob" (bottom right, run from a PDA) and "Tom" (not shown). At some stage, the `Teacher` agent places "Alice" and "Tom" into a group. While they are in this group, all messages they send can only be seen by other agents in the same group (group messages are indicated by a `(g)` in front of them). Teacher can end the group by clicking on the button representing the group (in this case, the "Alice-Bob" button). When this happens, all students in that group leave the group subscope and the subscope thread of execution terminates, but they all remain connected to the lecture main scope.

# 4. FUTURE WORK

Our long-term goal is to support formal development of fault tolerant mobile agent systems. To achieve this goal we are developing a number of formal notations and models defining the CAMA abstractions and the CAMA middleware (some initial results are reported in [9]). We are now work-

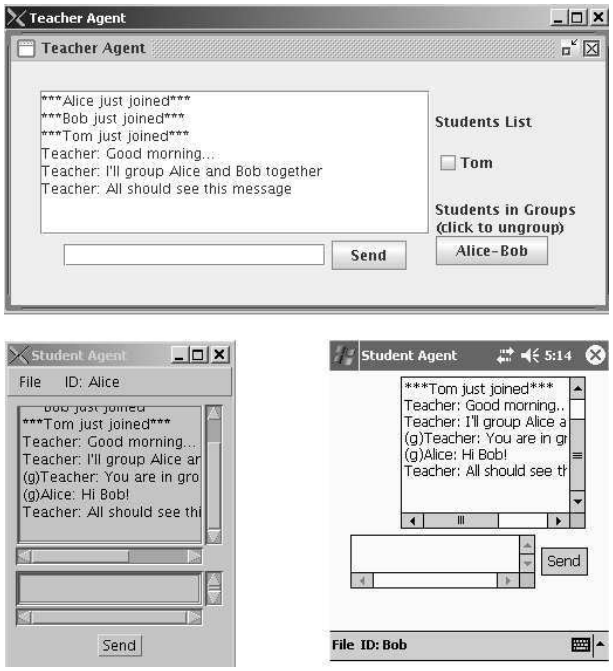


**Figure 4: Screen capture of ambient lecture agents**

ing on a top-down design methodology that insures that these systems are correct-by-construction. To ensure the application security, we will use an appropriate encryption mechanism that allows messages to be securely sent between PDAs and the location server. Our other plan is to implement the CAMA location middleware for PDAs to support applications in which locations are physically mobile. In our future work on CAMA for smartphone devices, we will address the facts that smartphones have capabilities that are different from PDAs. For example, smartphones utilise other means for connectivity (such as bluetooth and gprs), which might imply the need to adapt the communication support provided by CAMA.

## 5. ACKNOWLEDGEMENTS

This work is supported by the IST RODIN Project [1]. A. Iliasov is partially supported by the ORS award (UK).

## 6. REFERENCES

[1] Rigorous Open Development Environment for Complex Systems. IST FP6 STREP project, http://rodin.cs.ncl.ac.uk/ [Last accessed: 1 Feb 2006].

[2] B. Arief, J. Coleman, A. Hall, A. Hilton, A. Iliasov, I. Johnson, C. Jones, L. Laibinis, S. Leppanen, I. Oliver, A. Romanovsky, C. Snook, E. Troubitsyna, and J. Ziegler. Rodin Deliverable D4: Traceable Requirements Document for Case Studies. Technical report, Project IST-511599, School of Computing Science, University of Newcastle, 2005.

[3] F. Cristian. Exception Handling and Fault Tolerance of Software Faults. In M. Lyu, editor, *Software Fault Tolerance*, pages 81–107. Wiley, NY, 1995.

[4] R. J. Dufresne, W. J. Gerace, W. J. Leonard, J. P. Mestre, and L. Wenk. Classtalk: A Classroom Communication System for Active Learning. *Journal of Computing in Higher Education*, 7:3–47, 1996.

[5] G. Gay, M. Stefanone, M. Grace-Martin, and H. Hembrooke. The Effects of Wireless Computing in Collaborative Learning Environments. *International Journal of Human-Computer Interaction*, 13(2):257–276, 2001.

[6] D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[7] W. G. Griswold, P. Shanahan, S. W. Brown, R. Boyer, M. Ratto, R. B. Shapiro, and T. M. Truong. ActiveCampus - Experiments in Community-Oriented Ubiquitous Computing. *IEEE Computer*, 37(10):73–81, 2004. http://activecampus.ucsd.edu/ [Last accessed: 1 Feb 2006].

[8] A. Iliasov. Implementation of Cama Middleware. http://sourceforge.net/projects/cama [Last accessed: 1 Feb 2006].

[9] A. Iliasov, L. Laibinis, A. Romanovsky, and E. Troubitsyna. Towards Formal Development of Mobile Location-based Systems. Presented at REFT 2005 Workshop on Rigorous Engineering of Fault-Tolerant Systems, Newcastle Upon Tyne, UK (http://rodin.cs.ncl.ac.uk/events.htm), June 2005.

[10] A. Iliasov and A. Romanovsky. Exception Handling in Coordination-based Mobile Environments. In *Proceedings of the 29th Annual International Computer Software and Applications Conference (COMPSAC 2005)*, pages 341–350. IEEE Computer Society Press, 2005.

[11] I. Merrick and A. Wood. Coordination with Scopes. In *Proceedings of the ACM Symposium on Applied Computing 2000*, pages 210–217, 2000.

[12] G. P. Picco, A. L. Murphy, and G.-C. Roman. Lime: Linda Meets Mobility. In *Proceedings of 21st Int. Conference on Software Engineering (ICSE'99)*, pages 368–377, 1999.

[13] M. Ratto, R. B. Shapiro, T. M. Truong, and W. G. Griswold. The ActiveClass Project: Experiments in Encouraging Classroom Participation. In *Computer Support for Collaborative Learning 2003*, pages 477–486. Kluwer, 2003.

[14] G.-C. Roman, C. Julien, and J. Payton. A Formal Treatment of Context-Awareness. In M. Wermelinger and T. Margaria, editors, *Fundamental Approaches to Software Engineering, 7th International Conference, FASE 2004, part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, LNCS 2984*, pages 12–36. Springer, 2004.

[15] I. Satoh. MobileSpaces: A Framework for Building Adaptive Distributed Applications using a Hierarchical Mobile Agent System. In *Proceedings of the ICDCS 2000*, pages 161–168, 2000.

[16] A. Tripathi and R. Miller. Exception Handling in Agent-oriented Systems. In *Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems (SRDS'02)*, pages 304–315. ACM Press, 2002.

[17] P. van Roy, S. Haridi, P. Brand, G. Smalka, M. Mehl, and R. Scheidhauer. Mobile Objects in Distributed Oz. *ACM Transactions on Programming Languages and Systems*, 19(5):804–851, 1997.