On 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, and 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 within nested scopes. 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. This scenario is developed starting from a thorough definition of the traceable requirements including the fault tolerance requirements. This is followed by the design phase at which the CAMA abstractions are applied. At the implementation phase, the CAMA middleware services are used through a provided API. This work is part of the FP6 IST RODIN project on Rigorous Open Development Environment for Complex Systems.

Keywords: Mobile agents, exception handling, system structuring, coordination, middleware, Linda, ambient lecture

1 Introduction

The mobile agent paradigm is now used in developing a variety of complex applications as it supports systems structuring using decentralised, distributed and autonomous entities cooperating to achieve their individual aims. These applications include smart house, urban traffic management, information search and retrieval, Internet trading, network monitoring, load balancing, healthcare systems and enterprise quality management. The mobile agent paradigm promotes system openness, flexibility and scalability, and naturally supports mobility of code and devices. Very often the applications developed using agents must meet various dependability requirements. This, in particular, includes various business (money, information) and safety critical applications. This is why ensuring system fault tolerance is becoming imperative for successful deployment of

modern agent applications. Although there have been a number fault tolerance frameworks developed for agent systems, we have found that they have limited applicability due to several reasons. First of all, they typically focus on tolerating hardware faults, which, as a matter of fact, is not the main source of modern system failures. Secondly, they often provide means which are not adequate for achieving fault tolerance as they do not take into account the defining characteristics of the agent systems: agent mobility, autonomy and asynchronous communication, and system openness and dynamicity, which create new challenges for ensuring agent system fault tolerance. A typical example here is a naive assumption that the native Java and RMI exception handling is completely adequate for developing complex agent systems.

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 [1] 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 using the concepts of *tuples* and *tuple spaces*. A tuple is a data object that holds several objects; it can be seen as a vector of typed data values, some of which can be empty, in which case they match any value of a given type. A tuple space is an implementation of the content-addressable memory, providing a repository of tuples that can be accessed concurrently. It provides operations to allow processes to put tuples in it, get tuples out if they match the requested types, and test for them. Certain operations, like *get* (or `in`) can be blocking, whereas others, such as *test* (or `inp`) are non-blocking.

A number of Linda-based mobile coordination systems have been developed recently; these include Lime [2], Klaim [3], and TuCSoN [4].

Lime is one of the most developed, supported and widely-used examples of such environments. 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.

Klaim is a Linda-based process algebra with a notion of explicit locations. Absolute or relative location addresses can be attached to Linda operations

to specify the execution site of an operation. Klaim also has a type system extension used for control access. It is one of the few systems supporting strong code mobility [5].

TuCSoN [4] is another agent coordination system which is designed to be used with the existing mobile agent infrastructures. It mainly focuses on solving communication problems, but it ignores agent mobility and security. Coordination is based on the Linda tuple space paradigm. Each host provides a set of named tuple space which can be used for both local and remote coordination. A destination for a remote operation is specified using a tuple space name and a globally unique host name.

Exception handling [6] 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 [7] 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. 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. Fig. 1 shows how these abstractions are linked.

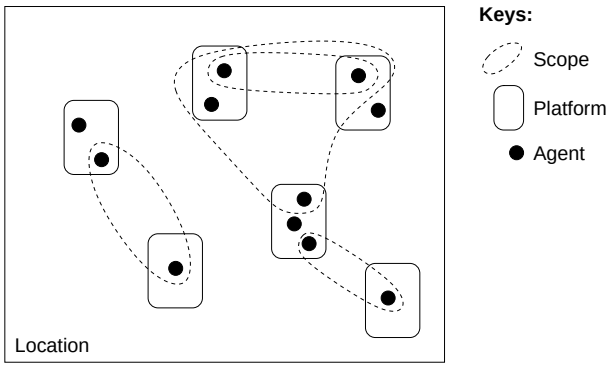

Fig. 1. Location, scopes, platforms and agents in Cama

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.

A location can be associated with a particular physical location (such as a lecture theatre, a warehouse or a 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 [8]. 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 a service discovery based on the scoping mechanism.

Scoping mechanism. The CAMA agents can cooperate only when they participate 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 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 the visibility of the 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 the newly created scope. The agent creating the scope can use this name to join the scope, to make the scope public (visible to other agents), to leave the scope and to delete it.

A 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 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. An 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. [9, 10]) are not explicitly developed to support data and behaviour encapsulation or isolation, which are 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 [7]). 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 an 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 an agent identifier in all other role operations.
- `disengage(a)` - makes the issued name `a` invalid.
- `create(a, n, R)@s` ($\mathtt{n} \notin \mathtt{l.s}$) - 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` ($\mathtt{n} \in \mathtt{l.s} \wedge \mathtt{a}$ *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` ($\mathtt{n} \in \mathtt{l.s} \wedge \mathtt{r} \in \mathtt{n} \wedge \mathtt{n}$ *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 has become engaged to 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 a special event notifying it about the 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 heart-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 (such as the connection-disconnection exceptions and the violation of the scope constraints exceptions).

In developing the exception handling support for CAMA, we relied on our previous work reported in [7], in which we proposed and evaluated a novel exception handling scheme developed for coordination-based agent applications. Here we give a brief overview of our exception handling mechanism; the full description can be found in [11]. 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 the agents. The ultimate task of the propagation mechanism is to transfer the 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 the 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.
- `wait` - waits until any inter-agent exception appears for the agent and raises it in the same way as the `check` operation.

Systematic use of exception handling should allow developers to design mobile agent applications tolerating 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, reduced amount of resource available, as well as users' mistakes.

Unfortunately, there has not been much work carried out in this area. Tripathi and Miller [12] 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 a concurrent object-oriented language called Oz [13]. 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 [14].

The CAMA middleware does not suffer from scalability problems inherent to system for distributed tuples spaces or a remote tuple access features. Due to the local nature of coordination in CAMA, the complexity of coordination rises linearly and has a small coefficient.

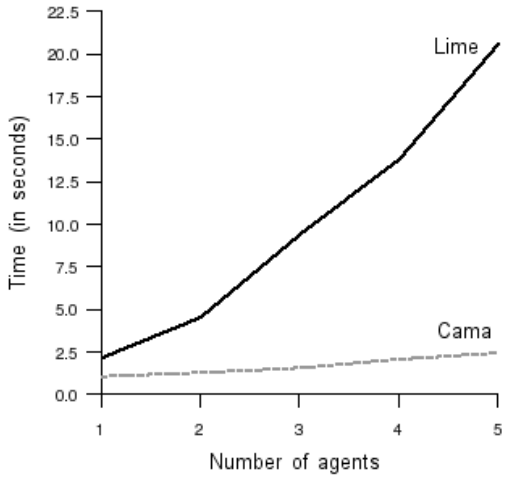

Fig. 2. The performance of Lime compared to that of CAMA

Fig. 2 compares the performance of Lime and CAMA systems. Results for both systems are given on the same scale. In each run, a given number of agents

perform non-destructive read on 1000 distinct tuples (each tuple is around 1000 bytes in size). The Y-axis represents the execution time in seconds and the X-axis represents the number of agents simultaneously reading from the tuple space.

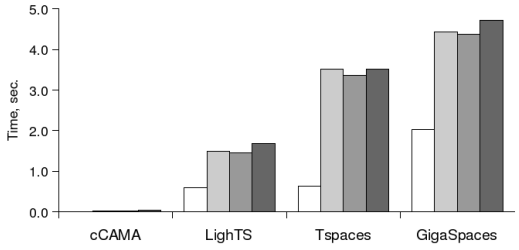

Fig. 3. Comparative performance of CAMA and other Linda-style tuple space systems

Fig. 3 presents another set of results from our experiment. Different bar shades correspond to different test cases. Test cases are made of a fixed number of `out` and `rd` operations with different tuple sizes and number of tuples. This experiment shows that CAMA performance compares favourably against several other Linda-style tuple space systems, such as LighTS [15] (which is a part of Lime), TSpaces [16] and GigaSpaces [17].

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.

A diagrammatical representation of the CAMA-based system architecture can be seen in Fig. 4. 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 or wired 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.

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

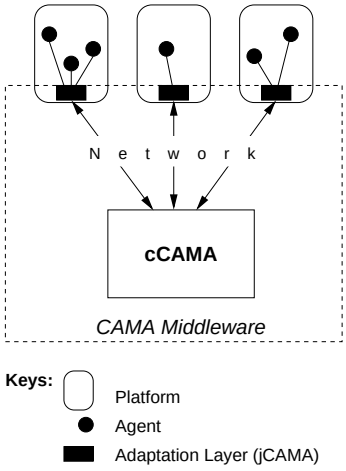

Fig. 4. CAMA architecture

3.1 Introduction

We focus on the activities performed by students and teachers during a lecture (the *ambient lecture* scenario – see [18] for more details) 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 [19] 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 [20] 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 [21]. 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 [22] 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 Sect. 2).

3.2 Traceable Requirements

We started our work on the scenario by producing a requirements document [23], 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 5: Lecture state has two sub-states: individual state and
group state
```

```
SV 1: For ACE-supported lecture to begin, there should be one
teacher agent and several student agents in the same location
```

```
SE 5: Each student agent belongs to only one group at any
given time during a lecture
```

```
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.

Each teacher and student has an agent associated with him/her and assisting his/her participation in the lecture. During a lecture, the teacher and the 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 [23]. 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 the scoping mechanism described in Sect. 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. The 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 – in this case, a B specification [24]. Students within the same group work together towards a solution, using a shared editor to modify the specification, and carrying out B operations such as proving and type-checking (which are provided by the system).

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, assigns a B project for this group to work on, 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 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.

The full details of the operations that can be carried out by both the teacher and the student agents during the ambient lecture can be seen in [18]. Here we outline the operations for the group work:

Teacher
The teacher prepares the group work by organising the students into groups, assigning a B project for each group to work on, and monitoring each group.

- *Assigns a B project to a group*
  Each group will be given a B project to work on, which contains at least one B machine specification that the students need to edit and run B commands on.

- *Watches the activity of each student*
  This monitoring activity is useful to measure each student's participation during the group work. A passive student might require further help or different group arrangements might be needed.
- *Inspects edited files*
  The teacher can check the progress of the group work by inspecting the changes that the students made on the files and by checking the status of the B commands already issued.
- *Assists by editing files*
  The teacher may modify the B machine specification files in order to make it clearer for the students how to solve the problem, or to "reset" the file if the students made too many mistakes.
- *Takes part in a discussion*
  The teacher may help the students to understand the problem they are trying to solve by asking probing questions as well as giving hints and advice.
- *Forces unlocking of resources*
  If a student appears to hold a file for too long (this could happen, say if the student agent crashes), the teacher can manually unlock the file to allow other students to edit it.

Student

The students' actions during group work mostly concern with editing B machine specification and carrying out the B commands such as proving and type-checking. They can also communicate with other student agents within their group, the teacher, as well other student agents in the global lecture scope. We are thinking about disabling the communication with other student agents in the global scope.

- *Chooses a file to work on within a project*
  Each project will have a list of associated files, and the student can choose which file to work on. This file represents a B machine specification and each student is allowed to work with only one file at a time.
- *Edits a file*
  There is a *shared editor* window that provides concurrency control (multiple readers, one writer) for editing a file. A student agent needs to obtain a lock before it can edit a file. We decided to use a non-blocking mechanism for obtaining the lock, so that the student can carry on with other activities if somebody else possesses the lock at that time. Only one agent can edit each file at any one time, although other agents can read the content of this file and see the update in real time. The lock must be released by the writing agent upon the completion of the editing process.
- *Proves/model checks/type-checks/does interactive proving*
  The Ambient Lecture software allows the students to carry out these commands on the B machine specification they are working with.

With the current implementation, the student agents are not required to obtain the editing lock first before carrying out these commands. We agree that this is not a desirable feature, and we will fix this in the later implementation.

- *Takes part in a discussion*
  During the discussion, students may ask questions, and other students in the group may provide the answer. If the questions remain unanswered, the group may ask the teacher for assistance.
- *Asks teacher's assistance*
  The teacher monitors group work, and from time to time, students may ask the teacher for clarification on the task they are working at.
- *Sends messages to other students*
  Students can send messages to other students in the same group.

Students cannot explicitly leave a group; only the teacher can decide whether a student must leave a group, for example at the end of the group work. Students can leave the Ambient Lecture setting altogether though, and when this happens, they will automatically leave the group subscope as well.

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.

3.4 Implementation

We developed an application for the group work activity described in Sect. 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.

Fig. 5 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.

Fig. 6 shows the "Lecture Overview" screen-capture of the `Teacher` agent. The icon **S** represents a student, the icon **G** represents a group, and the icon **R** represents a resource or a file containing B specification. It shows that there are

```
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) { ... }
...
```

Fig. 5. Sample code: scope creation by the Teacher agent

three `Student` agents: "Bob" and "Alice" (these agents are run from a desktop computer) and "John" (run from a PDA).

At some stage, the `Teacher` agent places Alice and John into "Group1". Alice is shown viewing a specification file called "Chat" while John is editing it. Fig. 7 on the left shows the screen-capture of the PDA used by John as he edits the Chat specification. Alice then asks John (through the group messenger) to carry out type-checking on this specification, as can be seen on the right hand side of Fig. 7.

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 [25]). We are now working 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.

Fig. 6. Screen capture of the Teacher agent's Lecture Overview

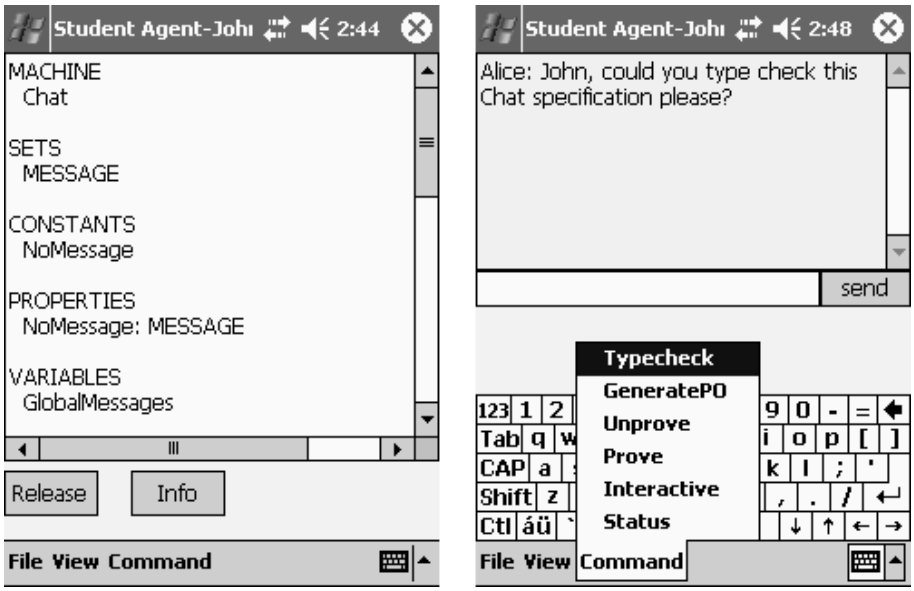

Fig. 7. Screen capture of John editing Chat specification and receiving a group message from Alice

5 Acknowledgements

This work is supported by the IST RODIN Project [26]. A. Iliasov is partially supported by the ORS award (UK).

References

1. Gelernter, D.: Generative Communication in Linda. ACM Transactions on Programming Languages and Systems **7**(1) (1985) 80–112
2. Picco, G.P., Murphy, A.L., Roman, G.C.: Lime: Linda Meets Mobility. In: Proceedings of 21st Int. Conference on Software Engineering (ICSE'99). (1999) 368–377
3. Bettini, L., Bono, V., Nicola, R.D., Ferrari, G., Gorla, D., Loreti, M., Moggi, E., Pugliese, R., Tuosto, E., Venneri, B.: The Klaim Project: Theory and Practice. In Priami, C., ed.: Global Computing: Programming Environments, Languages, Security and Analysis of Systems, LNCS 2874, Springer-Verlag (2003) 88–150
4. Omicini, A., Zambonelli, F.: Tuple Centres for the Coordination of Internet Agents. In: SAC '99: Proceedings of the 1999 ACM symposium on Applied computing, New York, NY, USA, ACM Press (1999) 183–190
5. Bettini, L., Nicola, R.D.: Translating Strong Mobility into Weak Mobility. In Picco, G., ed.: Proceedings of 5th IEEE International Conference on Mobile Agents (MA), LNCS 2240, Springer (2001) 182–197
6. Cristian, F.: Exception Handling and Fault Tolerance of Software Faults. In Lyu, M., ed.: Software Fault Tolerance. Wiley, NY (1995) 81–107
7. Iliasov, A., Romanovsky, A.: Exception Handling in Coordination-based Mobile Environments. In: Proceedings of the 29th Annual International Computer Software and Applications Conference (COMPSAC 2005), IEEE Computer Society Press (2005) 341–350
8. Roman, G.C., Julien, C., Payton, J.: A Formal Treatment of Context-Awareness. In Wermelinger, M., Margaria, T., eds.: 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. Springer (2004) 12–36
9. Satoh, I.: MobileSpaces: A Framework for Building Adaptive Distributed Applications using a Hierarchical Mobile Agent System. In: Proceedings of the ICDCS 2000. (2000) 161–168
10. Merrick, I., Wood, A.: Coordination with Scopes. In: Proceedings of the ACM Symposium on Applied Computing 2000. (2000) 210–217
11. Iliasov, A., Romanovsky, A.: Structured Coordination Spaces for Fault Tolerant Mobile Agents. In Dony, C., Knudsen, J.L., Romanovsky, A., Tripathi, A., eds.: LNCS 4119. (2006) 181–199
12. Tripathi, A., Miller, R.: Exception Handling in Agent-oriented Systems. In: Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems (SRDS'02), ACM Press (2002) 304–315
13. van Roy, P., Haridi, S., Brand, P., Smalka, G., Mehl, M., Scheidhauer, R.: Mobile Objects in Distributed Oz. ACM Transactions on Programming Languages and Systems **19**(5) (1997) 804–851
14. Iliasov, A.: Implementation of Cama Middleware. http://sourceforge.net/projects/cama (Last accessed: 3 Jan 2007)

15. Balzarotti, D., Costa, P.: LighTS: A Lightweight, Customizable Tuple Space Supporting Context-Aware Applications. In: Proceedings of the 20th Annual ACM Symposium on Applied Computing (SAC 2005), ACM Press (2005) http://lights.sourceforge.net/ (Last accessed: 3 Jan 2007).
16. IBM: TSpaces. http://www.almaden.ibm.com/cs/TSpaces/ (Last accessed: 3 Jan 2007)
17. GigaSpaces: Grid Computing - Distributed Computing Application Server. http://www.gigaspaces.com/ (Last accessed: 3 Jan 2007)
18. Troubitsyna, E., ed.: Rodin Deliverable D18: Intermediate Report on Case Study Development. Project IST-511599, School of Computing Science, University of Newcastle (2006)
19. Griswold, W.G., Shanahan, P., Brown, S.W., Boyer, R., Ratto, M., Shapiro, R.B., Truong, T.M.: ActiveCampus - Experiments in Community-Oriented Ubiquitous Computing. IEEE Computer **37**(10) (2004) 73–81, http://activecampus.ucsd.edu/ (Last accessed: 3 Jan 2007).
20. Ratto, M., Shapiro, R.B., Truong, T.M., Griswold, W.G.: The ActiveClass Project: Experiments in Encouraging Classroom Participation. In: Computer Support for Collaborative Learning 2003, Kluwer (2003) 477–486
21. Gay, G., Stefanone, M., Grace-Martin, M., Hembrooke, H.: The Effects of Wireless Computing in Collaborative Learning Environments. International Journal of Human-Computer Interaction **13**(2) (2001) 257–276
22. Dufresne, R.J., Gerace, W.J., Leonard, W.J., Mestre, J.P., Wenk, L.: Classtalk: A Classroom Communication System for Active Learning. Journal of Computing in Higher Education **7** (1996) 3–47
23. Arief, B., Coleman, J., Hall, A., Hilton, A., Iliasov, A., Johnson, I., Jones, C., Laibinis, L., Leppanen, S., Oliver, I., Romanovsky, A., Snook, C., Troubitsyna, E., Ziegler, J.: Rodin Deliverable D4: Traceable Requirements Document for Case Studies. Technical report, Project IST-511599, School of Computing Science, University of Newcastle (2005)
24. Abrial, J.R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press (2005)
25. Iliasov, A., Laibinis, L., Romanovsky, A., Troubitsyna, E.: 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)
26. Rodin: Rigorous Open Development Environment for Complex Systems. IST FP6 STREP project, http://rodin.cs.ncl.ac.uk/ (Last accessed: 3 Jan 2007)