Exception Handling in Coordination-based Mobile Environments

Alexei Iliasov
University of Newcastle upon Tyne
Newcastle upon Tyne, UK
Alexei.Iliasov@ncl.ac.uk

Alexander Romanovsky
University of Newcastle upon Tyne
Newcastle upon Tyne, UK
Alexander.Romanovsky@ncl.ac.uk

Abstract

Mobile agent systems have many attractive features including asynchrony, openness, dynamicity and anonymity, which makes them indispensable in designing complex modern applications that involve moving devices, human participants and software. To be comprehensive this list should include fault tolerance, yet as our analysis shows, this property is, unfortunately, often overlooked by middleware designers. A few existing solutions for fault tolerant mobile agents are developed mainly for tolerating hardware faults without providing any general support for application-specific recovery. In this paper we describe a novel exception handling model that allows application-specific recovery in coordination-based systems consisting of mobile agents. The proposed mechanism is general enough to be used in both loosely- and tightly-coupled communication models. The general ideas behind the mechanism are applied in the context of the Lime middleware.

1. Introduction

With recent innovations in mobile computing devices, there is a demand to develop and use a new kind of software that supports mobility and, by doing this, brings new features to end users. Along with new capabilities, mobility gives rise to a number of difficulties that never emerged in conventional programming. One of them is that an ordinary exception handling mechanism does not fit in with mobile coordination environments. Our analysis shows that the existing middleware solutions for mobile software do not address this problem adequately. This paper introduces a novel mechanism for exception delivery from one mobile communicating agent to another. The proposed mechanism is powerful enough to deal with both stationary and mobile agents without imposing any additional restrictions on their behaviour.

One of the most common coordination environments, Linda [1] is a set of language-independent coordination primitives that can be used for communication between and coordination of several independent pieces of software. Thanks to its language neutrality, Linda is quite popular and many programming languages have implementations of its coordination primitives. First used for parallel programming, it later became a core component of many mobile software systems because it fits in nicely with the main characteristics of the mobile systems: openness, dynamicity and loose coordination. Linda coordination primitives allow processes to put tuples in a tuple space shared by these processes, get them out 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’ and ‘test’, can be blocking. This provides effective inter-process coordination; other kinds of coordination primitives, like semaphores or mutexes, can be simulated with the Linda primitives in a straightforward way.

Lime [2] is a Linda-based coordination system specifically designed for mobile applications. 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 hosting environment 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 distributed tuple space and allows agents to treat it as normal tuple space using conventional Linda operations. However, it is possible to have a fine-grained control over the distributed tuple space. Agents may choose a tuple space of a particular agent as a source or destination for Linda operations.

In addition to all kinds of faults found in sequential and concurrent systems, mobile agents are susceptible to a number of unique faults and situations due to mobility, openness and asynchronous communication. Fault tolerance mechanisms can be created at different levels – that of hardware, the operating system, middleware or application. There are several schemes that focus on tolerating hardware and communication faults. Certain failures, such as connectivity loss, can be tolerated by moving transaction participants onto a single reliable host [3].

A few existing solutions for fault tolerant mobile systems do not provide any general support for structuring application-specific recovery. Thus, the guardian model presented in [4] introduces a global exception handling facility shared by several processes. It ensures required synchronization and exception resolution for tightly cooperating processes, like atomic action participants. It is unclear, however, if nesting of guardians is possible. A form of software redundancy, the *shadow agent*, can be used to tolerate unanticipated software and hardware faults [5]. One of the important issues involved in exception handling in agents is separation of normal and abnormal activity. Paper [6] presents a solution whereby recovery actions are contained in a separate *meta-agent*. Meta-agents can be updated during agent life and can handle exceptions for migrating agents. Mobile agent middleware systems have been developed since early 90s. Some of them use their own programming languages, others rely on existing ones. Interpreted and scripting languages can be successfully employed for simple code migration; for example, the *D'Agents* system [7] implements strong migration based on the TCL scripting language. Most of the recent mobile agent middleware projects are based on Java, which already has some basic notions of mobility and is platform-neutral [2] [8] [9]. There are a great number of other mobile agents and libraries that support mobile agent infrastructure (for example, there are links to well over 50 systems in [10]).

In this paper we introduce an exception handling support for mobile agent systems. We show how exceptions can be raised and propagated between agents, and how to decide on the agent and the handling method to deal with a particular exception.

2. Motivations and Requirements

Developing general mechanisms that would combine the Linda-based mobility with exception handling smoothly is a big challenge. 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 paradigm. However, many traditional fault tolerance and exception handling schemes are not applicable in such environments. For example, transactions involve tightly-coupled, frequently-synchronized parties. Their implementation for mobile agents would result in execessively restrictive agent behaviour patterns. In our work on developing exception handling mechanisms for mobile coordination-based systems we start from the premise that the interference of such mechanisms with the programming and behavior patterns should be minimized, with no restrictions imposed on mobility, anonymity or the communication model. At the same time, these mechanisms must ensure consistent and reliable handling of all exceptions to allow systems to ensure the required service. Exception handling mechanisms should provide a clear separation of system normal and abnormal behaviour, simple means for exception propagation and for finding the appropriate handler.

We use Lime middleware as a basis for our experiments; our view on the architecture of mobile agent software is different, however, from that suggested by Lime (see Section 6 for an explanation).

All the possibilities for handling thrown exceptions need to be employed. Even if for some reason we cannot deliver the exception to the destination agent at this particular moment, the exception must be either redirected to the next location where the destination agent might have moved, or handled by a local entity. This entity could be another agent or a specialized process left by the original agent to handle exceptions. In either case, we have to guarantee that the exception is eventually processed and appropriate recovery actions are taken. Since the mobile agent environment is highly dynamic and new configurations can be easily established over time, the configuration of the corresponding exception handling rules should be dynamic as well.

Unlike the conventional exception handling mechanisms (e.g. found in object-oriented languages), where we protect parts of the code with guards for exceptions, in the coordination environment we protect tuples emitted by agents. The conventional exception handling is used inside agents to recover from internal agent errors. Since agents may produce tuples that require different recovery actions, we should be able to separate recovery actions in several exception handling units. In addition, agents should be able to structure recovery actions from most general to more specific. This means that we need a provision for nesting handling scopes. If an exception is not handled within the current scope, the responsibility for exception handling is propagated to a higher level, i.e. more general, scope.

One non-trivial case is when an exception is thrown for an agent which has already migrated. As said above, our aim is to guarantee that any exception is eventually handled. For this purpose our scheme allows redirecting exception to a remote host and handling it by a different agent or by a special handler code left by the agent before migrating. Redirection means sending and reraising the exception in a different location. The exception may pass through several locations before it reaches the agent, which makes it necessary to employ some mechanism to preclude loops and excessively long travel paths. Handling delegation can be used if there is a friendly agent that can perform exception handling when the original agent is not present in the location. Such friendly agent may be just a spawned version of the original agent or a dedicated stationary agent that handles exceptions for a whole class of mobile agents. Overall, the general strategy is to be adaptable and provide agent developers with a good range of error recovery solutions to choose from. All these methods will need to complement each other to achieve our ultimate goal – a guaranteed and predictable exception handling.

The requirements of reliability and predictable system behaviour during exception handling exclude the usage of federated tuples spaces. The approach that we adopted is based on a stationary and persistent tuple space provided by a *location* (see Section 6 for an explanation). By using a stationary tuple space we achieve full control over the communications happening in the space and can guarantee certain features important for the exception handling mechanism without compromising any characteristics of mobile agent systems.

The requirements for exception handling in mobile coordination systems can be summarised as follows:

1. Exceptions should be raised and handled asynchronously.
2. Agent anonymity must be preserved.
3. There should be no restrictions imposed on the agent behavior or its internal activity.
4. The exception handling mechanism should be flexible enough to support migrating agents.
5. It should be effective for both loosely- and tightly-coupled communication patterns.

3. Exception Handling Model

Our model uses tuple space (TS) exceptions to provide support for exception handling in Linda-based communication middleware. The main functionality of this scheme is to allow exception propagation between mobile agents communicating via tuple spaces. It is quite obvious that sometimes agents will be unable to recover from the exceptions caused by bad data in the read tuple. The most natural thing for the agent in this situation is to abort the current session and delegate handling to the original producer of the trouble data. Our solution tries to deliver such ability to agents though asynchronous communication presents a serious obstacle. However with some help from the agents our exception handling mechanism can reliably deliver exceptions even across several locations.

Agents communicate by anonymously placing and reading tuples from a tuple space (Figure 1). Read (i.e. get) and write (i.e. put) operations may be separated by a large time gap.

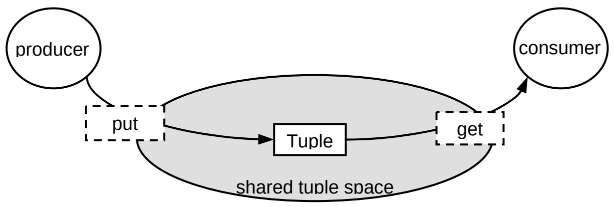

Figure 1. Linda agents communicate by exchanging tuples of data using a shared tuple space

When an exception happens in the consumer the producer may have already migrated, switched to another activity, etc. In spite of this the recover actions on the consumer side may require sending an exception to the producer (Figure 2) and that is exactly what our proposed mechanism provides.

The transportation medium for the TS exception is a tuple space. To make the proposed EH mechanism universal and portable we only use commonly available features. In the most mobile agent systems the tuple space is the only inter-agent communication channel and thus the only way to pass exceptions.

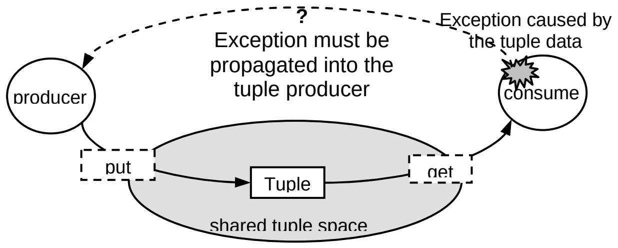
?
Exception must be propagated into the tuple producer
Exception caused by the tuple data
producer
consume
put
Tuple
get
shared tuple space

Figure 2. A mechanism to propagate exceptions between agents

However, we require that communication middleware implement strong logical separation of the normal and exceptional tuples, for example, by using a dedicated tuple space for exceptions. A special system service – the tuple space exceptions agent (*TSE agent*) – is responsible for processing and routing the exceptional tuples. We want to make exception routing independent from the agent activity, locality, connectivity and even existence (agents may cease to exist before a thrown exception is handled). It is also a security measure as we need to hide information that refers to the producer of a particular tuple. The TSE agent finds a handling rule for the thrown TS exception and routes it to some location or an agent that will perform the actual handling. Routing of exceptions is made on the basis of agent-specific information. The TSE agent gets this information from the tuples of a special kind, called tuple space traps (*TS traps*).

Agents interested in handling TS exceptions can produce these tuples with a help from the underlying middleware. TS traps can be updated or removed at any time thus enabling dynamic exception handling patterns. TS traps are organized in hierarchical structure; this allows agents to associate a set of reactions contained in TS trap with a set a tuples or regions of a program. The later is especially important for building gateway between TS exceptions and conventional exception handling mechanism. Unlike normal tuples, TS exceptions are always addressed. In general, they may reach multiple destinations.

Our Lime-based implementation uses a system tuple space to store TS traps and process TS exceptions. In Lime, normal tuple spaces, associated with agents, migrate along with their agents. System tuple space is associated with Lime server and is always stationary. It is imperative to process TS exceptions and store TS traps in a host-stationary tuple space.

The TS traps are attached using the normal method calls. Thus the creation and the structure of TS traps can be defined dynamically, as required by the state of the agent or by the configuration of a multi-agent system.

Two TS operations (Figure 3) are provided: inX() – any read operation, outX() – any write operation. The TS exceptions are propagated in the following basic steps:

1. Consumer throws TS exception thus creating a proto exception (EP in circle on the Figure 3) in the local system tuple space. It contains the EHTag field from the original troubling tuple.
2. When a proto exception appears, the Guard agent wakes up and consumes it. Then it finds out how and where to propagate the exception using a set of rules given in the TS trap that is pointed by the EHTag field in the proto exception.
3. If the exception is to be delivered to the producer it is placed into the system tuple space as a final exception (EF in circle on the Figure 3).
4. When exceptions appears, the producer agent may react asynchronously (using reactions), poll at some points for a presence of the TS exceptions or abort the current blocking Linda operation to handle the exception.

The guard agent is a service running on each server that hosts tuple space. It runs in its own thread and has a number of privileges like access to hidden fields and TS traps. Guard uses the EHtag field of the proto exception to locate the first TS trap tuple that will be used for handling the current TS exception. To throw a TS exception agent must supply a tuple produced by another agent as the source of the EHTag value. A proto exception is a tuple of a special structure located in the

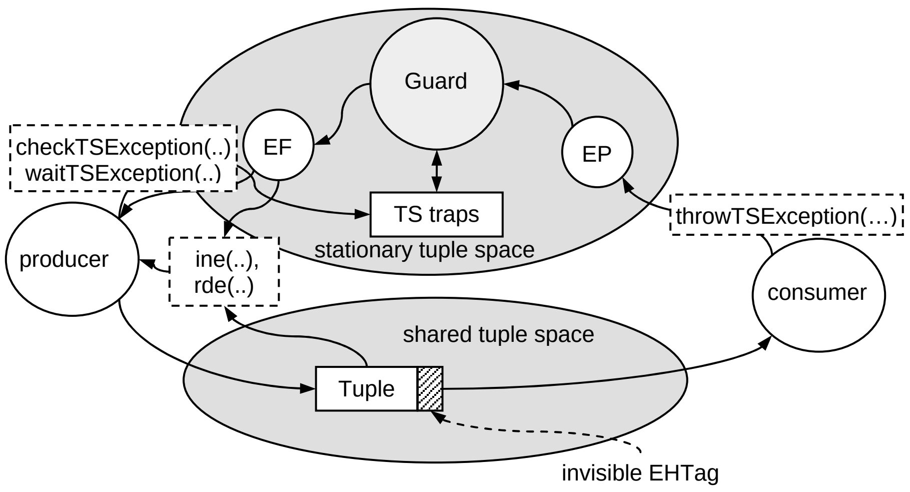

Figure 3. A tuple space exception mechanism for Lime

system tuple space of the current host. The guard agent uses traps as the rule set for propagating exception thrown for a particular agent. Eventually the proto becomes a final exception. The final exception is a tuple in the system tuple space created by the Guard agent as an indication of an exception raised by some other agent. It is up to the agent to consume and to handle this exception. However, the middleware implementation can make this obligatory. Proto and final exception tuples are invisible to the normal agents; they cannot be directly produced or read by them with the normal Linda operations.

The TS trap rules are organized into a tree-like hierarchy where more general rules are placed on the upper-tree nodes. This tree can be associated with nesting of the exception handling contexts scopes in the agent code, though it is not strictly required. Pointers to traps are contained in all tuples produced by a given agent. A tuple may bring pointers to different traps, depending upon the context of the tuple creation. An important implication of this scheme is that it is impossible to raise a TS exception without first having read a tuple of the destination agent.
To read someone's tuple agent has to know its structure. Thus, only agents that communicate with each other can send TS exceptions. Normally agents throw TS exceptions as a reaction to a particular tuple produced by another agent. But the identity of the agent that produced that tuples remains hidden for them. Only the Guard agent possesses information that points out at the owner of the tuple. This is one of the reasons why processing of the TS exceptions is done at the system level. TS trap is very powerful and general construct. The next section discusses the exception handling rules that can be used in a trap.

4. Java/Lime based implementation[1]

The API of the tuple space exceptions (TSE) can be divided into three sections: operations for throwing, checking and waiting for TSE, operations for setting up the TSE traps in the server and private tuple spaces and semantically extended Linda operations. There is one method for throwing TS exception and it takes two arguments – the trouble tuple and the exception to be thrown.

There are three ways for TS exceptions to manifest themselves. Agents may poll for any pending TS exception, for example at the end of some communication operation. If there is at least one unhandled TS exception this Java method is completed with an appropriate Java exception. In addition to polling, agents may wait for TS exception to appear, this may be useful in particular situations, for example when TS exception handling is asynchronously executed by a separate thread. And finally, special versions of blocking Linda operations will return when a TS exception appears. This allows synchronous exception handling, when handling rules can be attached to the section that produces bad tuples.

TS traps are a serialization of the LimeEHTuple class instances. They are placed in the system tuple space and control the TS exception propagation for the particular agent. The class contains various methods for associating reactions with exceptions. Exceptions must

[1] The full code of our implementation can be downloaded from http://www.cs.ncl.ac.uk/~alexander.romanovsky/home.formal/limeh.zip. The complete description of the API and Lime extended operations can be found in Exception Handling in Coordination-based Mobile Environments, Iliasov, A., Romanovsky, A. School of Computing Science, University of Newcastle, CS-TR 878, 2004.

be subclasses of the TSException class which extends the standard Java Exception class. There are following exception handling methods:

- **throw** – deliver exception to the agent;
- **relay** – propagate exception to a different location;
- **abort** – leave the current scope;
- **delegate to an agent**.

Operation relay assumes that a communication channel exists between the locations. This channel does not have to be a tuple-space based and is totally hidden from agents. For more explanations on our locations-based approach see Section 6.

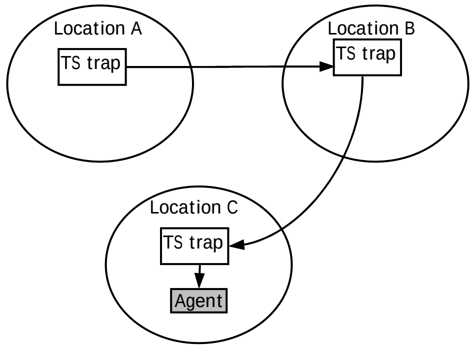

Figure 4. RELAY reaction allows a TS exception to trave between several locations treach a migrated agent

When action abort is set the higher-level (more general) trap is used to handle the current exception. If there is no upper-level trap the system middleware attempts to report this situation to both the sending and destination agents.

In addition, actions can be concatenated and executed conditionally.

With such flexible reactions, the TS traps constitute a quite powerful instrument for implementing almost any exception handling strategy. Using action concatenation, several agents may cooperatively recover from an exception. This is an essential feature for implementing transactions in mobile environment. The conditional action allows agents to decide on the way in which TS exception handling will go depending upon the current state of the environment. For example, an agent can handle all exceptions itself, however when it moves away exceptions are handled by a dedicated process or another agent. As soon as it returns back into the location, it starts receiving the exceptions again.

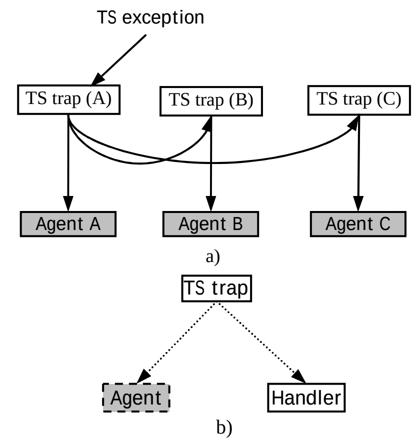

Figure 5. a) an exception is automatically propagated in a group of agents participating in a transaction using action concatenation; b) TS trap chooses where to send exception depending upon some external condition, such as agent's locality

5. Case study

In this case study several agents play a simple word game. They use tuple space exceptions to clearly separate the normal and exceptional control flows. The game starts when two agents willing to play this game (and knowing how to do this) meet at the same tuple space. They do some kind of a handshake, learn each other names and start playing. The idea of the game is the following – one of the players says a name of a city and another must answer with another city name beginning on the last letter of the first one. The agent will lose if it is unable to find an appropriate answer. Agents may not answer with a word that was already used in this game. Below is a short transcript of a game between two agents – Alice and Bob:

```
Alice - Hello, Bob
Bob - Hello, Alice
Alice - 'Warsaw'
Bob - 'Washington'
Alice - 'Nantes'
```

Writing such an agent may look trivial at the first glance. However, a more thorough consideration reveals several interesting details. Since agents are developed independently, there is a real competition and output of a game is unknown. Agents may try some tricks like giving the same word twice or fabricating illegal words from a random selection of letters. They should also expect the same from their counterparts and take measures to detect invalid words. We will consider two configurations of mutually mistrusting agents playing this game and see how the tuple space exceptions can help to build them.

In Case 1 two agents play against each other, trying to win using all the means they have and also trying to detect any illegal words produced by their counterparts. In Case 2 we add another agent that serves as a judge that checks all the words and may stop the game if one of the agents cheats. In the third case study we add mobility. The whole activity of the three agents can migrate into a new location. This process is made in several steps and is initiated by one of the agents.

Case 1. Two agents, B(ob) and A(lice), are playing against each other. They use the TS exceptions to indicate various abnormal situations such as an illegal city name, inability to find an answer and so on. This configuration is shown on Figure 6.

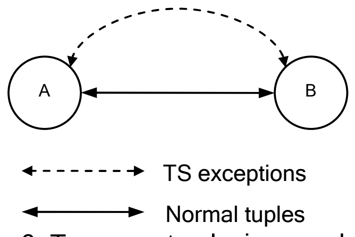

Figure 6. Two agents playing word game with TS exceptions

A typical work cycle for these agents without TS exceptions is the following:

```
repeat {
       take tuple with a city name, in()
       find an answer
       put the answer in the TS, out()
}
```

The major problem of the algorithm above is that a deadlock will happen eventually. This may happen if one of the agents is unable to find an answer or just wants to quit the game and thus a tuple expected by its partner never appears in the tuple space. Active polling for a matching tuple or using timeouts is a poor solution here, which also brings many complications to initially very straightforward scheme. Tuple space exceptions help us to solve this problem gracefully:

```
create and activate TS trap for general
recovery actions
try {
 repeat {
 create TS trap for recovery without
 stopping the game
  try {
   take tuple with a city name,
   InTuple = ine()
  } catch(TS.. e) {
   handle the exceptions from
   which we can recover without
   ending the game
  } catch(TSException e) {
   abort the current scope with
   an unhandled TS exception
   say to the peer that game is aborted
   if (InTuple != null)
    throwTSException(InTuple,
      new TSBreakException());
   throw;
  } catch(...)
    Handle other non-TS exceptions
  }
  find an answer
  put the answer in the TS, out()
 }
} catch(...) {
 handle other TS and normal exceptions
}
```

Blocking read operation *in()* was replaced with *ine()* which may be interrupted by a TS exception. This allows synchronous reaction to abnormal situations without active polling. There are two levels of TS exceptions handling. The outer one provides general recovery actions for exceptions like connection loss, problems in middleware, serious error in the peer agent and so on. Upon recovery from these exceptions, the agent is expected to return to the initial state to later start another game. The purpose of the inner handling sections is to recover from the less serious exceptions without ending the game. Some examples of such exceptions are the refusal of the last word, request to restart the game or may be proposal of a draw.

Case 2. New agent J watches agents A and B playing and prevents invalid answers. A and B now use non-destructive operation *rde()* to retrieve words from the tuple space to guarantee that J can check all the words in background, J is also responsible for deletion of outdated tuples.

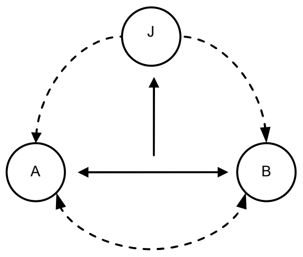

Figure 7. Two agents play while the third one forces them to obey the game rules

J sends an exception to one or both sides if it detects situation when one of agents cheats or breaks game rules. The flow of normal data and exceptions in this configuration is presented on the Figure 7.

Code for A and B is almost the same except for: *ine()* is replaced with rde() and inner try-catch block is extended to catch exceptions sent by agent J. The pseudo code for J is as follows:

```
create and activate TS trap for general
recovery actions
try {
 repeat {
  learn player names
  create and activate TS trap for
  recovery without stopping the game
   try {
    read next word tuple,
    InTuple = rde()
    remove previous word tuple, if any
   } catch(TS.. e) {
    handle the exceptions from agents like
    game abort, connection lost and other
    situations.
   } catch(TSException e) {
    abort the current scope with an
    unhandled TS exception and break
    association with other agents
   } catch(..) {
    handle other non-TS exceptions
   }
   check the word against the
   spell-checker database and if this
   word was already used in the game
   if word is invalid stop the game,
   send WinnerTSException to the winner
   and BadWordTSException to the agent
   that emitted the wrong word.
  }
  remove trap
} catch(..) {
 handle other TS and normal exceptions
}
```

J iteratively checks tuples placed by agents A and B using its own internal algorithm. Before the game starts agents must agree upon the same instance of J agent. There may be a number of various J agents implementation in the same tuple space and each with its own rules and capabilities of judging the game. J agent never produces any tuples itself; it can only send TS exceptions to the playing parties to indicate various game situations.

A transcript of the actual game between the agents is given below. Note that J has changed the game result by denying one of the words, which it believes was incorrect. It has happened after one of the sides had already given up on that incorrect word.

```
Alice - Hello, Bob
Bob - Hello, Alice
Alice - I will start the game
Bob - Hi, Jack
Alice - Hi, Jack
Jack - Hi, Bob
Jack - Hi, Alice
Jack - See two players playing - Bob and Alice
Alice - 'Nante'
Jack - Word 'Nante' is ok
Bob - 'Ekatirenburg'
Alice - I give up!
Jack - Word 'Ekatirenburg' is invalid
Bob - Nice, I am the winner
Bob - BadWordTSException: Ekatirenburg
Bob – Hmm, my word 'Ekatirenburg' is invalid
Alice – Bob cheated, I won!
```

This configuration of playing with a third party turned out to be rather effective and simple to implement. Without the TS exceptions, agents would have to actively poll for various kinds of tuples and watch for a large number of abnormal situations along with the main activity. With the TS exceptions, the agent code is well structured and highly adaptable to different kinds of communication patterns.

Case 3. The previous case study is modified so that agent B at some random moment can change its location. By default this would cause TSBreakException in agents A and J. However, just before the migration, agent B throws a parameterized extension of exception TSBreakException with the parameter describing the destination location to A and J to indicate that it is leaving. Agents A and J migrate to that location, reestablish a shared tuple space, make a new handshake and continue the game from the point where it was interrupted. We can also consider a situation when one of the agents, say J, cannot migrate, because of binding to a local resource, such as spelling dictionary. To handle this situation A and B attach additional TS trap that will relay TS exceptions to a remote location. The locations involved must be connected via LAN or Internet for TS exceptions relay to work. It is J's responsibility to inform its peers about the problem with migration. It does this by sending a TS exception to A and B that is automatically routed to a remote location (see Figure 8). This exception makes A and B to look for a new judge locally and if there is none they have to stop the game. If a new judge agent is found, the game will continue from the point where it was left before the migration.

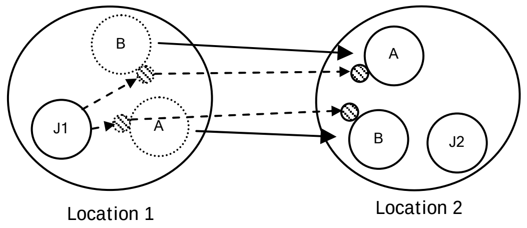

Figure 8. TS exceptions (⊘) are routed by the Guard to the location 2 to reach the migrated agents

6. Discussion and Conclusions

6.1. Location-based middleware

Mobile agent middleware systems are often symmetric in a sense that each system participant roughly carries the same middleware implementation. Agents can dynamically and autonomously form new groups and communicate. However in this paper we explore an asymmetric approach in which different parts of the system carry different basic functionality. One particular example of such view is a location-based scheme. In this model locations provide services to the agents, such as connectivity and coordination space. Agents are not able to communicate with each other without a location support. The choice of the scheme is supported by our analysis that shows that the majority of the mobility applications assume that agents meet in physical or logical locations providing a set of designated services to them.

Let us sum up the pros and cons of these two approaches. In the symmetric scheme agents are autonomous as they carry with them everything they need to establish an ad-hoc network. There is no single failure point since a failed agent does not affect the rest of the agents. Disadvantages are the direct consequence of the advantages. Running the same middleware version on a powerful desktop and on a PDA does not seem to be an efficient solution. It is quite natural to expect that more powerful nodes would provide services to less capable. In addition, the full agent autonomy is still not well supported by the current technology. For example, application of the ad-hoc model is often reduced to peer-to-peer communications as there is no standard WiFi ad-hoc mode routing capability in the consumer PDAs.

The asymmetric scheme is closer to the traditional service provision architectures. It can support large-scale mobile agent networks in a very predictable and reliable manner. It makes better use of the available resources since most of the operations are executed locally. Moreover the location-based architecture eliminates needs for employing complex distributed algorithms or any kind of remote access. This allows us to guarantee atomicity of certain operations without sacrificing performance and usability. This scheme also provides a natural way of introducing context-aware computing by defining location as a context. The main disadvantage of the location-based scheme is that an additional infrastructure is always required to support mobile agent collaboration.

6.2. Conclusions

The proposed mechanism brings full-fledged exception handling support into mobile agent applications. The mechanism is asynchronous; it preserves agent anonymity and can be easily incorporated into almost any coordination-based mobile agent middleware. This type of handling exceptions fits well into the main characteristics of the pervasive systems, open and dynamic by their nature as it does not impose any restrictions on asynchrony and dynamicity of agents. Our Lime-based implementation does not consume any CPU ticks when there are no exceptions raised.

This mechanism by itself does not always guarantee exception delivery. We believe that this is an unavoidable consequence of the asynchronous communication style of agents. Instead of imposing excessive restrictions on the agent activities we offer a rich exception propagation mechanism that, if supported by a proper agent programming, can deliver exceptions to agents in almost any abnormal situation when handling is required.

We do realize that many related problems were not covered in this work, including garbage collection, agent naming, security problems, etc. We will address some of them in our future work

Our main plans for future work include design of a scheme for deriving and using agent interfaces. These interfaces will ensure agent compatibility at a number of levels, from compatible tuple structure to common semantics of thrown exceptions. We plan to develop a support for scoping and nesting tuple spaces, and to combine it with exception propagation by assigning tuple space traps to particular scopes. There is also a need to design a common multi-party communication schemes with exceptions, examples of which are atomic actions, voting, multicast and broadcast algorithms. The main theme of our future work is developing a formal design methodology for fault tolerant coordination-based applications. Among many other benefits and improvements this will allow us to verify that for any specific application all TS exceptions are delivered.

Acknowledgments. This work is supported by the IST RODIN Project and by the School of Computing Science at the University of Newcastle upon Tyne (UK). A. Iliasov is partially support by the ORS award (UK).

References

[1] D. Gelernter. "Generative Communication in Linda". *ACM Computing Surveys*, 7(1): 80-112, 1985.

[2] G. P. Picco, A. L. Murphy, G.-C. Roman. "Lime: Linda Meets Mobility". *Proc of the 21st Int. Conference on Software Engineering (ICSE'99)*, Los Angeles (USA), May 1999.

[3] A. I. T. Rowstron. "Mobile Co-ordination: Providing Fault Tolerance in Tuple Space Based Co-ordination Languages"**.** *Proc of the 3rd Int. Conference on Coordination Languages and Models*. 1999, pp. 196–210.

[4] A. Tripathi, R. Miller. **"**Exception handling in agent-oriented systems". *Proc. of the 21st IEEE Symposium on Reliable Distributed Systems (SRDS'02)*, 2002. pp. 304-315.

[5] S. Pears, J. Xu, C. Boldyreff. "Mobile Agent Fault Tolerance for Information Retrieval Applications: An Exception Handling Approach". *Proc. of the 6th Int. Symposium on Autonomous Decentralized Systems (ISADS'03)*, 2003.

[6] G. Di Marzo, A. Romanovsky. "Designing Fault-Tolerant Mobile Systems". *In N. Guelfi, E. Astesiano, G. Reggio (Eds.). Scientific Engineering for Distributed Java Applications Int. Workshop, FIDJI 2002*, Luxembourg, LNCS 2604. 2003, pp. 185-201.

[7] R. S. Gray, G. Cybenko, D. Kotz, R. A. Peterson, D. Rus. "D'Agents: Applications and Performance of a Mobile-Agent System". *Software - Practice and Experience*, 32(6):543-573, May, 2002.

[8] R. de Nicola, G. L. Ferrari, R. Pugliese. "KLAIM: A Kernel Language for Agents Interaction and Mobility". *IEEE Trans. on Soft. Eng*. 24(5): 315-330, 1998.

[9] C. Bryce, C. Razafimahefa, M. Pawlak. "Lana: An Approach to Programming Autonomous Systems". *Proc. of the 16th European Conference on Object-Oriented Programming, ECOOP'02*, 2002.

[10] "The Mobile Agent List" *reinsburgstrasse.dyndns.org /mal/preview/preview.html.*