Click here to view PowerPoint presentation; Press Esc to exit

The Use of Tryllian Mobile Agent Technology in Military Applications

Ms. Christine Karman
Tryllian BV
Joop Geesinkweg. 701
1096 AZ Amsterdam
The Netherlands

1 Introduction

1.1 TARGET AUDIENCE

This technical white paper discusses version 1.3 of Tryllian's Agent Development Kit (ADK). The goal of the paper is to provide experienced Java programmers with a feel for what they get by using the ADK.
For information with a more commercial angle (e.g. applicability of agents, benefits, availability of the ADK etc.) please visit our website, www.tryllian.com.

1.2 BACKGROUND

The ICT world is synonymous with change, and to keep up with the speed and sophistication at which this takes place, appropriate services need to be provided. These services not only need to be reliable but they also need to be fast, mobile, flexible, and cross-platform, all aspects of what mobile agent technology has to offer. Mobile agents can be used as solutions interfaces in various areas such as, e-commerce, network security and wireless computing.

Commercial services and infrastructures using agent technology will be able to provide these benefits for their users, allowing them to provide better service along with creating time and money savings. Flexibility, through extensibility and interoperability will allow new and legacy systems to benefit.

Paper presented at the RTO IST Symposium on "Technology for Evolutionary Software Development", held in Bonn, Germany, 23-24 September 2002, and published in RTO-MP-102.

From a technical standpoint, mobile agents are independent software programs that can transparently mobilize entire applications or carry out tasks autonomously, and have the ability to travel across any number of networks or other devices, or in other words, distributed objects. Agents can therefore transmit information, or request services from each other, across a widely distributed heterogeneous network. The current trend towards distributed computing underlines this vision and this principle.

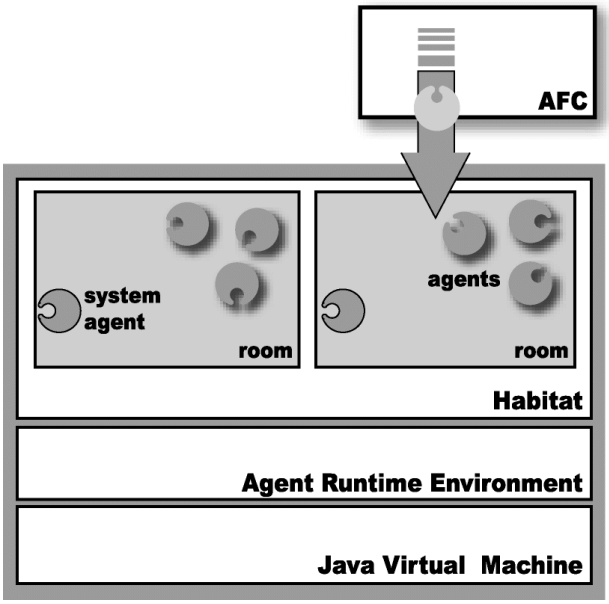
Tryllian's software provides the necessary elements such as security, mobility, intelligence and various sets of plug-in behavior. By combining these various sets, highly modular and scalable systems can be built using agent "building blocks." Expanding on them, system integrators can use our development tools to build any type of mobile agent framework or application they wish.

2 The Agent Development Kit

The kit is made up of different components that work together with each other to form a developing and runtime environment. First of all below you can see from the diagram how the architecture and different components work with each. After that we will give a brief description of some of the most important components of the ADK.

1.3 OVERVIEW

The picture below shows the architecture and main concepts of Tryllian's agent world. The various concepts are explained in the following paragraphs.



Two of the most important components of the ADK are the AFC and the ARE. Most developers will interact with the AFC to construct their agents. The AFC encapsulates all the functionalities of the ARE and therefore you will normally not need to use the ARE directly. When you want create functionalities that are not provided by the AFC, you will then directly interact with the ARE.

1.4 THE AGENT FOUNDATION CLASSES (AFC)

The Tryllian ADK allows application programmers to define all of the components that are required to build an agent-based application. The AFC is the interface layer and contains all interfaces and classes needed to interact with the agent-based elements. All of the API's that are seen by the developer can also be found here.
An important part of understanding the development kit is the use of the communication protocols (see below). The ADK provides some easy to use tasks that make the actual language used extremely simple.
By using the AFC classes, you can:

- Create your own agents
- Create tasks for your agents to perform
- Create messages for your agent to send to other agents
- Create your own agent languages
- Log your agent's activities
- Use the ADK's development tools and support software to test your agents

1.5 THE AGENT RUNTIME ENVIRONMENT (ARE)

All communication between individual agents and the system is implemented by the ARE in the form of messages sent to system agents. The ARE runs on top of the JRE (Java Runtime Environment). Because of this a habitat can run on any machine that is running an operating system that supports Java 1.3 or higher

1.6 HABITAT

A habitat is a collection of one ore more rooms that share a common code base and a Java Virtual Machine. It provides services like the agent execution model, inter-platform communication, inter-habitat travel, room and agent persistence and the security model. Agent applications typically exist of several habitats, each hosting a number of rooms.

1.7 ROOMS

Rooms are the travel destinations for mobile agents. Agents can only exist in rooms. Rooms typically provide agents with a registry service where agents can check in and check out when they enter or leave the room. This is also where agents can advertise their properties and capabilities, in order for other agents to find them.
Rooms can be created when the habitat is start-up (by specifying them in the habitat configuration file) or dynamically by agents with sufficient privileges.

1.8 SYSTEM AGENTS

All actions in the ARE are requested by messages sent to a number of system agents. They are based on the same philosophy as general agents, but they are created and maintained by the ARE. Developers will interact with agents through messages, this is the only form of communication that a developer will have with the agents. To make life easier a lot of this communication has been wrapped and is available through predefined tasks.
The most important system agents are:

- *Habitat agent.* This agent allows communication with the entire habitat.
- *Room agent.* The room agent is the primary point of interaction for any agent. Agents can query this agent for information on their environment (rooms, other agents etc.).
- *Transporter agent.* The transporter agent is the point of interaction for an agent that wants to move to another location.

In general systems agents are not mobile.

1.9 AGENTS, TASKS AND COMMUNICATION

A Tryllian agent consists of two parts: its body and its behavior. The body is the part of the agent that executes the agent code, sending messages and moving the agent code over the network. It is the most 'technical' part and least configurable.
The behavior specifies the actions and the knowledge of an agent. It has the form of a task model: a set of interdependent tasks. Communication between agents and between an agent and its environment (i.e. system agents) is done by sending and receiving messages.

3 Development and deployment tools

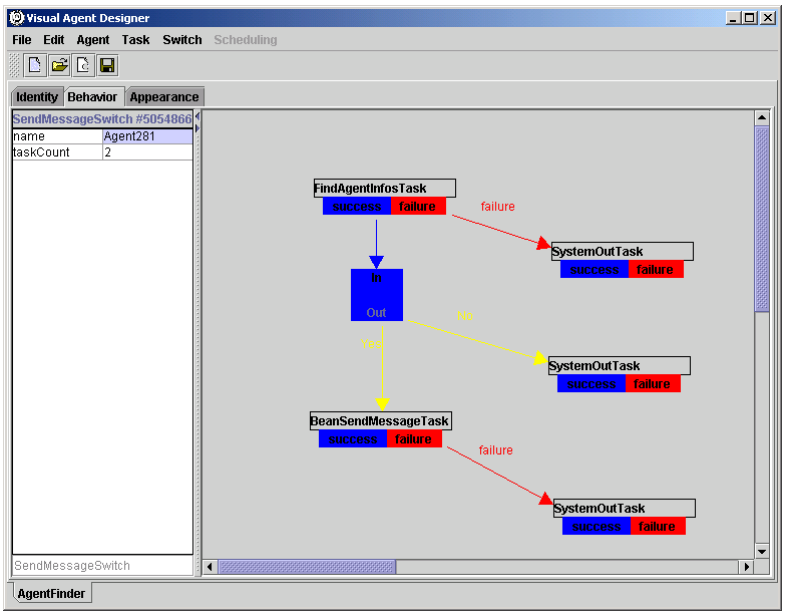
Clients of Tryllian are building applications and solving day-to-day problems that provide new facilities that would be hard to deliver without agent technology. So far an application for on-line diagnosis of telephone switching systems has been developed as well as an application to bring consumers and suppliers of merchandise together. Many more applications are conceivable and are under development.

Below you can find a number of the tools that allow a developer to create and manage agent applications.

1.10 VISUAL AGENT DESIGNER

The Visual Agent Designer (VAD) is a graphical interface for agent building. The VAD allows you to add pre-defined tasks with the aid of template behaviors or create them yourself so that you can create custom agents. Agents are directly generated from the VAD.

Because of the visual nature of the VAD it is not necessary to have any knowledge or experience with Java to create or modify simple agents. The whole process is done through the VAD interface relieving you of programming chores. However, if you want to create more intricate and developed agents, with added functionality and flexibility, you will also have that opportunity. You can make you own tasks available using Java and use the VAD as a high level behavior-modeling tool.



The Visual Agent Designer

1.11 BUILDING BLOCKS

The ADK provides basic mobile agent functionality and libraries. The kit itself does not address the needs of specific application types or domains. To this end separate building blocks are provided: specialized task libraries that can be used in the same way as the AFC task libraries. You can plug-in any number of building blocks and even define your own.

Currently building blocks are available for database connectivity, negotiation and workflow management. The *database building block* provides a generic JDBC interface and an agent language that can support SQL-queries. The *negotiation building block* provides a mechanism for agents to auction and trade according to specified negotiation policies. The *workflow building block* provides a generic framework to describe all kinds of workflow processes common in organizations and industry. Agents can use this to automate or monitor these processes. The workflow building block is specifically targeted for use in combination with the Visual Agent Designer.

1.12 UNIVERSAL HOME BASE

The Universal Home Base (UHB) is a separate application (not an agent in the habitat) providing a very user friendly view on a local habitat. The UHB gives an up-to-date view of the places and agents in the habitat. It is mainly aimed at end users, providing an aesthetically pleasing GUI loosely based upon Tryllian's previous agent application called Gossip.

The UHB allows the user and the agents in the habitat to easily interact at a basic level. The user can request some common tasks (like move to a room, move to the Tryllian server, die) from the agent using regular GUI methods like drag & drop and menu selections. Also, the user can create a random message in a dialog box and send that to an agent, and see the agents reply, and view the properties of an agent. The agent can request input from the user, which will result in (Gossip-like) balloon-menus or (Swing) forms popping up.

1.13 REMOTE HABITAT VIEWER

The remote habitat viewer is a more down-to-earth tool that allows you to view rooms and agents on both local and remote habitats. Not only will you be able to view these but you will also be able to interact with them. The habitat viewer lets you:

- View rooms and their properties.
- View the agents in a room and their properties.
- Send messages to you agents.

Security settings determine the level up to which you can view and modify these things.

1.14 MANAGEMENT TOOL

A management tool is provided to locally or remotely monitor and manage habitats. Whereas the remote habitat viewer focuses on keeping track of individual agents, the management tool is used for the systems management of one or more habitats.

The management console is the graphical interface between the habitat administrator and the management tool. It provides the administrator with all the information he needs in order to manage the habitat (viewing and editing), and for managing room and agent properties.

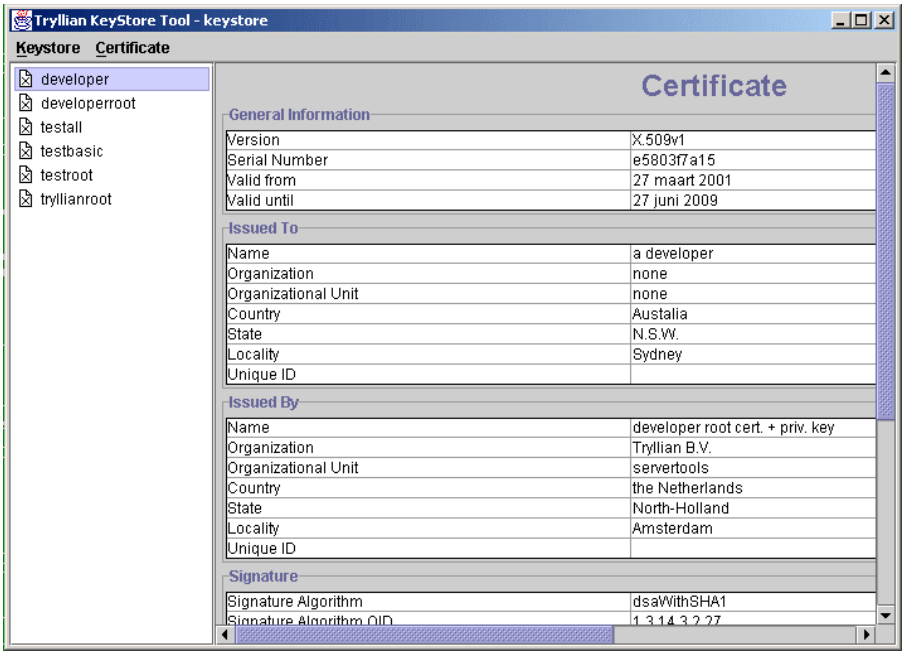
The management tool also comes with a text based command line interface. This allows one to automate the habitat management using shell scripts.

Monitoring

The management tool provides statistical information, keeping track of the traffic, number of agents and other pertinent information. The habitat administrator can also access and search through audit logs. The tool can send alerts on events like a high room occupancy, high amount of message traffic etc.

1.15 KEYSTORE TOOL

The keystore tool is used to create and manage security certificates. Refer to chapter 7, Security for detailed information on certificates.



The Keystore tool

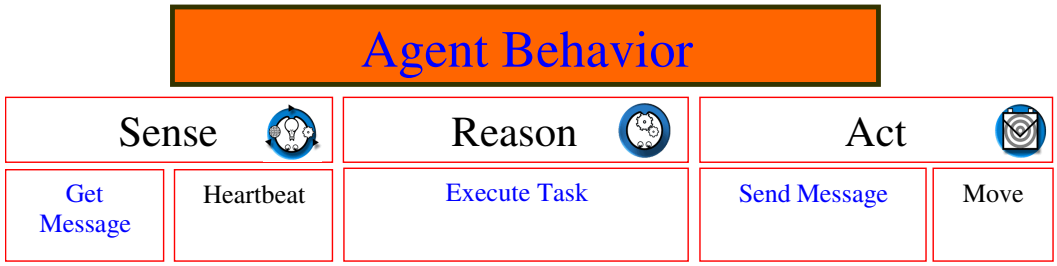
1.16 USING THIRD-PARTY DEVELOPMENT TOOLS

One can use any of the popular integrated Java development environments (JBuilder etc.) for editing, compiling and running agent applications. Details on specific configuration settings for these IDE's are provided in the ADK documentation.

4 Agent behavior

1.17 SENSE-REASON-ACT

The developer of an agent has the ability to create the behavior part of the agent, modeling the behavior on a sense-reason-act loop. This loop is a mechanism that models the interaction an intelligent entity has with a dynamic or (partly) unknown environment.



Sense: observe the environment and model it internally; Reason: update the internal state and determine a sensible action based on the state of the environment and the internal state; Act: carry out the proposed action

Agents receive stimuli from their environment and store these observations in their memory. If and how the agent reacts to these stimuli is decided during reasoning. As a result of reasoning the agent commits to one or more actions and plans them in his personal agenda. These actions are then executed during the act phase. Agent actions can implicitly or explicitly influence the agent environment, for example, by sending a message into the current room.

The implementation of the sense-reason-act mechanism has been done in such a way that it does not require you to be an expert in artificial intelligence to program an agent; knowledge of Java and some common sense should do:

- *Sense*. Receive an event. An event can be caused by an incoming message (causing reactive behavior) or by receiving a *heartbeat*. Heartbeats are the 'clock ticks' of the agent scheduler: they activate an agent's task once every few milliseconds and can be used to trigger proactive behavior.
- *Reason*. Internal processing of the agent. Reasoning is done by executing tasks linked to the sense events. Tasks can be predefined or user defined, either by composing other tasks or by Java code.
- *Act.* Cause events. An agent can act and influence its environment by sending messages to other agents or by moving itself to another room and/or habitat.

1.18 THE TASK MODEL

In some cases, tasks can be executed simultaneously, that is, in no particular order and with more than one at a time. While one task is being executed, another can be started, provided it is unrelated to the other task. If you build a price comparison agent, finding the cheapest price for a CD, you can query five different CD vendors in parallel.

In other cases, tasks must be executed in a defined order. Firstly, this means that some task cannot start before another is finished. Secondly, it means that the result of one task may influence which task is executed next. For example, your price comparison agent will only present the cheapest CD vendor once all vendors have stated their prices (or have failed to respond in time).

All of these considerations are dealt with by the task model that is used in the ADK. The task model describes the way in which the various tasks of an agent are to be executed by the task scheduler. It behaves differently from the Java code you're probably used to, and so it requires some explanation. In the following section, the motivation for the choice for this particular task model is explained. The subsequent section explains the implementation of the task model.

1.19 MOTIVATION OF THE TASK MODEL

The task model as we present it is based on three considerations:

1. Tasks should be executed independent of each other as much as possible. This is called *asynchronous processing*. The reasoning behind this is that tasks that don't need to wait for each other can be executed faster.
2. Tasks should be tailored to process messages easily. Virtually all behavior of agents is defined in terms of sending and receiving messages, rather than calling methods.
3. Tasks should be organizable in a robust way. This means that success or failure of a task can be detected and used as a condition for executing other tasks.

If you've ever built multi-threaded Java programs, possibly communicating by asynchronous messages, you know that this quickly grows very complex. The task model relieves you from all low-level implementation, coordination and housekeeping details, allowing you to focus on functionality instead of debugging.

Furthermore the task model enables you to encapsulate subtasks in a higher-level container task. This way functionality can be structured hierarchically. With the Visual Agent Designer one can link and compose an agent's tasks visually, using an UML-like representation. This way programming agents becomes much like defining a workflow.

1.20 IMPLEMENTATION OF THE TASK MODEL

The task scheduler keeps track of which tasks are active and listens for certain events. A task defines what to do with an event, depending on the state of the task. Each task has its own internal task state, which can be used to see if the task has started, is active or has finished. When the task finishes, its state will become either success or failure. These states can determine which task to execute next or to stop executing tasks altogether.

5 Agent communication

1.21 INTRODUCTION

Interactive collaboration between agents can make possible the sharing of costs and economies of scale ('distributed problem solving'). This is particularly useful in solving problems that require many varied and complex inputs, or computations that are divisible into numerous, independent steps. Examples: security monitoring, diagnosis, and maintenance of a distributed system, or query of a distributed database.

Communication is what makes your agents cooperate and interact. As with every form of communication, both sender and receiver have to use the same set of rules in order to understand what the other is saying to them. This set of rules is called a protocol. It defines what can be sent, how it is sent and what can be sent back in response.

But defining a protocol is not enough for full communication. For example, the protocol defines, that a question can be sent and that in response an answer is sent back. But what is asked and how the answer should be interpreted isn't defined by the protocol. This is defined by the language, which gives meaning to the messages within a certain context.

An agent can use multiple languages for communication in different contexts. For example, an agent can use a Trade language in communication with another agent, which has something to offer. After it has sealed the deal, it finds a Transporting agent and starts communicating with it in a Transporting language to organize delivery of the products.

1.22 THE FIPA PROTOCOL

There are a number of defined protocols agents can use to communicate, one of them is the *FIPA protocol*. FIPA is the standards organization for agent systems, see www.fipa.org. Agents built with the ADK adhere to this protocol. This protocol is based upon messages. Messages are sent back and forth between sender and receiver, and are of a specific *performative*, all of which are defined by the FIPA protocol. Example performantives are:

- *Request*. Ask the receiver to perform some action
- *Agree*. Tell the sender the request is granted
- *Refuse*. Tell the sender the request is not granted
- *Subscribe*. Request notification of state changes in some object of the receiver by the sender.

The FIPA protocol also defines which messages the receiver can send in response to a received message, for example a request can be replied with not-understood, refuse or agree.

Apart from a message performative, messages have a defined internal structure, containing all sorts of data. For example, who sent this message, for who it is intended, and why was it sent. This internal structure is defined by the FIPA protocol as well, in the form of message parameters. Some of these parameters are set automatically by the ARE/AFC when a message is sent, but others can be sent by the agent. These agent definable parts are used to implement an agent language. For non-trivial agents these parts are typically structured using XML.

1.23 MESSAGE TRANSPORT

The receiving agent for a message is specified using an agent address. An agent address is a Tryllian-specific, globally unique URL type. As the URL fully specifies the habitat, room and agent ID, this allowing addressing of agents in different habitats, on different platforms, too. A remote communicator built into every habitat takes care of such inter-habitat messages. Such messages are received just like intra-habitat messages - the receiving agent will only notice the difference if it bothers to check the sender's address.

1.24 COMMUNICATION WITH NON-AGENTS

Several approaches are possible for communication with non-agents. An agent can be written to provide a specific wrapper service. This agent can be given more privileges than the default ones if it needs access to resources that are normally prohibited such as files. Alternatively, it's possible to communicate more or less directly with the habitat from the outside by sending/receiving messages. Currently this can be done using the HTTP protocol, using a provided system-agent implementing this communication service.

6 Mobility

Mobility is one of the key aspects of Tryllian's agent technology. Mobility is not only the ability of the agents to be able to go and do requests or tasks for their owners, but also the ability to cross over different networks and platforms thereby. In order to create an agent that is able to move from one location to another one you need:

- *Code and state mobility*. When an agent travels to another habitat, it is not necessary for its code to be present on the other side. In that case the agent's code is automatically transferred together with the agent's state (data). If the code is already present just the state is transferred, keeping mobility cost efficient.
- *Code compatibility*. Two codebases (e.g. Java classfiles) can be different while caving the same name (e.g. newer versions). The ARE provides a mechanism for agents of different codebases to coexist. The advantage is that creating a new version of an agent does not require the modification of existing applications or infrastructures.

Code that is shared between multiple types of agents can be stored in a separate jar-file. The ARE can dynamically load the classes of an agent from multiple jar-files. A jar-caching mechanism in the ARE reduces the need for transferring code/jars.

7 Security

1.25 Introduction

This chapter contains an overview of the various aspects involved with agent security in the ADK. It will show how security is implemented and which mechanisms play a role in it. For the developer, security is reduced to signing his agent using the jarsigner tool provided with the Java Development Kit, but he should know how security works as a whole.

1.26 The big picture

In Java, the security is centered around the ClassLoader. A ClassLoader is responsible for loading the necessary classes at runtime. These classes sometimes provide functionality for accessing and modifying the system directly. For applications started by the owner of the system, this is normally desired, but for applications loaded from the Internet, like agents, this is not safe. To safely allow agents in your habitat, you have to define permissions and configure who will get which. In order to decide, who will get certain permissions and who will not, you have to determine where the agent originated. This is accomplished by including a certificate with the agent's jar files. With a certificate, you can check that nobody has tampered with the agent. You can also find out who created the agent and who this creator is trusted by. Assigning permissions to certificates allows the habitat to determine what an agent is allowed to do when it enters the habitat. The mechanism used by the ARE is similar to the one used by a browser running an applet.

1.27 Habitat permissions

On the lowest level, you have to decide which actions the ARE is allowed to perform. The developer usually isn't allowed to change this, since it is the task of the habitat manager.
What is important to understand, however, is the fact that the permissions of the ARE are not related to the permissions of agents. This is because the agents are loaded by the ARE with a different ClassLoader than the one used to load classes in the ARE. This is done in order to give the ARE more permissions than Agents will ever need,

without creating a security breach, and to completely separate agent classes from the ARE and from other agent classes.

1.28 TRUST CHAIN

The ARE keeps the information about certificates and their origins in the keystore file. When an agent requests to enter the habitat, the ARE inspect the agents jar files to see who created it and to check if nobody has changed the contents somehow. It does this by inspecting the certificate included in the jar files. This certificate contains the builder of the agent, his public key and an agent checksum. This checksum can be verified with the public key and the classes in the jar files. The checksum can only be created with the private key, which is only known to the builder. Although we can now check, that no one changed the jar files, this does not protect us against any malicious agent builder. Such a builder can create a private / public key pair on its own, and sign his harddisk-destroying agent without a problem.

To prevent this, we have to be able to check the integrity of the builder. A certificate issuer is a trusted authority. In order to become a trusted builder, the builder has to find an issuer who is willing to acknowledge his good intensions. The issuer does so, by signing the builder's owner and public key data with his private key. The resulting certificate checksum, together with the issuer data, is included in the certificate stored in the jar files of an agent. When a habitat owner decides to trust an issuer, it stores the name of the issuer and his public key, which is freely available, in his keystore file.

When an agent requests to enter a habitat, the ARE inspects the agents certificate to get the trust chain. It checks if the issuer at the lowest level is known in the keystore file. If he is, the ARE allows the agent into the habitat and gives it the permissions associated with the issuer. If he is not, the ARE checks the issuer at the next level. This process goes on until a match is found or no more issuers are left. In the case of no known issuer, the agent is denied access.

7.1 SECURE TRANSMISSION

All transmissions between habitats are encrypted using a secure socket layer. This protects both mobile agents and their messages.

8 Scalability and reliability issues

Tryllian has put significant effort in making the ADK an agent platform that can efficiently and reliably host tens of thousands of agents. Efficiency relates to use of system resources like memory, processor time and threads; reliability relates to the ability to handle peak loads and to recover from system crashes without loosing agents or their state. The main features are mentioned below.

8.1 SUSPENDING INACTIVE AGENTS

Agents that do not require any proactive behavior for a while, can hint the system that they can be suspended. An agent is suspended by storing it in a database and removing it from memory. Logically the agent is still present: other agents still 'see' it. But the agent does not claim memory or processor time anymore.
An agent is transparently reactivated when it is sent a message or when a predefined amount of time has elapsed.

8.2 BACKUP, SNAPSHOT AND RECOVERY

One can increase the reliability of an agent application by ‘persisting’ the agents and their state in a database. If the backup feature is enabled the habitat and room configurations and the agents and their states are stored in a database. Agents are back-upped each time they move, causing a very low performance hit and requiring no additional programming. In addition agents can request the habitat to back them up after an important event (for example completing a transaction). This is called snapshotting.

On restarting the habitat, be it after a proper shutdown procedure or a system crash, the habitat, rooms and agents are reconstructed from the database and carry on with their tasks.

8.3 THREAD USAGE

On most agent platforms the number of Java Virtual Machine threads grows as the number of agents grows. This seriously limits platform scalability, since large numbers of threads not only use a lot of resources but also cause stability problems on all major operating systems.
The ADK solves this problem by having the agents share a thread pool. One can set the amount of threads used by a habitat to any suitable number, in which one can take into consideration whether the server is shared by a number of applications or is dedicated, and the number of processors available.

9 For additional information contact

Tryllian
Joop Geesinkweg 701
1096 AZ Amsterdam
The Netherlands
tel +31 - 20 - 888 4060
fax +31 - 20 - 888 4326

info@tryllian.com

www.tryllian.com

Tryllian USA
1300 Clay Street
Suite 600
Oakland, CA 94612
tel +1 - 510 - 446 7776
fax +1 - 510 - 446 7775

info-usa@tryllian.com