

A Buddy Model of Security for Mobile Agent Communities Operating in Pervasive Scenarios

John Page, Arkady Zaslavsky, Maria Indrawan

School of Computer Science and Software Engineering
Monash University, Melbourne, Australia

{john.page,arkady.zaslavsky,maria.indrawan}@infotech.monash.edu.au

Abstract

This paper examines the security aspects of different pervasive scenarios involving agent communities evolved using multi agent systems (MAS). It describes the motivation and the objectives behind the development of these agent communities and analyses the security vulnerabilities, which arise within them. To counter these vulnerabilities, the paper proposes a Buddy model of security for the agent community. In this model, every agent protects its neighbour within the community, thereby sharing the responsibilities of the security function. This feature makes it a better option as compared to other hierarchical models of security, which can be brought down by a concerted attack at the controller agent. This paper also demonstrates the applicability and the effectiveness of the Buddy model in different pervasive scenarios and makes a strong case for its adoption.

Keywords: Mobile Agents (MA), Malicious, Community, Vulnerability, Security

1. Introduction

The advent of web based mobile services allowing users to remain connected to the network while on the move have gained in popularity leading to an increase in revenues from web based mobile applications. This trend has prompted many businesses to hop on to the mobile bandwagon in the hope of expanding their customer base. While this move is being seen as a positive trend and a bright future is envisioned for the advancement of pervasive technologies, it has also created pressure on the supporting infrastructure to keep pace with the growing number of users and applications. This aspect led developers to consider different ways of easing the pressure on the underlying infrastructure of the network while meeting the changing requirements. The agent paradigm appears to meet the challenges of a pervasive computing scenario and different viewpoints and observations regarding the paradigm are being made. One of the early distinctions between an agent and a program has been made by Franklin & Graesser (1996). According to them, an agent is an autonomous system, which continuously senses the immediate environment

while in pursuit of its own goals. An agent has certain identifying characteristics such as reactivity, flexibility, the ability to communicate and a learning nature. The advent of programming languages such as Java (Gosling & McGilton 1996) and TCL (Ousterhout 1994) extended the agent paradigm to include mobility. These languages have led to the development of several mobile agent systems such as Aglets (Lange & Oshima 1998), Grasshopper (Grasshopper Release 2,2 2001), Odyssey (General Magic, Odyssey n.d) and Voyager (Wheeler n.d) to name a few.

The advantages of disconnected computing, network load balancing and a fault tolerant and robust infrastructure, offered by the mobile agent paradigm, made it a popular destination for the development of mobile applications (Lange 1999). Encouraged by the user response to the agent paradigm, several research groups and universities have developed agent systems and several MAS toolkits are available. As a result, a diverse range of applications ranging from space exploration at NASA (Truskowski & Rouff n.d.), to the development of e-commerce scenarios has been met using agent systems. Other customized agent applications such as a JobFinder, BargainFinder, speech recognition agents and office assistant agents have been developed to meet the needs of an individual user (Cohen et al 1994, AgentBuilder Toolkit n.d.).

The last couple of years have seen the evolution of agent communities within which there is agent collaboration and cohesiveness. These agent communities developed using multi-agent systems (MAS), such as Concordia (Wong et al 1997) have successfully demonstrated their capabilities by seamlessly integrating with different applications and carrying out different tasks while working towards a common objective of the community. For example, in a web based airline-ticketing scenario, a travel agency could employ a community of agents to carry out ticket sales over the web. While the common objective of the community might be to maximise sales and to expand the customer base for the travel agency, there might be several sub-communities each with their own sub-tasks such as data gathering, information retrieval and filtering, handling of customer queries etc.

While these scenarios, highlight the ease of operation and the advantages of MA; they also expose the agent systems and the applications they support to different security risks. These security vulnerabilities are a big hindrance in the development of trust within and across agent communities and increase the overheads involved in the set up and maintenance of the applications. This paper analyses the vulnerabilities as faced by a mobile agent community and proposes a Buddy model of

community development, which allows the agents in the community to carry out their functions while providing protection to each other. The rest of the paper is as follows: Section 2 describes the setting up of an agent community involved in airlines ticketing over the web. It examines the various interactions, within and across the community members. Section 3 examines the different scenarios, which give rise to security vulnerabilities compromising the functioning of the agent and the agent community as a whole. Section 4 describes the implementation of the Buddy model of security for the agent community and details the advantages of using this model. Section 5 concludes the paper, with an indication of ongoing work.

2. A Mobile Agent Community Based Airline Ticketing Scenario

This section describes the setting up of an agent community to carry out the tasks of an airline ticketing travel agency. The first sub-section explains the various terms associated with a mobile agent agency, the second sub-section describes the different business functions of the travel agency carried out by the agent community.

2.1. Mobile Agent Community: Definitions

An *agent's authority* refers to the individual or the organization on behalf of which the agent executes (Milojicic et al 1994). It uses this authority to approach different servers, communicate, negotiate and enter into business deals with other agents. Agents moving about in

an agent space are identified using *agent identifiers*, which are assigned to them when they are created. These agent identifiers uniquely identify them and are valid for the agent's lifetime. For example, the agents created on a Grasshopper agent system have five attributes in the agent identifier. A sample Grasshopper agent address is `<Agent#130.194.67.210#2003-09-15#0.1>`. The first attribute is the prefix "agent". Other possible values of the first attribute are "service", "listener" or "unknown". The second attribute is the IP address at which the agent was created. The third is the date in "yyyy-mm-dd" format, the fourth is the time in "hh:mm:ss.msmsms" format and the last attribute is the number of clones of the agent. The current example shows the existence of one clone. An agent system or *Agency* refers to the platform which instantiates, transmits, interprets, executes and concludes the agent lifecycle. Similar, to the concept of an agent's authority, the server represents the organization, responsible for its operation. Each agent system is identified by an *agency identifier*, which is made up of a name and IP address. Figure 1 describes the various interactions that take place between the user, who controls the agent server, the agent server and its agents. It also depicts a virtual hierarchy of agent migration within an agent server. An agent's migration can occur between different Places. These Places could be a part of either the same agency or they can even belong to different Agencies. These Agencies could be set up on the same machine or exist on different machines, across networks. Agencies are registered as a part of a Region.

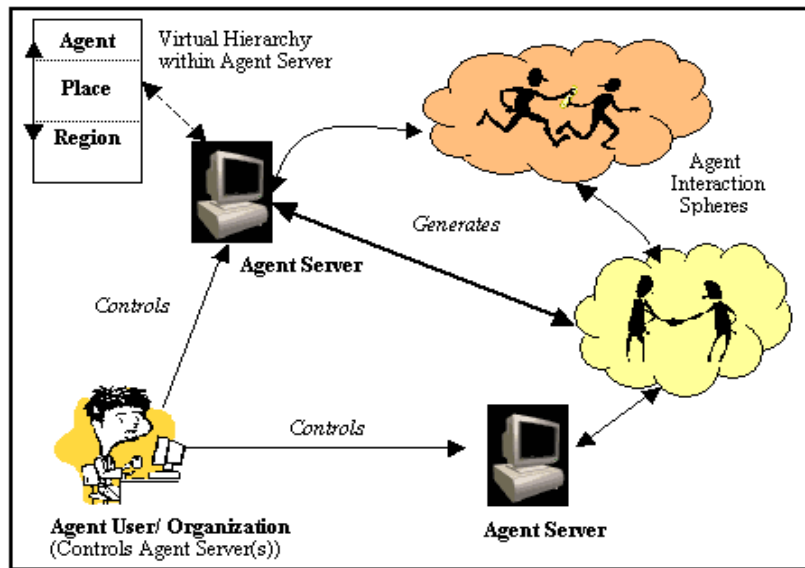


Figure.1. A Mobile Agent Operation Sphere

In an agent's context, a Place is a logical grouping of similar services available to an agent. The agent docks at a Place, to execute a particular function. In the ticketing example, explained in the next section, different Places offer different functions. For example, in an airline-ticketing scenario, some Places might offer tickets to particular destinations, while others might act as enquiry counters. An agent system, usually hosts more than one Place, while each Place is responsible for providing support and facilities to multiple agents. A Region is an aggregation of multiple agent systems or Agencies,

usually belonging to the same controlling organization or individual.

A mobile agent community evolves from the collective interactions of its members. The significant aspect of the community is that it will always have ONE goal that will motivate and drive its members to fulfil their individual tasks and functions. Within a community, it is possible to create sub-communities, each with their own goal. In the airline-ticketing scenario discussed in the next section, different sub-communities are created to carry out the sub-tasks within the community and work towards

achieving the community's goals. The community at the top of the hierarchy is referred to as the super community, depicted by the green boundary in figure 2. Within the super community, there are three different sub-communities each represented by a different colour. The goal of the super community is the primary goal for all agents belonging to that super community, while the goal of a sub-community is referred to as a secondary goal. As apparent from the figure, there are several secondary but only one primary goal. Agent communities can be,

closed, open or semi-open. A closed community does not permit any new members to join, while open communities allow new members. Semi-open communities only allow new members to join if current members of the community send out an invitation to join the community. Various communities follow different protocols in implementing this semi-open community model, for example, some communities may require that two or more members send an invitation before the agent is granted access and allowed to join the community.

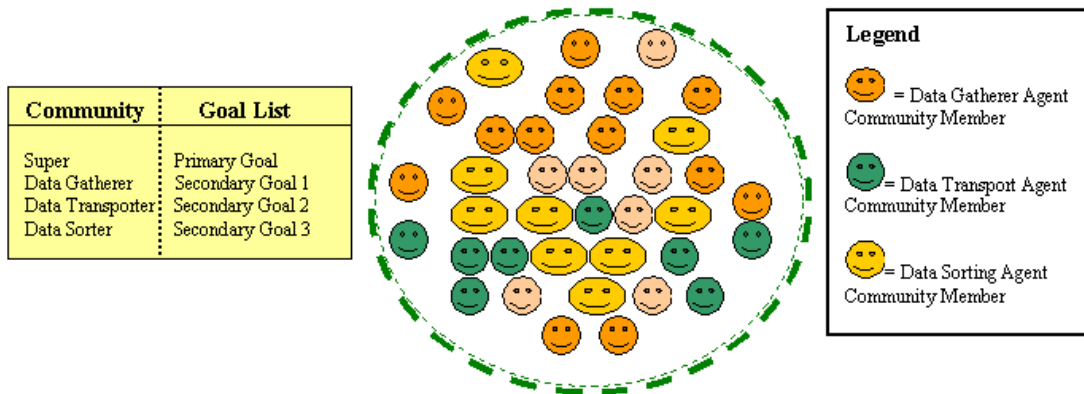


Figure.2. Agent Communities within a Community

2.2. An Airline Ticketing Scenario

A user intending to buy an airline ticket through the Internet is presented with an array of options to plan his travel in the best possible way. In the current pervasive world, an airline ticket is not just a booklet transporting a user from point a to point b, rather it has become more of a lifestyle guide. A glance at any airline ticketing web site, for example Orbitz <<http://www.orbitz.com/>>, allows the traveller access to various options of hotel room bookings, sightseeing tours, car hire options, cruise bookings along with his or her ticket. Travel agencies compete against each other to expand their customer base and their business turnover by offering travellers package deals. Regular travellers are awarded with discounted offers and encouraged to extend their business tours with a holiday added for a reduced cost. Such offers and options have made the web based airline ticketing industry, an intensely competitive and a complicated entity. This section, examines the various considerations, which influence the sale of airline tickets using a web based system. These considerations can be viewed from the customer as well as the airline perspective.

2.2.1 The Customer Perspective: Case 1

Airline travel being an expensive proposition, customers usually are interested in getting the best value for their money and will evaluate several choices before deciding on a particular airline or service. Consider Sam, a frequent flyer and a regular web user, uses the web to search for the best travel deals rather than relying on a human travel agent and paying a hefty commission to him. Sam's key considerations while making a travel booking in descending order of importance are as follows:

1. The travel itinerary should conform to his requirements.
2. The cost of the ticket should be reasonable and comparable with other rates.
3. The airline service offering the ticket should have a past record of reliability.
4. The airline service should be able to meet his special preferences, for example: A non-smoking flight is desired or a special dietary requirement exists.

Various permutations and combinations of these factors and past experience influence Sam in his decision-making. In most cases he will try to choose a solution that is familiar to him. For example, if Sam has travelled via airline X from point A to B via C and his travel was immensely satisfactory, he will remember the pleasant experience and in all future journeys, Sam will search for a flight route with airline X because he is confident that the airline will provide him with an expected standard of service. In such cases, it will be difficult for an airline Y to influence Sam to fly with them unless they are able to offer him a lucrative incentive or in some way, they are able to destroy Sam's confidence in airline X.

2.2.2 The Airline Perspective: Case 2

The airline perspective being capital driven has several more aspects to it than the customer's perspective. To enable more customers to access to its ticketing function, airlines either directly or via the medium of booking agents and / or travel agencies set up various interfaces, which allow the prospective traveller to choose from a variety of travel options. Most airlines realize that a significant amount of clientele is available on the web and ticketing via the web-based interface has become a common and in many cases, the cheapest option of booking an air ticket. The next challenge was to reach

mobile users with business deals and offer them an interface of logging into the system and making a booking. The use of mobile agent based systems has provided ticketing agencies with the chance of reaching a large section of mobile users. These users now have the option of making ticket bookings and executing queries to the booking system while on the move.

Various complexities define the business dynamics of the airline travel industry. The major considerations faced by the industry in its operations are as follows:

1. To increase the volume of business.
2. To generate a level of confidence and trust with existing customers and to forge long-term relationships.
3. To provide an easy to use client interface medium.
4. To effectively manage the business infrastructure and to ensure that there are minimum downtime periods.
5. To analyse and explore other possible avenues for expanding the business.

These considerations shape the business goals of the airlines. The travel agencies, which represent the airlines in the market, are educated and motivated towards achieving these goals. While these factors are the major elements in defining the business strategy, for the company, they are also the main points of attack for a malicious agency such as a rival airline organization.

2.2.3 A Mobile Agent Based Travel Agency

All Grasshopper MAS specific terms used in this section have been defined in Section 2.1. Consider a travel agency, which uses the Grasshopper mobile agent system to provide an interface to its mobile clientele to make airline ticket bookings. This interface is also used for keeping a watch on the industry trends by browsing through various deals offered by other travel agencies and airlines on the web. These services are implemented using agent communities, an example of which was depicted in figure 2.

In this paper, the airline travel agency interface is available as a Grasshopper region called *SeeTheWorldAgency*. The region awards a ticket to each new agency, registering with it. In the current scenario, the region hosts two agencies, namely *Surya* and *Piracles*. These agencies each host a number of different places wherein, different travel options are available to approaching agents. The communication service between the agencies is implemented using a socket layer. The region keeps track of the agent movements and registers each agent arrival. These agents are sent from a number of sources, such as other travel agencies, airline offices, and existing as well as potential customers. Some of the approaching agents can also be agents returning to their home base. This agent migration creates a complex scenario of interactions, which are handled by the receiving agency, according to the information carried by the agent. Authentication of agents is simplified, using a context-based authentication layer (Page et al 2003). The agency examines the information provided by the agent and decides on the processing steps for the agent. For example, certain agents can request information regarding travel to a particular destination while other agents may approach the agent server to make or cancel a booking. In

the current implementation scenario, the agent system creates a community of agents. The primary goal of the community is to maximise the business of the agent system. Within this super community, sub communities based on specific functions of the agent exist. For example, information searching agents, ticket-booking agents, query-handling agents, advertising agents etc each form their own communities. Each of these sub-communities will have their community specific goals, which in terms of the super community will rank as a secondary goal. Agents in these sub-communities could employ auxiliary agents to assist them in their daily tasks. For example, advertising agents used to advertise the current schemes and services on offer might require a list of addresses to post their advertisements to. An auxiliary agent could provide this list to the advertising agent community. Similarly, many such interactions exist and occur within an agent space, which contribute towards the successful realisation of the super community goal.

3. Security Vulnerabilities Within An Agent Community

A mobile agent system is an open entity with agents regularly arriving from different sources. In such a dynamic scenario, there is always the possibility of agents assuming a malicious nature and attempting to damage the agent server. On the other hand, agents in pursuit of their goals migrate and wander in and out non-trusted domains. It is possible that on such forays, a malicious agent server might capture and subvert an agent forcing it to attack its own parent server. This section examines the security vulnerabilities to a mobile agent space at three different levels:

1. The agent level
2. The agent system level
3. The agent community level

Apart from these three levels there exists a minor fourth dimension involving entities external to the agent space. For example, a virus may attack the agent system or the agent code base. These attacks are tracked by organizations such as the Computer Emergency Response Team (CERT), at the Carnegie Mellon University and the Federal Computer Incident Response Team (Fed CIRC).

3.1. The Agent Level

The agent level is the most vulnerable part of the mobile agent space. It is susceptible to malicious attacks from rival agents as well as hostile agent servers. The object of these malicious attacks is to steal, capture or destroy the information carried by the agent. An agent could attack another agent for the purpose of disrupting its function and gaining access to the information carried, as well as the credentials of the agent. This allows the malicious agent to masquerade as another agent while visiting new servers. Thus when any malicious activity is detected the agent, whose identity was stolen, is suspected for the resulting consequences. Malicious agents can also block the transit and communication channel of other agents by spamming it or by intercepting messages from its parent server. The intention behind these malicious attacks is to destroy the viability of the agent's function. Another

implication of these malicious attacks is the commercial impact. At certain servers, agents are charged for the number of CPU cycles they use. If an agent is forced to process Spam, while docked at a server, it will be paying for using resources, which it did not require.

An agent's body is composed of two parts, data and code. Both these parts are targeted by malicious entities while attempting to capture the agent. In the airline-ticketing scenario described in the previous section, Sam requires a ticket to fly from Melbourne to New York on the 17th of October. Sam's agents have been fed this information and have been sent out into the web to search for the best matching choices and available options. Since the agent's have limited data space, they cannot be expected to bring back all the various options they encounter. Certain considerations such as 'finding the cheapest ticket' and 'hotel package deals' act as filters in the information search. The agent user feeds the search criterion to the agent and signs it using a digital signature to prevent unauthorised tampering. The search function of a customer agent searching for an airline ticket is similar to the pseudo code given below:

The variable *Choice_List* is a list of choices viewed by the agent and stored for further action by the agent user. This list is ordered on the basis of prices in descending order i.e. most expensive option is stored first. In this example the list has a limit of holding ten options.

*/*The search operation for a client */*

1. **Find** " Airline Ticket Booking "&& " Status = Available"&& "Departure=MEL"&& "Arrival= NY"&& "Date=17/10/03"
2. *If (found) Check (Choice_List)*
3. *If (Choice_List is empty)*
4. *Add new_option at the top*
5. *else (Compare available price option with the first entry in Choice List)*
6. *If (Cheaper && Current Entries > 10)*
7. *Insert new_option at appropriate position*
8. *else {Delete first entry; Insert new_option at appropriate position}*

From this example, it is clear that the agent is involved in two major operations while on a mission. They are *Find* and *Compare*. These two operations form the core of the agent search function. Malicious entities targeting data will direct their attack at the *Choice_List* carried by the agent and attempt to tamper with the information held there. Since agents cannot dynamically sign the information they receive they have to rely upon the signing capabilities of the visited agent server to protect the information. This dependency creates holes in the agent's defence mechanism. Malicious agent systems can view the information carried by the agent, along with all past servers visited and the next server to be visited. This is possible because, the agent server at which the agent is currently docked has access to the agent's code and business logic. The agent is in a classic catch-22 situation (Kirkland et al n.d). On one hand the agent is forced to reveal its code to the server in order to gain its trust and be allowed permission to execute and get access to

resources. On the other hand, if it does not allow the server to examine its code, the server might consider it to be a malicious entity intending to take over the server and destroy system resources. Due of this reason, the server might not give it permission to execute. Allowing the server permission to view the agent's code can have several implications. One of the ways of attempting a malicious attack is to change the information carried by the agent and then signing it with a forged signature of the previous signer. Another possibility of a malicious action is when two servers join up to fool the agent. This scenario is possible because the current agent server is aware of the next server in-line, to be visited by the agent. Since the current server has viewed the contents of the *Choice_List* and is aware of the business logic of the agent, it can signal the next server in line to offer a ticket price lower than the highest price in the current list. This will guarantee the agent picking up information from the server even though; it might be untrue. To accommodate this information the agent could be forced to drop a genuine record. In this attack scenario, the focus of the malicious agent server's attack was the agent data. In another possible attack scenario, the agent server could attempt to tamper with the agent code and change its business logic. For example, in the search function discussed earlier, if the parameter "cheaper" occurring in the instruction on line 7, is made "expensive" the entire agent operation is rendered useless, as the agent will return to Sam, with confusing data. From these scenarios, it is clear that an agent can be attacked in various ways. The nature of these attacks can be either concealed or open (Page et al 2003) but in both cases they target either the agent code or its data and attempt to destroy the viability of the agent operation.

3.2. The Agent System Level

An agent system has to essay the role of a host to agents from different sources. These agents request server resources in order to carry out their tasks. The agent server analyses and prioritises these agent requests and sanctions the requested resources based on its policy file. Different policies are responsible for influencing the server's decision on whether or not to grant the requested resource to the agent. Qusay (2000) has given examples of various applications including mobile agent servers using policy files and engines. He has also has described a possible attack by a malicious entity attempting to delete a sensitive server file. To thwart such attacks, the author has proposed the implementation of a security engine, which extends the Java *SecurityManager* class. While such an approach might protect the agent server from malicious attempts of file deletion, it might also prevent a genuine file deletion request. For example, a customer wishing to delete his profile on an agent server could send an agent with a delete request. Such a request might be denied owing to the additional level of security imposed in the system.

While the agent system level security policies are more stringent as compared to those seen at the agent level, the agent system is protected by the array of resources at its command. For example, in an agent system such as Aglets (Lange & Oshima 1998), several layers of security protect the server from malicious access. These layers are

provided by the supporting software, in this case, the operating system, the Java security implementation and the execution environment, i.e. the Aglets workbench. In many cases these layers are not sufficient to protect the agent system from a malicious denial of service attack as shown in Page et al (2003). The following pseudo-code lists the agent server action on an agent docking.

```

/* Sever Action on Agent Arrival */
1. Authenticate the approaching agent on docking.
2. If Agent Authentication is Successful
3. Check Agent Authorization /*This step establishes the extent of resources the agent can access*/
4. If Agent Authorization is Valid
5. Check agent resource request list
6. If resource(s) are available
7. Grant resource(s) /* Resources can also imply Information requested by the agent*/
8. Else {Put agent request in queue
9. Set agent status to SUSPEND}
10. Else Disallow agent from accessing system resources
11. Else Disallow agent from docking on to server

```

At the agent server level there are two main security related functions that the server executes on an agent. These are authentication and authorization of the approaching agent. Authentication refers to establishing the agent bona fides of the agent while authorization refers to detailing the scope of agent rights on system resources (Farmer et al 1996). Both these functions are responsible for defining the scope of operation of the agent within the agent space and are important from the agent as well as the agent server point of view. Malicious agents attempt to bypass these two functions and gain access to the inner sanctum of the server. If they are successfully able to do so, they can damage system files, steal information and in some cases even take control of the server. These attacks manifest themselves in several ways; the most common symptoms of an attack are a general degradation in service and unexplained movement of data from the node. A successful malicious attack on an agent server can have financial implications for the organization controlling the server. A far more major and sensitive issue is the loss in client confidence and future business when prospective customers learn about the attack. This is the reason why organizations tend to keep security breaches a secret rather than publicising it. The disadvantage of this attitude is that malicious entities get a free hand in implementing their attacks on other agent servers with impunity causing extensive damage for the entire industry. If agent system users were more open about the security breaches and made the attacks made on their servers public, the community could be educated and armed to protect itself against such attacks reoccurring.

3.3. The Agent Community Level

While identifying the boundaries of an agent community might not be a simple task, it can be roughly estimated from the collaboration that happens between agents. Collaboration and cohesiveness in achieving the same

goal are major identifiers of an agent community. While the evolution and use of agent communities have significant advantages for agent users, they also increase the administrative workload of the agent system, as community level interactions also have to be taken into account and monitored. These interactions can be either a simple data passing method or communication between agents. It is possible that there are different levels of control within the community and at each different level; agents have different access rights with respect to server resources. In such a volatile scenario, with agents regularly joining and leaving the community, the possibility of malicious agents assuming the identity of a controller agent and entering the community becomes high. If a rogue agent is able to bypass the security policies and enter the community, it can manipulate and even terminate other agents since it is possible for an agent to set the state of another agent, as evident in the Grasshopper MAS. The following pseudo code explains the working of a community of mobile agents, from the perspective of a single agent within the community.

```

/* Agent Function Within a Community */
1. Identify other community members and their authority levels
2. Identify the central theme or the goal of the community
3. Identify allocated tasks
4. Create and Open a communication channel(s) with other community members
5. Identify and Forge collaborations with other agents
6. Assign and withdraw rights from other agents

```

From the sequence of steps given above, it is clear that an agent is faced with several considerations when it operates at a community level, as it has to interact with other agents. This interaction is based on collaboration, which stems from working towards the same goals. To keep track of the progress of each other and of the community as a whole agents communicate with each other using various methods and protocols. In our experimentation, with the Grasshopper MAS, socket communication was used to create the communication channel. Several security loopholes and vulnerabilities can arise in a community level interaction scenario. Since the community space is a dynamic entity in itself, it is difficult to monitor and exercise complete control over all members simultaneously. Rogue agents take advantage of this factor enter the community and create confusion by assuming the role of a community supervisor and exercising the power invested in the role. Once inside, they can start sending out misleading instructions to other agents, block the communication channels and incorrectly set agent-working status. For example if an agent is waiting for the server to give it certain information, the supervisor can set its status to SUSPEND. When the information becomes available the agent can be re-activated. To create an imbalance in the community, rogue agents can incorrectly change the status of agents. Thus, the rogue agent acting from within a community can execute a hostile behaviour affecting the health of the

entire agent community. These vulnerabilities arise due to two reasons. The first reason being that rogue agent was able to bypass the community authentication function and enter it. The second reason being that the community was implemented on a hierarchical model, with some agents wielding more control and power than others. This allowed the agent to assume the identity of a community supervisor and issue misleading instructions to other members. In the security solution proposed and explained in the next section, each agent within a community has the same *access rights* with respect to *agent control* and *communication*.

4. Security Solutions For An Agent Community

Agent communities being dynamic spaces, the security solution for an agent community requires to reflect this aspect and protect the community from vulnerabilities, which crop up due to changing conditions within the community.

4.1. Security Goals for the Agent Community

To have a complete and uniform coverage, the security schema should attempt to meet the following goals:

1. *Flexibility*: It should be possible to configure the schema according to changing conditions within the community. For example: Within an advertising agent community, agents may receive communiqués from other communities but after the promotion is over this facility may become a security loophole and may need to be withdrawn. The solution should be flexible enough to allow this configurability.
2. *Coverage*: The solution should cover the entire community rather than key members as partial solutions leave loopholes for malicious attackers to exploit. Protecting a particular member is disadvantageous in two ways. Firstly, it allows the attacker to study the construction of the community and to gain insight regarding the working of the community and secondly, it allows the malicious attacker to find weaknesses in the model by targeting non-protected members.
3. *Manageability*: The security solution should have ease of management and implementation. In other words it should be possible to view the status and the health of the community using a single view. The migration of community members across the community should be reflected within the security view and it should be possible to track their movement.
4. *Non-Hierarchical Implementation*: Even though several models will attempt to contradict this thought, our study has seen that hierarchical models are more susceptible to malicious attacks. These models are easier to penetrate inspite of additional layers of security that may be applied. For example, in a hierarchical model a specific agent executes security tasks. This aspect is similar to the Java SecurityManager class that is responsible for the implementation of the security features of Java. The disadvantage of this approach is that the effectiveness of the security implementation becomes dependent on the performance of that particular agent or group of

security agents. To implement a uniform strength security coverage solution, each agent of the community should contribute towards protecting itself and the community as a whole in a self-reliant manner.

5. *Risk Management*: While no security schema can guarantee complete protection, proposed solutions should be able to reduce the existing threat and bring it down to a manageable level.

4.2. The Security Solution: The Buddy Model

Each time an agent departs on a mission, it exposes itself to capture and subversion. Agents moving to only one platform before returning to their home base face a limited amount of risk and are classified as a one-hop threat (Jansen 2000). Agents travelling via more than one destination, before returning to the home base, classify as a multi-hop threat and are more vulnerable to being attacked. Since an agent is dependent on its host platform for its protection and security while in transit. These factors assume a complicated dimension within the agent space.

To protect a mobile community of agents, the Buddy schema is an innovative and effective solution. In this security model, agents protect their neighbours, which are their Buddies. As all agents within the model are constructed identically, it is difficult for a potential attacker to target a central agent. This model provides the agent community user, the flexibility to hide any critical piece of data within a single agent or break up the information and secrete it in pieces among the community members. The agent community forms various topologies while in transit for example, a star or ring formation is common. This allows it to add or reduce the number of members attached to the community without disrupting the community travel. Following a certain topology allows the agents to be aware of the other's position, and call for help, when attacked. This is similar to agents recording the itinerary (Roth 1998) of each other with some essential differences. In this model, each agent 'senses' the presence of its nearest neighbor by issuing or receiving a token; they do not record their actual locations.

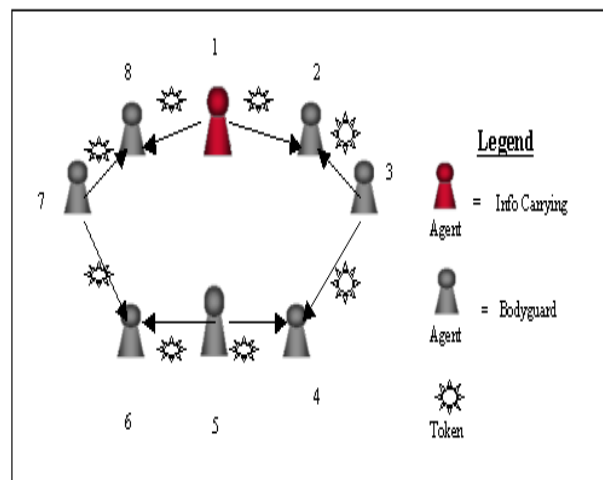


Figure.3. Token Passing in the Buddy Model

Figure 3 diagrammatically explains the Buddy model of agent communities. If this token is received, within a certain timeframe the agent is assured that its Buddy is in good health. If the token is not received, it will indicate to the agent that its Buddy is unable to communicate. The agent will then do a global broadcast and inform the other agents to investigate the problem. For the token broadcasting mechanism to be efficient, the chance to broadcast is given to each agent in the ring.

In the scenario depicted by figure 3, there is a group of eight agents as belonging to the same community. In the first half cycle, agents at odd locations, in the topology i.e. 1,3,5,7, are currently broadcasting the token while agents at even locations i.e. 2,4,6,8 will sense. In the next half cycle, this will be reversed, i.e. the odd group will sense while the even group will broadcast. The current scenario has considered a group of eight agents just for the reason of making the model simple to understand. There are no limitations on the number of agents in the topology however; a consideration does exist that the total number should be an even. This model can be further understood using the pseudo code given below:

```

/* Token Passing in the Buddy Model */
Phase A: Assigning Groups & Identities
1. Assign agents in even locations to Group A
2. In Group A, assign identities as ma1, ma2, ma3, ma4
3. Assign agents in odd locations to Group B
4. In Group B assign identities as mb1, mb2, mb3, mb4
Phase B: Broadcasting and Sensing
(The First Half Cycle)
5. ma1 broadcasts locally : mb1, mb4 sense.
6. ma2 broadcasts locally : mb1, mb2 sense.
7. ma3 broadcasts locally : mb3, mb2 sense.
8. ma4 broadcasts locally : mb4, mb3 sense
Phase C: Token Verification
9. Check for Token Receipt:
10. If token is received, move to the indicated location
11. Reverse Broadcast and Sensing cycle

```

4.3. Implementation Of The Buddy Model

The Buddy model is implemented using the Grasshopper MAS version 2.2.4b running on an Intel P4, 512 MB RAM, Windows 2000 machine. In the scenario shown in figure 3, an agent issues a token to its immediate neighbors. In the model there are two main players, the token sender agent and the token receiver agent, which is referred to as a Buddy for the previous agent. Both these agents uses two Java classes namely the *TokenReceiver* class and the *TokenSender* class as shown in Figure 4. The *TokenSender* class creates the token and sends it to the *TokenReceiver* class. The token is a Grasshopper agent address to which an agent can migrate. In the mobile agent community, the agent implementing the *TokenSender* class keeps track of its Buddy by constructing an agent proxy. This proxy communicates with the agent system through the agent system *region* and locates the Buddy for the token sending agent. Communication between agents is carried out using a socket layer. The *TokenSender* class has two methods, which are

different from the default agent functions. They are *getName ()* and *requestLocation ()*. The first method returns the name of the agent to the agent system while the second method is used to form the next location for the Buddy agent. In our current implementation a popup appears on the screen and the user can input an agent location. This agent location is used to form a token and is passed on to the Buddy agent Other agent classes have been left out from the class diagram of figure 4, in order to maintain readability.

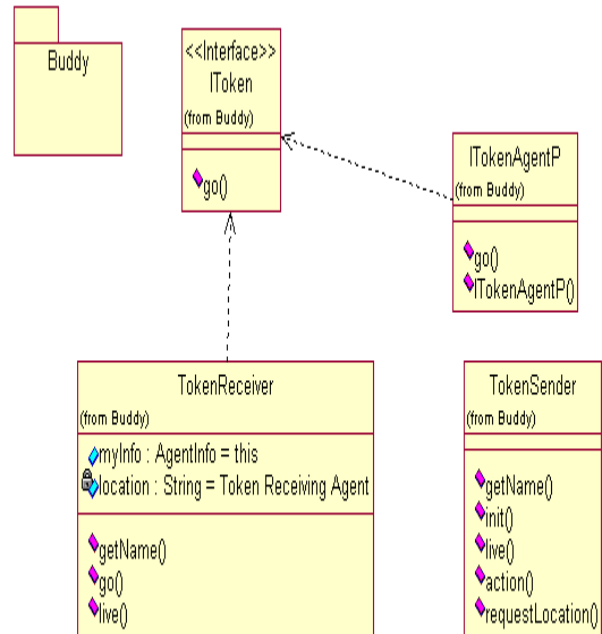


Figure.4. Class Diagram of the Buddy Model

In this particular scenario, the time out for the Buddy agent i.e. the agent waiting for the token has been set to five minutes. If the Buddy agent does not receive a token within the time period, it broadcasts a general alarm and moves back to its homebase. If the token is received the Buddy moves to the address given by the token completing the first half of the cycle. In the next half cycle, the roles of the two agents are reversed. The receiver becomes the sender and vice-versa. In the model described in figure 3, each agent is shown to be emitting a token to its immediate neighbours. In the example taken two tokens are being emitted as each agent has two neighbours. However, this number is not a limitation as an agent can send the token to a larger number of agents, provided they are located as neighbours for the agent within the community topology.

5. Conclusion

This paper analysed the security vulnerabilities of mobile agent communities operating in pervasive environments. It described the operation of a pervasive environment using a mobile agent based interface of an airline ticketing travel agency. Different vulnerabilities of the agent space were examined from the agent, the agent server and the agent community levels. To counter the security vulnerabilities, a Buddy model of agent travel was proposed for mobile agent communities. This model is a better option as compared to other hierarchical

models of security as in this; there is no central controller. All agents within the community share the responsibility of protecting each other and hence the model is referred to as the Buddy model. Since there is no central controller within the model, it makes it a difficult target for an attacker. Another advantage of this model is that sensitive information can be collectively carried by agents while in transit or can be hidden within a single agent. If the first case is followed, the attacker has to capture all members of the community before it can piece the data together. In the second case, the attacker has to accurately guess the information-carrying agent, a difficult task as all agents have the same construction and exhibit similar behaviour. Thus, this model has several advantages over the traditional unstructured modes of agent community migration and is successful in providing a security cover for each member of the mobile agent community.

References

- 'Agent Builder Toolkit', URL <http://www.agentbuilder.com/Documentation/brochures/ABProBrochure.pdf>, Date of Access 10/07/2003
- 'General Magic, Odyssey'. URL: <http://www.genmagic.com/technology/odyssey.html>, Date of Access 03/05/2003
- 'Grasshopper Release 2.2'. 2001, Basics and Concepts Revision 1.0, URL <http://www.grasshopper.de/download/doc/BasicsAndConcepts2.2.pdf>, pages 70, Date of Access 03/12/2002
- 'Orbitz: Airline Tickets, Hotels, Car Rentals, Travel Deals', URL <http://www.orbitz.com/>, Date of Access 02/09/2003
- Cohen, P.R., Cheyer, A. J., Wang, M., & Baeg, S.C. 1994, 'An Open Agent Architecture', In Proceedings of AAAI Spring Symposium, pp. 1-8.
- Farmer, W. M., Guttman, J. D. & Swarup, V. 1996, 'Security for Mobile Agents: Authentication and State Appraisal', In Proceedings of the European Symposium on Research in Computer Security (ESORICS), pp 118-130.
- Franklin, S. & Graesser, A. 1996. 'Is it an Agent, or just a Program? : A Taxonomy for Autonomous Agents', In Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages, Springer-Verlag.
- Gosling, J & McGilton, H. 1996, 'The Java Language Environment: A White Paper', Sun Microsystems.
- Jansen, W. 2000, 'Countermeasures for Mobile Agent Security' Computer Communications, Special Issue on Advances in Research and Application of Network Security, Elsevier Science BV.
- Kirkland, A., Lee, J., Stephens, D. & White, M. 'CATCH-22 by Joseph Heller: An analysis and interpretation', URL <http://www.beaconlc.org/ctech/stuwork/CATCH22.HTM>, Date of Access 04/09/2003
- Lange, D. & Oshima, M. 1998, Programming and Deploying Java Mobile Agents with Aglets, Addison-Wesley.
- Lange, D. 1999, 'Seven Good Reasons for Using Mobile Agents', In Communication of ACM, vol. 42, no. 3, pp. 88-89.
- Milojicic, D et. al. 1998, 'MASIF The OMG Mobile Agent System Interoperability Facility', In Proceedings of Second International Workshop, LNCS No 1477.
- Ousterhout, J. K. 1994, TCL and the TK ToolKit. Addison-Wesley. Reading MA.
- Page, J., Zaslavsky, A., & Indrawan, M. 2003, 'Security Aspects of Software Agents in Pervasive Information Systems', In Proceedings of the 14th Australasian Conference on Information Systems (ACIS 2003), To appear.
- Page, J., Zaslavsky, A., & Indrawan, M. 2003, 'Evaluating Security in Software Agent Systems using a Security Analysis Tool', In Proceedings of the 1st Australian Information Security Management Conference (InfoSec 2003). To appear.
- Qusay, M. 2000, 'Security Policy: A Design Pattern for Mobile Java Code', In Proceedings of the Seventh PLOP Conference Washington University Technical Report number: wucs-00-29. URL <http://jerry.cs.uiuc.edu/~plop/plop2k/proceedings/Mahmoud/Mahmoud.pdf>, Date of Access 12/03/2003
- Roth, V. 1998, 'Secure Recording of Itineraries Through Cooperating Agents', In Proceedings of the ECOOP Workshop on Distributed Object Security and 4th Workshop on Mobile Object Systems: Secure Internet Mobile Computations', pp. 147-154.
- Truskowski, W. & Rouff, C. 'New Agent Architecture for Evaluation in Goddard's Agent Concepts Testbed', Date of Access 12/09/2003, URL <http://agents.gsfc.nasa.gov/papers/pdf/aap41.pdf>
- Wheeler, T. 'Developing Peer Applications With Voyager. A white paper' URL <http://www.recursionsw.com/products/voyager/whitepapers/Developing%20PeerApplications%20With%20Voyager.pdf>, Date of Access 03/08/2003
- Wong, D., Paciorek, N., Walsh, T., DiCelie, J., Young, M. & Peet, B. 1997, 'Concordia: An infrastructure for collaborating mobile agents', In Proceedings of the First International Workshop on Mobile Agents, Germany, LNCS 1219.