Implementation of Secure Mobile Agent for Ad-Hoc Networks

Khaled E. A. Negm
Department of Computer Engineering
Etisalat College of Engineering
Sharjah, POB 980
UAE

Abstract: - Security is an important issue for the widespread deployment of applications based on software agent technology. It is generally agreed that without the proper countermeasures in place, use of agent-based applications will be severely impeded. However, not all applications require the same set of countermeasures, nor can they depend entirely on the agent system to provide them. Instead, countermeasures are applied commensurate with the anticipated threat profile and intended security objectives for the application. Such countermeasures can be integrated directly into an agent system, or incorporated into the design of an agent to supplement the capabilities of an underlying agent system.

Key-Words: - Mobile Agent Security, Secure Ad-Hoc Networks, Secure M-Commerce

1 Introduction

Agents are independent pieces of software capable of acting autonomously in response to input from their environment. Agents can be of differing abilities, but typically possess the required functionality to fulfil their design objectives. To be described as ‘intelligent’, software agents should also have the ability of acting autonomously. Agents are, to various degrees, aware of their environment, which often also can be affected by the agents’ actions.

A mobile agent is a particular class of agent with the ability during execution to migrate from one host to another where it can resume its execution. It has been suggested that mobile agent technology, amongst other things, can help to reduce network traffic and to overcome network latencies [1]. An agent’s ability to move does however introduce significant security concerns.

Autonomous agents and multi-agent systems represent a relatively new way of analysing, designing, and implementing complex software systems. In this article we are only concerned with the security of the system and its components (leaving design methodologies to others). Several multi-agent systems are available as commercial products and many more have been implemented in various research projects, with varying success. Recent standardisation efforts [2-5] have proven rather successful and are still evolving. Today there is growing interest and research in implementing and rolling out (open) multi-agent systems on a wider scale. Mobile VCE (www.mobilevce.com) is undertaking one such project where the agent paradigm is researched in a mobile telecommunications setting.

In this paper we present an implementation of a secure mobile agent the framework that was introduced in previous research [6-8]. The proposed system is built up based on the Master-Slave pattern introduced by Buschmann in 1996 to support fault tolerance, parallel computation and computational accuracy [9]. Lange demonstrates that it is also applicable to support tasks at remote destinations and extended it to fit mobile agents [10].

The framework consists of a coordinating entity (the supervisor) and several independent entities (the workers). The workers move from host to host and try to
finish the tasks they got from the supervisor. The supervisor holds all the current knowledge found by the workers and uses this knowledge to accomplish its task. If it needs more information it creates a new worker and sends it to places to get this information. The key difference to the client-server paradigm is that the supervisor component is mobile as well. So it can move to a host near the area its workers will operate in. The only prerequisite is that the supervisor must exclusively visit secure trusted places. In the worst case this is the computer where it has been initialized. We have demonstrated that this framework solves special aspects of mobile agent security. We showed that eavesdropping information and tampering the agent is no longer possible or does not reveal any confidential information.

2. Constraint Manager Components

The main component of the Framework is the constraint manager. It stores and administrates the constraint database. Its constraints can be modified on-the-fly (i.e., the constraint database may be modified while the agent is traveling) or supplied by the user.

The constraint manager component provides mechanisms to edit constraints. Especially it enables adding and deleting constraints on-the-fly. It is possible to group constraints at least by task and by constraint type (routing, execution, result, and merging), so they can be checked group wise. Additionally the constraint manager performs constraint checking and queries constraints after they have been entered. This might be useful for debugging an agent or for including the constraints in the final report.

The constraint manger contains three parts: The first is a parser that is capable to test variable constants in the expression set of the branches. The second is an interpreter that enables adding special test methods in the test expressions. The parser and interpreter are used to parse and evaluate the constraints and are implemented by following common rules of compiler building [9,10]. And the thirds is an interface (as shown in Figure 1) that provides the basic methods for editing and testing constraints. It is the only part of the constraint manager directly used by the programmer. The interface encapsulates the other components (the parser and the interpreter) and catches all errors to convert them to user-errors.

```
addConstraint(t:Task, c:IConstraint)
deletConstraint(t:Task, c:IConstraint)
CheckConstraint(t:Task, r:Results): boolean
getConstraint(t:Task): Vector
updateSymbolTable(SymbolTable:Hashtable)
```

```
IConstraintManager
interface
addConstraint(t:Task, c:IConstraint)
deletConstraint(t:Task, c:IConstraint)
CheckConstraint(t:Task, r:Results): boolean
getConstraint(t:Task): Vector
updateSymbolTable(SymbolTable:Hashtable)
```

```
IConstraint
interface
ROUTING: int
EXECUTION: int
RESULT: int
MERGING: int
```

```
JBranchConstraintManager
stores
uses
JBranchParser & JBranchInterpreter
```

```
JBranchConstraint
rule: String
values: Hashtable
```

Figure 1: Class diagram of the constraint manager component.

The parser performs syntactical checks of the constraints stored by the constraint manager. In case of a syntactical error the parser stops checking further constraints and an exception will be thrown. Consider the following constraints:

* if hostname == "A" then begin true end.
* if localUtilization() <> 1 then begin true end.
* if agentsystem == "MySystem" then begin true end.

The interpreter uses the abstract syntax tree generated by the parser. It interprets the statements and performs semantical checks. When a type error is recognized further interpretation of the statements is stopped and an exception is thrown. Type checking is done while interpreting the statements.

The reference implementation used in the proposed framework is the JBranchConstraintManager. It consists of an implementation of the constraint
manager interface and an interpreter.

Constraint modification operations have been realized by implementing the methods of the constraint manager interface allowing to add and remove of constraints. If a constraint has to be changed, it would have to be removed first and then added again. Constraints are grouped by using an attribute in the constraint object encoding the values given in the constraint interface. To offer the functionality to include constraints in reports they are stored in human-readable form. They are stored in an array of characters and then getConstraints() method returns this array. As constraints are stored in strings comparing constraints is implemented by a simple string comparison. There is no comparison if the referenced variables have the same value and there is no comparison if the referenced methods are the same.

3. Supervisor Component
The supervisor manages and controls the workers while they are executing their sub-tasks. Besides this it merges the results of the workers and sends messages to the user.

One of the basic requirements of the supervisor component is capable to divide tasks. As this is a non trivial task the component provides an external interface to allow the extension of this functionality [10]. The supervisor component is also capable to coordinate workers. It especially creates and deletes them and assigns tasks to them. Besides coordinating the workers the supervisor component coordinates subtasks.

It allows to keep track of the subtasks that are finished and to keep track of the subtask/worker relation. This component is also responsible for generating reports of the worker’s results.

Figure 2 shows the structure of the supervisor component. The supervisor component stores lists of: the current active workers roaming the network, the current list of subtasks, and a list of secure places. The list of active workers is used to keep track of them and to control them. The list of current subtasks is used in combination with the list of active workers. This allows the supervisor to store the worker/task relation. The list of secure places is used to check if a secure place closer to its area of activity exists.

The supervisor component has an open interface for strategy objects. So various strategies can be plugged in. Figure 2 shows the example of a routing strategy. These strategies can be added on-the-fly. So even when the component has already moved to a different place strategies can be added.

The task is passed to the strategy which is responsible to divide the task into smaller pieces. The component stores the strategies which have already been applied onto the task, so none of them is applied twice onto the same task. To coordinate the workers the supervisor component stores the worker/task relationship and uses it to keep track which worker is currently executing which task. The relationship is stored in a hashtable.

Figure 2: Class diagram of the supervisor-component.
using the unique ID of the workers as key. As a proposed solution for deeper hierarchies a heap-structure [11] could be used. The reports are generated by summarizing the results sent by the workers. The report is a plain ASCII file.

4. Worker Component

The worker moves from place to place and executes the tasks, created by the users or the supervisor. The worker component provides functionality for task execution. It should be able to execute a given task at different places. Furthermore the worker should implement result sending as the results created by the tasks at a foreign place have to be transmitted back to the supervisor component. Finally the worker component should also be capable to store all of its way points to visit one after the other and therefore needs some travel planning functionality.

The structure is shown in Figure 3. The worker is associated with an itinerary object. This object stores the places the worker will visit. Additionally it provides methods to access the list of places.

Executing the task has been implemented by inheriting and implementing the defined task interface. As soon as a worker arrives at a place, it calls the tasks’ “execute” method. The results are returned to the supervisor via a TCP connection. The worker contacts the supervisor and transmits the result object.

![Figure 3: Class diagram of the worker-component.](image)

This tasks are executed by the workers. A simple worker executes all of its tasks at each place it visits. A task component can store data in the result set, which is sent back to the supervisor either after the task has been executed at one place or when the task is finished at all. Additionally the task has to set a flag when its subtasks are finished.

One of the requirements is that the task component stores information. This information, whose can vary from simple primitive values to complex objects, will be used for later processing. Additionally the task component has to be capable to decide when the task was executed correctly and finished. So the objects can query the status of the task. Finally, the component stores ancestors. So it is possible to look up its super-tasks and merge the subtasks’ results to super-tasks’ ones. The structure is depicted in Figure 4. The task has an association to a class called Result.

![Figure 4: Class diagram of the task-component.](image)

The information is stored by the Result object associated to the task component. It stores the information in tuple form: ["name", @Reference Object]. This object can be easily sent back to the supervisor for further processing. The status is encoded in the task’s attribute and also in the result object. This way the status can be queried by the supervisor after the result has been returned.

The main problem of the user in its environment is to find a service to book a flight and to find a service to book a hotel. A use case diagram of this problem is shown in Figure 5. The first assumption needed for this implementation is about the available technical infrastructure which can be assumed to be Internet like.
In this scenario there are several elements which are divided into a logical group and a physical group.

Figure 5: Use case diagram of the case study application.

4.1 Physical Group
The physical group consists of elements like the user, travel agencies, hotels, and airlines. Each element plays a certain part in this scenario and acts in different ways.

User: represents a typical ordinary customer. He needs a return flight ticket and a hotel reservation.

Travel Agencies: The scope of duty has changed for the travel agencies. They do no longer all the work shown in Figure 5. They provide an infrastructure where to find information about flights and hotels.

Airline Companies: They present their available flights and the corresponding prices in flight databases. Depending on the airline company the advertise their flights in one or more databases.

Hotels: Like airline companies, hotels are represented in either one or many hotel databases. Using these databases they advertise their rooms and their services.

4.2 Logical Group
The logical group consists of abstract elements like workers, the supervisor, secure mobile agent systems, mobile agent systems, places, flight databases and hotel databases.

Worker: For this scenario the worker needs to find a flight or a hotel which suits the needs of the user. Then the results will be returned to the supervisor.

Supervisor: It is the coordinating element in this scenario. It manages the workers and collects flight and hotel information found by the workers. Finally it selects the best offer and creates a report which will be sent back to the user.

Secure Mobile Agent Systems: They are known by the supervisor which will choose the one nearest to the area of its operation. Secure mobile agent systems are trusted and secure. This means that they do not attack the residing mobile agents in any way. Furthermore it can be assumed that these systems will resist third party attacks.

Mobile Agent Systems: Each of them has at least one place. Some of these places belong to airlines which offer flight databases. Some of them belong to hotels which advertise their rooms. And finally some of them belong to travel agencies which provide special infrastructure.

Flight Databases: These databases contain all the information about available flights and their prices. These databases usually store the flight information of several airlines. However, they can also exclusively represent only one airline.

Hotel Database: As the user also needs a room reservation, hotel databases provide information about room availability and prices. Like the flight databases hotel databases store information about one or more hotels.

5. Logical Group

Interactions
In the scenario all these elements work together (those from the physical group as well as those from the logical group). Figure 6 shows the abstract interaction which happens within the logical group. The user initializes the supervisor component (step one). Then the supervisor component uses strategies (which were assigned at its initialization phase) to split the tasks (step two and three) and creates the workers (step four). After creating the workers, the supervisor waits until the workers return the results. While it waits the workers move to the first place and start their tasks (step five). The task collects the information and returns it to the worker (steps six, seven, and
1. initialize agent
2. split tasks
3. return tasks & constraints
4. create workers
5.1 move to place
5.2 invoke task
6.1 get flight info || 6.2 get hotel info || 6.3 book flight || 6.4 book hotel
7. return system information
8. return results
9.1 send results
9.2 loop to 5.1
10.1 merge results & create reports
10.2 return results

Figure 6: Interaction diagram of the case study application

The user creates a supervisor and adds the parallelizing strategy. Then it clones the tasks and defines routing constraints ([host, task] tuples for each one being created by the strategy). In our case the strategy creates the following routing constraints:

The user creates a supervisor and adds the “Parallelizing” strategy. Then the user adds the generic “find-flight” and “find-hotel” task and specifies the constraints.

The constraints for “find-flight”:
* if flight_price < 8000 then begin true end
* if flight_departure == "25.11.2003" then begin true end.
* if flight_destination == "JFK" then begin true end.
* if flight_source == "LHR" then begin true end.

The constraints for “find-hotel”:
* if hotel_price < 10000 then begin true end.
* if hotel_stay == "01.01.2004-06.01.2004" then begin true end.
* if hotel_location == "NY_USA" then begin true end.

After the supervisor is initialized it applies known strategies to split the main task. In this case the supervisor applies the “Parallelizing” strategy. It takes as its input a list hosts and a task. Then it clones the tasks and defines routing constraints ([host,
task] tuples for each one being created by the strategy). In our case the strategy creates the following routing constraints

1. if hostname = "A" then begin true end.
2. if hostname = "B" then begin true end.
3. if hostname = "C" then begin true end.

The tasks are not modified because the strategy only parallelizes the tasks. After creating these constraints the strategy returns [task, constraints] tuples. The supervisor then creates three workers-one for each tuple returned. After assigning the tasks and constraints each worker gets its initial list of hosts (A, B, C). Then all the workers take off.

Before starting to test its constraints each worker updates the values of the variables names. In this case it is the variable “hostname”. Then each worker starts with the first host of its list (Machine A) and tests its routing constraints. Only the first worker’s constraint is valid (all others get false as return value). So only the first worker moves to machine A, while the others try the next server of their list. Corresponding to this scheme the second worker moves to machine B and the third to machine C.

At these machines each worker executes its tasks ("find-hotel" and "find-flight"). Each task got the list of available flights and tests it against the constraints. After the tasks have been finished, the workers send the results back to the supervisor. Afterwards each worker disposes itself as there are no further tasks to fulfill and no further hosts to visit (the routing constraint does not allow the first worker to move to machine B or C after he finished his task at machine A). Finally the supervisor marks the best offer and sends back the results to the user.

7. Summary and Conclusion

Servers being visited by mobile agents are often a target of an attack. The mobile agent might steal resources, confidential data or use the server as starting point for a new attack. Accordingly many researchers devote their time to this area. However, another main problem of mobile agent security is nearly neglected: the attack of a malicious server against a mobile agent. The server can steal resources and confidential data, too. Due to the fact that the server has in general no access restrictions this problem is even harder to attack.

The current study presents an implementation for a solution for this particular aspect of mobile agent security: the Supervisor-Worker Framework.

We believe that the model implemented here is a representative of the initial work required in the construction of a complete trust model for a mobile agent system. Such a model would permit a detailed insight into the complex interactions that involve trust in mobile agent system. The success of such model will enhance greatly the security of m-commerce and e-commerce in ad hoc networks.

Future Work

Currently we are elaborating an extensive testing towards the validity of the model in a simulated environment.

References