# Simulation of Mobile Agent Network

Mario Kusek, Kresimir Jurasovic, and Ana Petric
University of Zagreb
Faculty of Electrical Engineering and Computing
Department of Telecommunications
Unska 3, HR-10000 Zagreb, Croatia
{mario.kusek, kresimir.jurasovic, ana.petric}@fer.hr

***Abstract*— This paper deals with an event based multi–agent system simulator. Agents in the simulator are based on the Mobile Agent Network formal model. The paper explains parts of the simulator such as operations, agents, nodes, network elements, etc. and how they work. In order to verify the simulation results, they were compared with the performance characteristics of a real multi–agent system, called the Remote Maintenance Shell, measured in a laboratory.**

## I. Introduction

Multi–agent systems based on mobile autonomous software agents that communicate and cooperate in order to perform operations instead of their owner have been applied recently. Their use has been recognized in telecommunications, business software modelling, computer games and many other fields. A multi–agent system containing mobile and intelligent agents is a promising paradigm for distributed system management. In our previous research, we have designed a formal model of a multi–agent system contining mobile agents, called the Mobile Agent Network (MAN)[1]. MAN was implemented into Remote Maintenance Shell (RMS)[2] used for remote software maintenance. Such systems are very complex and it is difficult to verify their properties formally or in real systems. Creating a simulation is the only viable approach to evaluate various behaviours of a multi–agent system.

The paper is organized as follows. Section 2 describes the basics of MAN, while Section 3 describes the MAN simulator. The agent system simulation is explained in Section 4 and the simulation of the network in Section 5. Section 6 compares and elaborates performances of the RMS system measured in the laboratory with the results form the MAN simulator. Section 7 concludes the paper.

## II. Mobile Agent Network

The Mobile Agent Network (MAN) is used for modelling agent organization and coordination in an agent team. The MAN is represented by a triple $\{A, S, N\}$, where $A$ represents a multi–agent system consisting of cooperating and communicating mobile agents that can migrate autonomously from node to node; $S$ is a set of $m$ nodes in which the agents perform operations; and $N$ is a network that connects nodes and assures agent mobility.

Each processing node $S_i$ has a unique $address_i$ from the set of addresses, $address = \{address_1, address_2, \ldots, address_i, \ldots, address_m\}$. An agent is defined by a triple, $agent_k = \{name_k, address_k, task_k\}$, where $name_k$ defines the agent's unique identification, $address_k \subset address$ represents the list of nodes to be visited by the agent and $task_k$ denotes the functionality the agent provides in the form of $task_k = \{s_1, s_2, \ldots, s_i, \ldots, s_p\}$ representing a set of assigned elementary operations $s_i$. When hosted by node $Si \in address_k$, $agent_k$ performs elementary operation $s_i \in task_k$. If an operation requires specific data, the agent carries this data during migration [1].

A network $N$ is represented by an undirected graph, $N = (S, E)$ which denotes network connections and assures agent mobility. The set of processing nodes is denoted as $S = \{S_1, S_2, \ldots, S_i, \ldots, S_m\}$. $E$ represents the set of edges $e_{ij}$ between $S_i$ and $S_j$ implying that nodes $S_i$ and $S_j$ are connected. The communication time $c_{ij}$ between tasks $t_i$ and $t_j$ (explained later) is associated with edge (link) $e_{ij}$ which connects these nodes. This way, a delay is incorporated into the communication channel. The following three types of network elements, with corresponding capacities, are defined: processing nodes, switches, and links (see Section 5).

User's request is decomposed to operations that are presented by a directed acyclic graph $G = (T, L)$, where $T$ denotes the list of elementary operations, while $L$ represents the set of directed edges that define precedence relations between operations [3]. Each operation is represented as an elementary operation with its input and output parameters. The operation from the list $T$ is defined by $t_i = \{i, s_{ti}\}$. The element $i$ represents the elementary operation number and $s_{ti}$ its elementary operation. Each $s_i$ is defined by $s_i = \{I_i, O_i\}$ where $I_i$ represents a set of input data $I_i = \{i_1, i_2, \cdots, i_{ni}\}$ and $O_i$ a set of output data $O_i = \{o_1, o_2, \cdots, o_{no}\}$. A set of directed edges $L$ is defined by $L = \{l_1, l_2, \cdots, l_i, \cdots, l_{nl}\}$. Each $l_i$ is defined by $l_i = \{t_{io}, o_i, t_{ii}, i_i\}$ where $t_{io}$ is an operation number of the output parameter $o_i$ and $t_{ii}$ is a task number of the input parameter $i_i$. If the operation receives input parameters in the process of creation then the edge is not presented in the operation graph [4].

## III. Simulator

### A. Related Work

The RMS is implemented in the JADE agent platform [5]. Such systems have a high level of complexity so it is difficult to verify their properties formally or in real

systems. In order to check various behaviours of a multi–agent system such as different agent coordination strategies, creating a simulation is the only viable approach [6]. The first step towards simulating the Mobile Agent Network was to study the features of existing simulators.

The Multi–Agent System Simulator (MASS) focuses on validating different coordinations and adaptive qualities of a multi–agent system in an unpredictable environment [7]. It does not consider environment with agents migrating from one place to another by using computer networks. In order to do that custom made component that conforms to Java Agent Framework has to be developed [8]. This component will not be compatible with JADE agent platform. Thus, this simulator is not viable option. The authors in the paper [9] are concentrated on how to simulate agents in a distributed system and they are using the network only for simulation. For example, in this simulator an agent can be moved from one place to another in a 2D environment; implementing a computer network in such environment is more complicated then to implement the whole simulator from scratch. On the other hand, the authors from [10] have built an event based simulation framework with a completely connected network. In this network different network topologies can not be modelled. This framework does not conform to our MAN model because duration time of one operation in MAN can not be simulated by modelling behaviour with Distilled StateCharts based approach. Another simulation toolkit is MASON [11], [12]. It is a single–process discrete–event simulation core and visualization toolkit. It is conceived as a core library for building domain–specific custom simulation library. Its special emphasis is on swarm simulations. In order to simulate the MAN model the custom simulation library must be developed. This is the reason why we divided our development into two phases: independent simulation, integration with MASON. The first phase is described in this paper.

### B. Simulator Core

The agent structure is defined in the MAN model. The simulator is programmed in Java as a part of PhD thesis [2]. Input data into a simulation are: duration of an elementary operation, size of a message in remote communication (between two nodes), size of an agent with operations and payload, coordination model, network topology with link capacity and network elements serving time. The simulation result is an operation graph execution matrix. The operation graph execution matrix analysis could find soft spots in the coordination model. These soft spots can be improved by coordination model. After correcting the coordination model, a simulation with the same parameters is repeated and the results are compared. An operation graph execution matrix generation could be omitted from the simulation and in that case the only simulation result is total execution time. This improves the simulation performance and resource consumption.

The class diagram shown in Figure 1 is the core of the simulator. The main class is `AgentSystem` which represents the whole multi–agent system. It contains a list of nodes

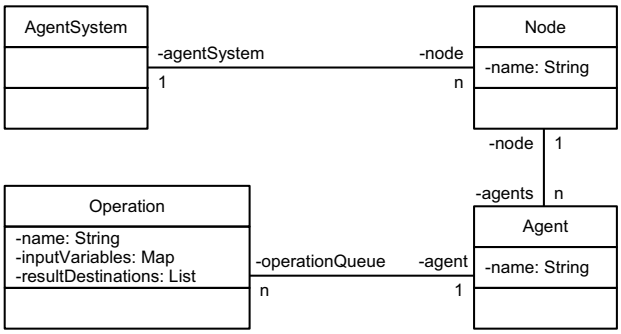


Fig. 1. Class Diagram of the Simulator Core

(class `Node`). Each node has a queue of agents (class `Agent`) at that node. An agent contains a queue of elementary operations (class `Operation` ) that must be executed. Each operation has attributes such as: name, input variables, and list of destinations where the execution results need to be sent. Operation input data are stored in a map with the input variable name as a key. The value can be null, which means that the value is not set. Input data are used for preconditions in the operation graph.

### C. Graph Definition

In order to run the simulation the operation graph needs to be defined. Let us use the simple graph from Figure 2 as an example. In this example there are three operations $t_1$ to $t_3$. The operation $t_1$ must be executed at the node $S_1$. The agent $A_1$ is going to execute operations $t_1$ and $t_2$.

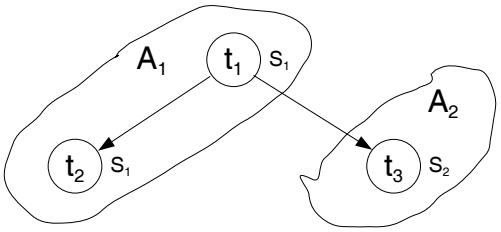


Fig. 2. Simple Operation Graph

The program needs to create an agent system and nodes (the following program lines 1–3). In line 1 the agent system is created. The node with name S1 is created and added to the agent system in line 2.

```
1 AgentSystem agentSystem = new AgentSystem();
2 agentSystem.createNode( "S1" );
3 agentSystem.createNode( "S2" );
```

The next step is to create agents. The method `createAgent` in the agent system creates the agent with the specified name, at the specified node (line 4). The agent name must be unique in the agent system and the node with the specified name must be created before. For example, the agent with the name A1 is created on the node S1 in line 4.

```
4 Agent a1 = agentSystem.createAgent("A1", "S1");
5 Agent a2 = agentSystem.createAgent("A2", "S2");
```

Furthermore, the operations should be created (lines 6–8), distributed to the agents (lines 9–11) and connected (lines 12–15). All operations are implemented as `NormalOperation` class. The constructor of the operation has two parameters: the operation name and the name of node where the operation will be executed (line 6). In order to assign operation to an agent `addOperation` method must be called. In line 9 operation t1 is added to the agent A1 for execution.

```
6 NormalOperation t1 =
  new NormalOperation("t1", "S1");
7 NormalOperation t2 =
  new NormalOperation("t2", "S1");
8 NormalOperation t3 =
  new NormalOperation("t3", "S2");
9 a1.addOperation( t1 );
10 a1.addOperation( t2 );
11 a2.addOperation( t3 );
```

Now the operations need to be connected. The operation t1 is connected to the operation t2 in line 14 and with the operation t3 in line 15. The connection represents sending the start signal between the connected operations. It is specified as destination address where the operation sends the result of its execution. The operation that receives the start signal must know that it should receive such signal before it is executed. For example, the operation t2 should receive the start signal and that is specified in line 13.

```
12 t2.createInputVariable( "start" );
13 t3.createInputVariable( "start" );
14 t1.addResultDestination( new
  DestinationAddressAtNodeOperation("start",t2));
15 t1.addResultDestination( new
  DestinationAddressAtNodeOperation("start",t3));
```

The last step is starting the simulation (line 18). If we want to see the simulation execution the execution logger must be created and registered. The creation of execution logger is in line 16 and its registration in the agent system is in line 17. While the simulation is running all relevant data are logged in the execution logger. In order to print this data method `serializeToString` converts it to the string (line 19). The structure of logged data is shown in class diagram in Figure 3, and the part of collected data from this example in object diagram in Figure 4. The method `getExecutionTime` in the agent system returns the total execution time.

```
16 SimulationExecutionLogger logger =
  new SimulationExecutionLogger();
17 agentSystem.setLogger( logger );
18 agentSystem.simulate();
19 String log = logger.serializeToString();
20 System.out.println( log );
21 System.out.println("execution time = " +
  agentSystem.getExecutionTime());
```

## IV. Agent System Simulation

### A. Simulation Execution

Results of the simulation are stored in the structure shown in class diagram in Figure 3. The main class is `SimulationExecutionLogger` which represents the whole simulation. It contains a list of elements executed at the specified time (class `InTimeExecution`). Each element has a

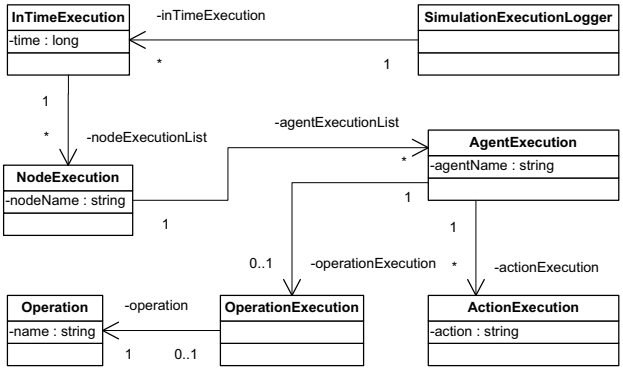


Fig. 3. Class Diagram of Simulation Execution Structure

time attribute. Each `InTimeExecution` object contains node executions (class `NodeExecution`). For each node in the system there could be one node execution at one time. Each node execution has a list of agent executions (class `AgentExecution`). Agent execution represents an execution of one agent at specified time. In the specified time an agent can execute infinite number of actions and only one operation. Actions do not consume processor time of the node while operations do. This is the reason why only one operation can be executed by one agent in the specified time on one node.

The print of simulation execution is the following:

```
0|S1|A1|B
0|S2|A2|B
1|S1|A1|t1
2|S1|A1|t2|CR->start@t2;CSI->start@t2;
CSR->start@t3@A2@S2
3|S1|A1|D
3|S2|A2|t3|CR->start@t3
4|S2|A2|D
```

The columns are separated by the symbol `|`. The columns represent: time period, node, agent, operation and actions. The first row represents that in time period of $0\Delta t$ on the node S1 the agent A1 is executing operation B. Since there is nothing else in that row the agent A1 is not going to execute any actions. There are two special operations: B — agent birth and D — agent death. Each agent consumes node processor when it is created and destroyed. The actions (see row at time $2\Delta t$) are separated by the symbol `;`. In this example we can see three actions: "`CR->start@t2`", "`CSI->start@t2`" and "`CSR->start@t3@A2@S2`". The actions can have the following marks:

- `CSI->iv@tn` — sending an internal message (between the operations executed in the same agent) to operation `tn` and setting the input variable `iv`;
- `CSL->iv@tn@an` — sending a local message (between the agents executing on the same node) to the operation `tn` executed by the agent `an` and setting the input variable `iv`;
- `CSR->iv@tn@an@sn` — sending a remote message (between the agents executed on different nodes) to the operation `tn` executed by the agent `an` on the node `sn` and setting the input variable `iv`;
- `CR->iv@tn` — receiving message in this agent on this

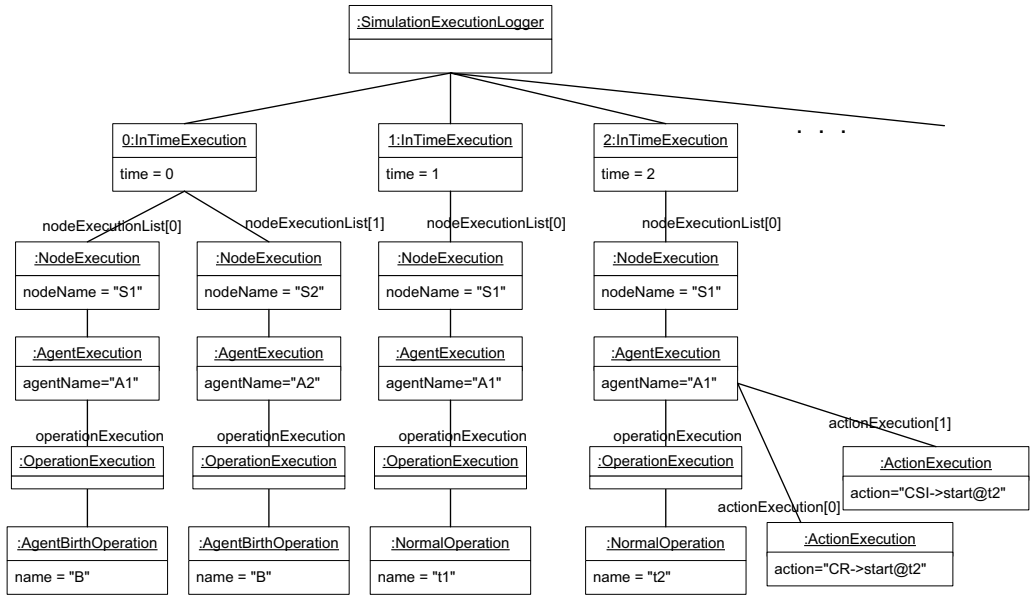


Fig. 4. Execution Object Diagram

node for operation `tn` and setting input variable `iv`;

- `TL->sn` — the agent starts to migrate to the node `sn`;
- `TA` — the agent has arrived to the node.

The first action is "`CR->start@t2`" and represents receiving a message for the operation `t2` and setting the input variable `start`. The second action is "`CSI->start@t2`" and represents sending an internal message to the operation `t2` and setting the input variable `start`. The third action is `CSR->start@t3@A2@S2` and represents sending a remote message to the operation `t3` executed by the agent `A2` on the node `S2`. As we can see the order of actions is not important.

The part of the execution, from the example, represented in memory is shown in object diagram (Figure 4). As shown, the `SimulationExecutionLogger` object has a reference to `InTimeExecution` objects, which represent simulation of time periods. Each `InTimeExecution` object has two references to `NodeExecution` objects. Each of these objects represents execution at one node. `NodeExecution` for node S1 in time period $2\Delta t$ has a reference to one `AgentExecution`, which represents execution of the agent A1. That agent is executing one operation (reference to the object `OperationExecution`). Each `OperationExecution` object must have a reference to the operation. In this case it is `NormalOperation` object with name attribute set to t2. This agent executes three actions in the same time period. Those actions represent sending and receiving messages. This is represented as references to the `ActionExecution` objects.

The simulation execution starts by handling events in the event queue. It handles the first event from the queue and checks if the queue is empty. In that case the simulation ends. In other case handling events in the queue is repeated. Each event has a reference to a certain object in the simulator eg. node, agent, operation, etc. Handling such an event is represented as an incoming asynchronous message in sequence diagrams. If an object is going to send an asynchronous message it puts a message event in the queue. Creating a node in the agent system puts the event with a *Start Node* message into the queue. After receiving a *Start Node* message the node starts the execution (Figure 5).

A *Start Node* message dequeues the first agent from the agent queue at the node and schedules the agent execution by sending a *Schedule Agent* message to the agent. The agent fetches the first operation from its operation queue and starts the execution of the operation by sending a *Start Operation* message. The operation execution does the work and after finishing, it sends an *Operation Finished* message to the agent. The agent removes the operation from the operation queue and sends an *Agent Finished Operation* message to the node. The node enqueues the agent as the last agent in the agent queue and after that schedules the next agent on the node. Since this agent is the only agent in our example the same agent is scheduled again. After the agent receives a *Schedule Agent* message it fetches the first operation from the operation queue. In our example this is the operation t1. The agent starts the operation and after it is executed the operation sends a *Send Message* to the agent. After receiving this message the agent sends the results to the operations that depend on the executed operation. In our example those operations are t2 and t3. The operation t2 is executed by the same agent as the operation t1 so the agent needs to send internal message to that operation. Precisely, the agent sends a *Receive Message* to the operation t2 and *Send Internal Message* to the operation t1 as a confirmation that the message was sent. When an operation receives a *Receive Message* it sets the input data with the value from the received message. After setting all input data the operation can be executed. If the first operation in the agent queue can not be executed, the agent sends an

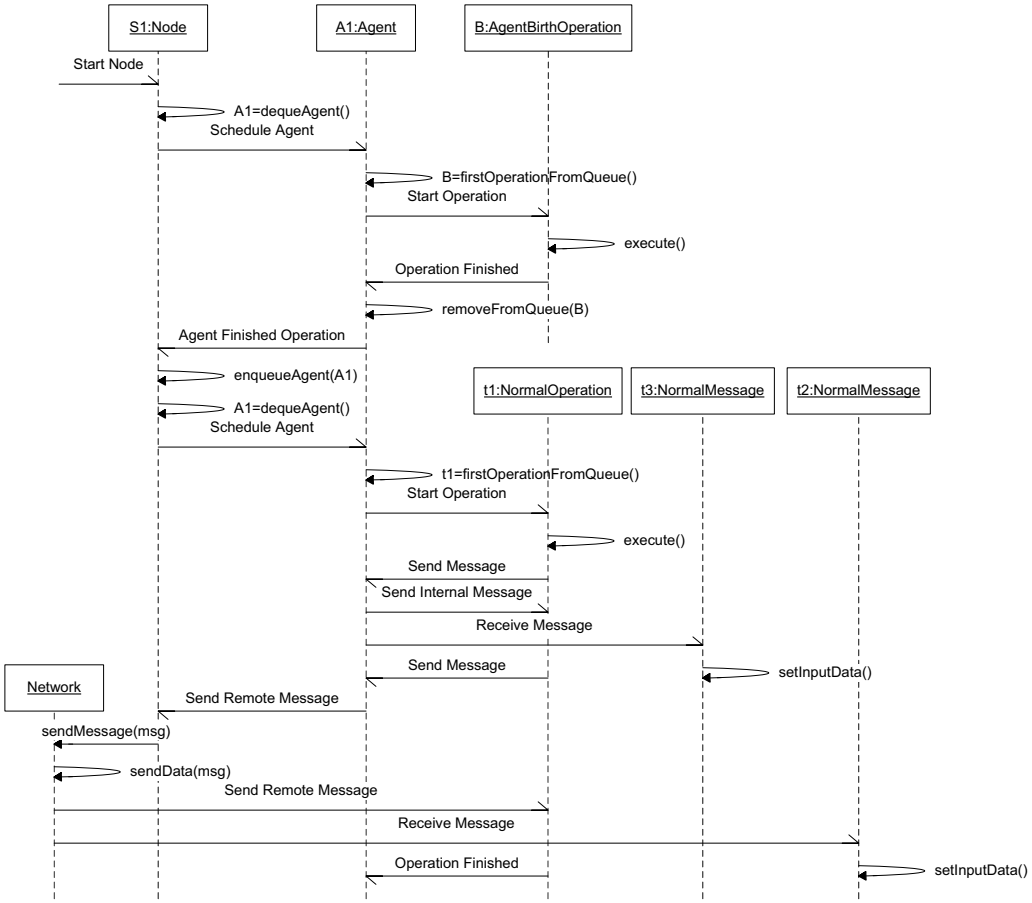


Fig. 5. Execution Sequence Diagram

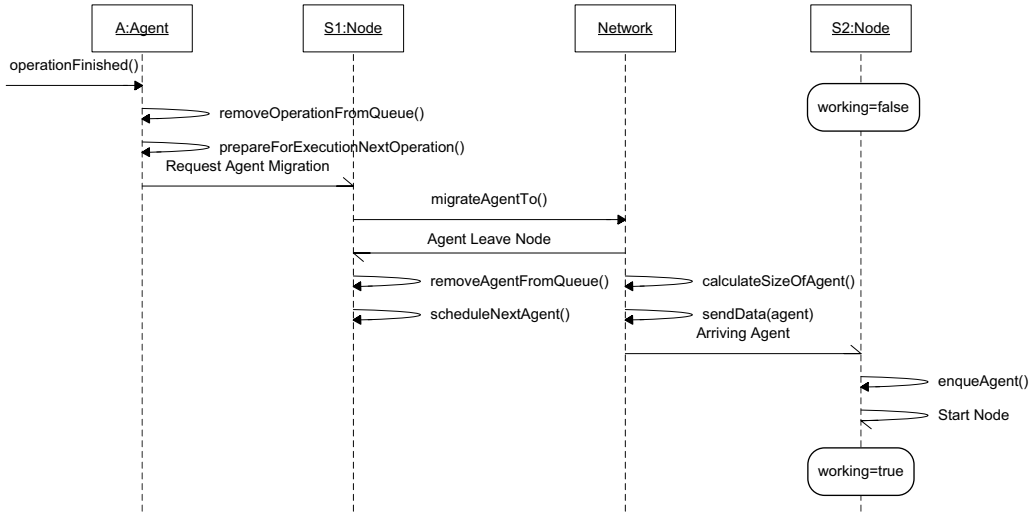


Fig. 6. Agent Migration Sequence Diagram

*Agent Can Not Execute Operation* message to the node and the node schedules the next agent in the queue. After sending an internal message in our example (Figure 5) the agent also has to send a message to the operation t2. Since this operation is executed by an agent in another node the remote message must be sent. To do that the agent A1 sends a *Send Remote Message* to the node which calls a `sendMessage` method in the network object. The network object is responsible for delivering a message through the network. It calls `sendData` method responsible for sending network message (details are described in the following section). After calling that method the operation t1 receives the confirmation that the message was sent by receiving a *Send Remote Message*. After a certain time (depending on the size of the message, the network topology, network element and link capacity) the message arrives on the destination and a *Receive Message* is delivered to the operation t2. After all messages are delivered the operation t1 is finished.

Figure 6 shows a scenario where the agent is migrated from the node S1 to the node S2. After the agent ends the previous operation, the `operationFinished` method is called. This method removes the previous operation from the operation queue and calls the `prepareForExecutionNextOperation` method, which checks if the agent has to migrate to another node for to execute the following operation. The agent sends a *Request Agent Migration* message to the node where it is currently located on (node S1) if it has to migrate to another node (node S2). The node S1 calls the `migrateAgentTo` method in the network object and the network confirms start of the migration by responding with an *Agent Leave Node* message. The node removes agent from the agent queue and schedules the next agent on that node. In order to migrate an agent to another node the network has to calculate the size of the agent and sends it to the network. The same mechanism used for sending agents is also used for sending messages. After arriving on the destination node the network sends an *Arriving Agent* to the arriving node (node S2). The node S2 enqueues the arrived agent in the agent queue and starts the node by sending a *Start Node* to itself, if it is not already working.

## V. Network Simulation

The N in the MAN represents the physical network that agents use while migrating and communicating. The core of the simulated network is a component common to all the network elements. This component can be regarded as a black box with a set of connectors. Each connector (marked with symbol $C_i$ where $i$ is the connector number) represents an input/output of the component. Connectors connect different components with logical links ($LL_i$). Logical links only connect entities in the network and do not introduce any link delay. There are three implementations of component: the link, switch and processing node entities.

Figure 7 shows an example of a network with one link, one switch and one processing node. The processing node is connected via its $C_i$ connector to the link's $C_j$ connector with a logical link. Furthermore, the link's $C'_j$ connector is connected with a logical link to the $C_k$ connector of the switch. The switch entity can have more than one connector allowing connections with multiple processing nodes or switches.

Processing node ($S_i$) represents a network node from the MAN model. It contains two elements: a network host ($V_i$) and an agent node ($AG_i$). The network host offers communication functions to the agent node. The agent node represents the agent platform running on the processing node.

Link entities represent full–duplex physical links which connect nodes and switches in the network. Each link is limited by its network capacity according to the queuing theory. A link can be divided into two components: a queue ($TQ_i$) and a service station ($P_i$) [13]. The queue is used to store processing requests that cannot be processed at that particular time since the service station is already processing some other request. In the network model, a processing request is data regarding the agent sent during the process of agent migration or the content of a message. The service station represents an Ethernet card used to send data through the network. The process of sending data over a link is performed in the following manner: first the link receives a processing request from a component connected to it through a connector. After receiving the request, it is stored in the queue. The service station then takes the request from the queue and sends the data to the destination component through the corresponding connector. The time needed to send the data is defined as follows: $t_{si} = b_i/C$, where $t_{si}$ is the service time for request $i$, $b_i$ is the size of the data being sent for request $i$ and $C$ is the link capacity. In our network model we assume that the queue is infinite employing the first-come-first-served queuing discipline. In our model there is only one service station at each link.

The switch entity represents a network switch used to transfer data between hosts. The switch is composed of three components: a queue, a service station and delivery logic. The queue and the service station are modelled using the same principles as for the link entity. The only difference is that the switch entity's service station has a deterministic service time. The delivery logic component was introduced since a request needs to be sent to the corresponding outgoing connector (depending on the destination) after processing. It contains a routing table with a list of hosts and the connectors leading to them. The routing table is updated every time data is received

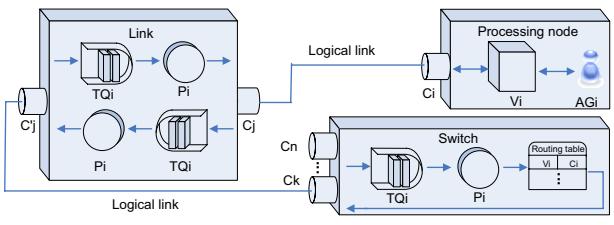


Fig. 7. Network Elements Structure

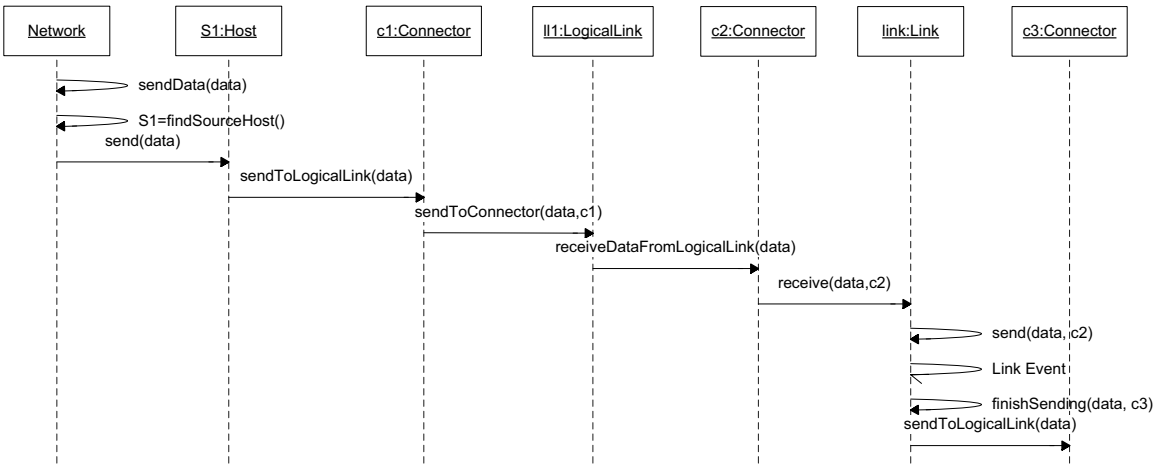


Fig. 8. Sequence Diagram of Data Sending

from a host not present in the table. The delivery logic is placed after the service station element.

The `sendData` method in the network object is called in order to send one packet (an agent or a message) to the network (Figure 8). The network object finds the host object (representation of the network host) of the source node (an agent node) and calls the `send` method of the host S1. The host sends data to its logical link by calling the `sendToLogicalLink` method in the connector c1. The connector forwards data to the logical link ll1 by calling the `sendToConnector` method. Since the logical link has two connectors it needs to know to which connector to send the data. This is the reason why the source connector is the parameter of the calling method. The logical link ll1 forwards data to the connector c2 by calling the `receiveDataFromLogicalLink` method. The connector knows from which logical link the data came and it calls the `receive` method in the link. The connector sends the reference to itself as the parameter. The link calls its `send` method which generates a *Link Event* message. This event represents the delay on the link. When this event is going to be handled by the simulator, the time will be advanced to the time of data arriving on the other end of the link. After that the `finishSending` method is called in order to send data to another logical link. This is repeated between components in the network until the data arrives on the destination host.

## VI. Performance Results

In order to prove that the MAN simulator can be used to simulate multi–agent systems based on the MAN model it was necessary to compare the results from the simulator with an actual multi–agent system. Experiments in [14] used the MAN simulator to simulate the execution of operations performed by the RMS system. The RMS system is an agent–based framework used to perform software management operations on remote locations. A total of 192 experiments were conducted in which both systems were used to perform software installation operations on remote locations. Three types of parameters were varied: number of remote locations (ranging from one to eight), number of operations to be performed (ranging from one to eight software installation operations) and the network bandwidth (512 Kbit/s, 1Mbit/s and 10 Mbit/s). Comparison of the results obtained from the experiments proved that the MAN simulator can be used to simulate multi-agent systems.

In this section a comparison of the results obtained from the previous experiments will be compared with the MAN simulator which introduces network elements. In the Figures 9 and 10 the x–axis represents the number of remote locations (PCs) where the software needs to be installed, the y–axis represents the number of software installation requests for every remote location and the z–axis represents the time needed to perform all operations in a single experiment (total execution time). Accuracy of the MAN simulator is shown in Figure 11 and is calculated by the following formula $RE = ((t_{rms} - t_{man})/(t_{rms}) * 100)$ where $RE$ is relative error in $\%$ between the results. $t_{rms}$ is the total execution time of the experiment in the RMS system and $t_{man}$ is the total execution time of the experiment in the MAN simulator. As it can be seen from the Figures total execution time raises linearly for both the RMS system and the MAN simulator. In the worst case scenario (experiments with

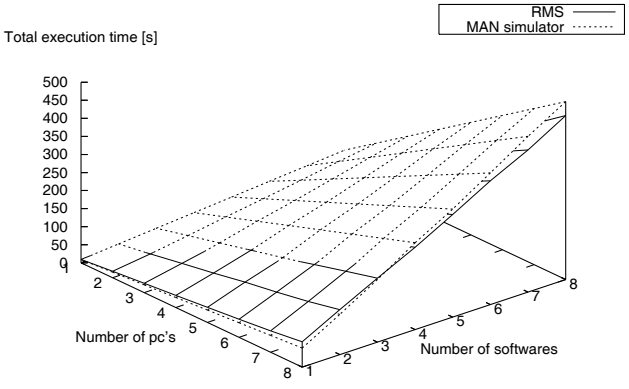


Fig. 9. 512 kbit/s Network Bandwidth

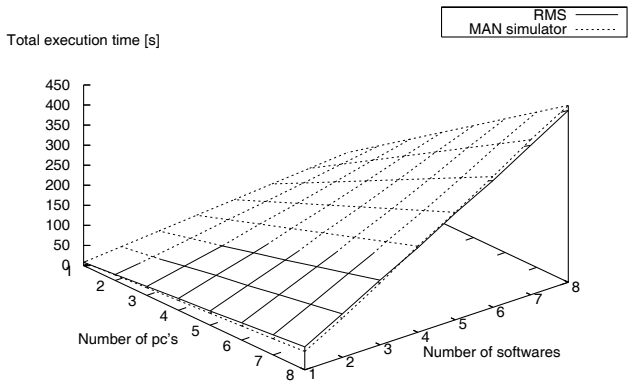


Fig. 10. 1 Mbit/s Network Bandwidth

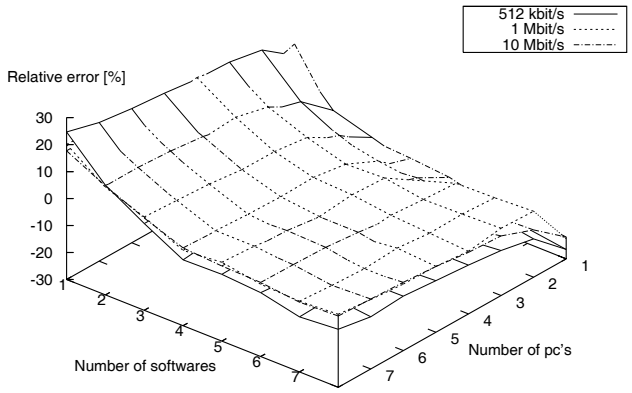


Fig. 11. Relative Error Graph

512 Kbit/s network bandwidth) the average $RE$ was 10.5% while in the best case scenario (experiments with 1 Mbit/s network bandwidth) the average $RE$ was 7.58%. The area with the small number of installations and where the number of remote locations is exactly one gives the worse case scenario results for all bandwidths. When comparing results within the experiments with the same network bandwidth the worst case scenario the results were obtained as the number of installation operations decreased and when the number of remote locations is one. There are several reasons for this. The way configuration parameters of the MAN simulator are measured is one of them. The simulator configuration parameters are obtained by measuring RMS system performance. The average operation execution time is used for the simulator configuration and that is the reason why is the $RE$ higher in the cases with the small number of operations in the experiment. The second lies in the fact that the MAN simulator does not take under consideration the influence of agent platform, operating systems or other network nodes. The MAN simulator performs operation sequentially while the RMS system performs operations in parallel. This also influences in increasing the $RE$.

When comparing the results from the previous experiments with the results from the MAN simulator, with introduced network elements, the average $RE$ decreases for 6.32%. The average $RE$ for the experiments with 512 kbit/s network bandwidth was 10.5% while in the previous results [14] it was 17%. In the experiments with 1 Mbit/s network bandwidth this difference was 7.58% (previously 13.6%) and in experiments with 10 Mbit/s network bandwidth it was 7.9% (previously 14.34%).

## VII. Conclusion and Future Work

In this paper, we have presented an agent simulator based on Mobile Agent Network formal model. This paper describes the basic simulator elements and their implementation. We have compared the simulation results with the results obtained by conducting experiments on the real multi–agent system (RMS). The results showed that the simulator can be used to simulate environments with the average $RE$ (relative error) under 10.5%. For higher network capacity the results were more accurate. In the future we plan to integrate the simulator into the MASON framework in order to improve $RE$.

## References

[1] M. Kusek, I. Lovrek, and V. Sinkovic, "Agent team coordination in the mobile agent network," in *Lecture Notes in Computer Science – LNCS*, ser. LNAI. Springer Verlag, 2005, vol. 3681, pp. 240–245, 9th International Conference Knowledge–Based Intelligent Information and Engineering Systems – KES 2005.

[2] M. Kusek, "Coordination of mobile agents for remote software system operations," Ph.D. dissertation, University of Zagreb, FER, 2005.

[3] I. Lovrek, V. Sinkovic, and G. Jezic, "Communicating agents in mobile agent network," in *Sixth International Conference on Knowledge-Based Intelligent Engineering Systems & Allied Technologies (KES)*, Crema, Italy, 2002, pp. 126–130.

[4] G. Jezic, M. Kusek, S. Desic, A. Caric, and D. Huljenic, "Multi–agent system for remote software operations," in *KES2003 Seventh International Conference on Knowledge-Based Intelligent Engineering Systems & Allied Technologies*, ser. Lecture Notes of Artificial Intelligence (LNAI) 2774, University of Oxford, United Kingdom, 2003, pp. 675–682.

[5] CSELT and Computer Engineering Group of the University of Parma, *Java Agent DEvelopment Framework (JADE)*, 2003, available online at: http://jade.tilab.com/.

[6] G. Fortino, A. Garro, and W. Russo, "An integrated approach for the development and validation of multi agent systems," *Computer Systems Science & Engineering*, vol. 20, no. 4, pp. 259–271, 2005, CRL Publishing Ltd.

[7] B. Horling, V. Lesser, and R. Vincent, "Multi-Agent System Simulation Framework," *16th IMACS World Congress 2000 on Scientific Computation, Applied Mathematics and Simulation*, August 2000. [Online]. Available: http://mas.cs.umass.edu/paper/186

[8] R. Vincent, B. Horling, and V. Lesser, "An Agent Infrastructure to Build and Evaluate Multi-Agent Systems: The Java Agent Framework and Multi-Agent System Simulator," in *LNAI: Infrastructure for Agents, Multi-Agent Systems, and Scalable Multi–Agent Systems*, vol. 1887. Springer,, January 2001, pp. 102–127.

[9] B. Logan and G. Theodoropolous, "The distributed simulation of multi–agent systems," *Proceedings of the IEEE*, vol. 89, no. 2, pp. 174–185, 2001.

[10] G. Fortino, A. Garro, and W. Russo, "A discrete-event simulation framework for the validation of agent-based and multi–agent systems." in *WOA*, F. Corradini, F. D. Paoli, E. Merelli, and A. Omicini, Eds. Pitagora Editrice Bologna, 2005, pp. 75–84.

[11] S. Luke, G. C. Balan, L. A. Panait, C. Cioffi-Revilla, and S. Paus, "Mason: a java multi–agent simulation library," in *Proceedings of Agent 2003 Conference on Challenges in Social Simulation*, 2003.

[12] S. Luke, C. Cioffi-Revilla, L. Panait, and K. Sullivan, "Mason: A new multi–agent simulation toolkit," in *Proceedings of the 2004 SwarmFest Workshop*, 2004.

[13] L. Kleinrock, *Queueing Systems, Volume 1: Theory*. John Wiley & Sons, 1975.

[14] K. Jurasovic and M. Kusek, "Verification of mobile agent network simulator," in *process of publication in KES AMSTA 2007 Conference*, 2007.