VERIFICATION OF THE MOBILE AGENT NETWORK SIMULATOR – A TOOL FOR SIMULATING MULTI-AGENT SYSTEMS

MARIO KUSEK, KRESIMIR JURASOVIC AND GORDAN JEZIC

University of Zagreb
Faculty of Electrical Engineering and Computing
Department of Telecommunications, Unska 3, Zagreb, HR-10000, Croatia
mario.kusek@fer.hr
kresimir.jurasovic@fer.hr
gordan.jezic@fer.hr

Received (Day Month Year)
Revised (Day Month Year)
Accepted (Day Month Year)

This paper deals with the verification of a multi-agent system simulator. Agents in the simulator are based on the Mobile Agent Network (MAN) formal model. It describes a shared plan representing a process which allows team formation according to task complexity and the characteristics of the distributed environment where these tasks should be performed. In order to verify the simulation results, we compared them with performance characteristics of a real multi-agent system, called the Multi-Agent Remote Maintenance Shell (MA-RMS). MA-RMS is organized as a team-oriented knowledge based system responsible for distributed software management. The results are compared and analyzed for various testing scenarios which differ with respect to network bandwidth as well as task and network complexity.

Keywords: multi-agent system; mobile agent network; verification, simulator.

1. Introduction

In recent years multi-agent systems based on autonomous software which migrates from host to host while communicating and cooperating with other agents in order to perform operations in place of their owner, have been applied in telecommunications, business software modeling, computer games, and many other fields. A multi-agent system containing mobile and intelligent agents is a promising paradigm for network and distributed systems management. This is particularly true for software operation and configuration in large environments, such as mobile telecommunication networks or Grid networks. Software operation and maintenance of communication systems distributed over the network is a complex procedure. It is known from experience that the same software run on a target system can give results different from those obtained on the test system. Isolating software under maintenance to prevent side effects which can influence normal operation, and support for
performing operations remotely are serious problems.

The Remote Maintenance Shell (RMS) is a solution for remote software operations and maintenance [12]. It represents the protected environment for software operations performed by mobile agents. The Multi–Agent Remote Maintenance Shell (MA–RMS) is based on a Mobile Agent Network (MAN) and is organized as a team–oriented multi–agent system. It consists of a master agent and a team of agents, where knowledge of the system is shared between the master and team agents. In systems with a high level of complexity, such as the MA–RMS, it is difficult to verify its properties formally or in a real system. In order to check various behavior of a multi–agent system such as different agent coordination strategies, creating a simulation is the only viable approach. Simulations are capable of simulating the functionality of the system and therefore be used to perform system analysis faster and cheaper.

In this article we present the Mobile Agent Network (MAN) simulator, a tool for simulating multi–agent systems. Agents in the simulator are based on the MAN formal model. It describes a shared plan representing a process which allows team formation according to task complexity and the characteristics of a distributed environment where these tasks should be performed. Verification of the simulator is done by the comparison with real multi–agent system MA–RMS responsible for distributed software management. Various testing scenarios are taken into consideration which differ with respect to network bandwidth, as well as task and network complexity.

The paper is organised as follows: after presentation of related work in this Sec., the next Sec. deals with the Mobile Agent Network formal model. Sec. 3 elaborates MAN simulator and explains simulation of agent systems and network nodes. Multi–agent Remote Maintenance Shell system, its architecture and remote software operations are presented in Sec. 4. Sec. 5 deals with laboratory architecture and measurements, and compares and elaborates performances of the MAN simulator with the results from MA–RMS system, and Sec. 6 concludes the paper.

1.1. Related Work

Our first step towards simulating a Mobile Agent Network was to study the features of existing simulators. Since MA–RMS was programmed using the JADE agent platform, it was necessary that the simulator also be capable of simulating these agents. The first simulator we analyzed was the Multi–Agent System Simulator (MASS). This simulator focuses on validating different coordinations and adaptive qualities of a multi–agent system in an unpredictable environment [6]. It does not consider an environment where agents migrate from one place to another using computer networks. In order to achieve this, a custom made component which conforms to the Java Agent Framework must be developed [19]. This component would not be compatible with the JADE agent platform. Thus, this simulator was not a viable option. The authors from [11] concentrate on how to simulate agents in a distributed
system and use the network only for simulation. In this simulator, an agent can be moved from one place to another in a 2D environment; implementing a computer network in such an environment is more complicated than implementing the whole simulator from scratch. On the other hand, the authors from [3] have built an event–based simulation framework with a completely connected network. However, different network topologies cannot be modeled. This framework does not conform to our MAN model because the duration time of one operation in the MAN cannot be simulated by modeling behavior using a Distilled StateCharts based approach. Another simulation toolkit is MASON [14, 15]. It is a single–process discrete–event simulation core and visualization toolkit. It is conceived as a core library for building a 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. For this reason, we divided the process into two phases: independent simulation and integration with MASON. The first phase is described in this paper.

2. The Mobile Agent Network

The Mobile Agent Network (MAN) is used for modeling agent organization and coordination in an agent team. The idea is that the user sends a request to the system. The request is then decomposed into a software operation task graph and executed by mobile software agents. 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, \( address_k \in address \) 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 \( S_i \in address_k \), \( agent_k \) performs elementary operation \( s_i \in task_k \). If an operation requires specific data, the agent carries this data during migration [9].

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.
2.1. Multi–Agent Systems

In systems comprised of people or organizations with different goals, a multi–agent system (MAS) is needed to handle their interactions. An MAS is categorized by the most important aspects of its agents: the degree of their heterogeneity and their communication. Examples of homogeneous communicating agents are agents that are inspired by the behavior of real ants [2]. These systems are typically used for agent–based routing and load balancing in telecommunication networks. Another MAS concept is a mobile agent team. This concept is used for solving complex tasks when individual agents’ expertise, information and/or resources are insufficient for the effective completion or performance of a task. Work regarding team–oriented multi–agent systems has mostly been inspired by two theories: the joint intentions theory [10] and the shared plans theory [5]. Both theories are based on observations of human teamwork and several solutions are based upon these theories. TEAM-CORE [18] introduces sophisticated monitoring techniques, which are separated from actual team–oriented communications. In the RETSINA system [4], agents only interoperate with other agents when they require another agent’s information or services. STEAM [17] integrates the following novel key concepts: team synchronization (to establish joint intentions); constructions for monitoring joint intentions and repair; and decision–theoretic communication selectivity. We propose a multi–agent system organized as a team of agents for remote software operations.

2.2. Team Oriented Multi–Agent Systems

Two multi–agent team oriented concepts are analyzed. The first is a master/slave concept, while the second consists of mobile agents with the same level of knowledge. Agents’ knowledge is composed of: agent capability, team capability, and situation–specific knowledge. Agent capability is knowledge regarding how and what kind of operation the agent can perform. Team capability is knowledge regarding the remaining agents in the team. Each agent knows where all other agents are and what operations they are executing. Situation specific knowledge is the capability of an agent to resolve unexpected situations.

The master/slave team model represents a centralized knowledge concept. An initial request composed of the set of software operations to be performed is submitted to the master agent. The master agent then creates a team of slave agents and sends them to different nodes in a network. All knowledge in this model is concentrated at the master agent who has three types of knowledge: its own capabilities, the team’s capabilities and situation–specific knowledge.

2.2.1. Centralized Knowledge Concept (CKC)

Based on user requests and operations, the master agent must be able to create a team plan, form a team of slave agents, and send them to perform the needed operations. Situation–specific knowledge represents the capability of the master
Verification of the mobile agent network simulator

agent to make decisions in unforeseen situations while executing the team plan. It can modify slave agents tasks during execution. Each slave agent from the team knows only how to perform one particular operation (i.e., the agents capability). If a problem arises during execution, the slave agent contacts the master agent. The master agent then takes appropriate action to try to solve the problem. If this is not possible, the master agent informs the user of the problem accordingly. When all operations have been successfully performed, the slave agents report their outcome to the master.

However, the centralized nature of situation–specific knowledge in this concept creates a considerable drawback, illustrated in the following example. Suppose the master agent has created a team plan and has formed a team of slave agents. If the state of certain remote nodes changes in the time interval between these two operations, a slave agent can be faced with an unexpected situation which it cannot resolve. The slave agent must contact the master agent in this situation creating the need for extra coordination.

2.2.2. Distributed Knowledge Concept (DKC)

In a distributed knowledge concept, all agents from the multi–agent system possess the same level of knowledge. Agents execute operations as a team while each individual agent possesses two types of knowledge: its individual capabilities and situation–specific knowledge. Each agent should know what kind of operation it can perform (its capability) and who can help if a problem arises during execution (situation–specific knowledge). A single agent can execute only one task. Furthermore, the agents correspond to the operations defined in Sec. 4.1.

Since a single agent executes only one task, the agent creates another agent which can solve its problem if an unexpected situation arises. Detection of such situations is part of its situation–specific knowledge. Thus, this concept does not require centralized knowledge nor centralized master–agent coordination. However, this requires creating a large number of agents in the system, each with only one task to execute. Agent creation and destruction in this environment generates more time and processor consumption than the centralized concept.

The organization of multi–agent systems suitable for network and distributed systems management considered in [20, 5, 16] is based on shared plans used by agent teams. An intelligent stationary agent is responsible for decomposing a complex management task into $n_t$ elementary operations and ordering these operations. The same agent also collects and interprets data regarding the characteristics of the nodes and the network in order to define a suitable agent team.

2.3. Shared Plan

The following assignments of elementary operations are considered the basic building blocks for identification of the agents’s shared plan:
**R1:** a single agent executes all operations on all nodes;

**R2:** an agent executes a single operation on one node only;

**R3:** an agent executes all operations on one node only;

**R4:** an agent executes a specific operation on all nodes;

**R5:** an agent executes a specific operation only once on all nodes;

**R6:** operations are assigned to the agents in order to exploit maximal parallelism of operation. Mutually independent operations are assigned to different agents, in order to execute them simultaneously on nodes with parallel execution supported;

**R7:** a hybrid solution combining R4 and R3. An agent is responsible for a specific operation on all nodes; all other agents execute all other operations, each on a different node;

**R8:** a hybrid solution combining R5 and R3 (specialization of R7 in the way R5 is specialization of R4).

Operations executed by agents are interdependent so when one operation finishes it must send its results to the operation which depends on it (explained in detail in Sec. 4.1). This is why we defined three types of agent communications:

1. **internal (I),** when the operations are performed by the same agent;
2. **local (L),** when the operations are performed by different agents at the same node;
3. **global (G),** when the operations are performed by different agents at different nodes.

Fig. 1 shows the lifecycle of an agent. Agent creation is characterized by its birth. After birth, the agent migrates to the first node where it has to execute an operation. If the agent carries data, its migration time is longer. Having arrived at the first node, it executes its operation and informs the other agents of its result via dialog. Dialog is the process of sending messages between two operations. The type of dialog depends on coordination (local or global communication). The agent then takes over the next operation on the task list. If this operation is to be executed at the same node, migration is skipped. If, however, the next operation is to be executed on another node, the agent migrates to the other node, executes the operation and then performs a dialog. This process is repeated for all operations from the task list \((task_k)\). The last operation is always the agents death by which the agent is disposed of.

The efficiency of the shared plan depends on the specific task submitted (its number, its ordering, and the complexity of its elementary operations) and environmental characteristics. The basic parameters which describe the environment are as follows: operation execution times, agent size, loaded agent size, message size, network topology, link bandwidth, shared plan type and network elements serving times. When an agent sends a message, the corresponding dialog fails if the receiving agent is not at the expected destination. In direct communication, the sender
periodically retries until the dialog is successful. In indirect communication, the sender creates a transport agent which migrates to the destination and delivers the message to the receiving agent upon arrival [1].

3. The MAN Simulator

3.1. The Simulator Core

The agent structure is defined in the MAN model. The simulator was programmed in Java as a part of the PhD thesis from [8]. The input data required to run a simulation are the same as environment characteristics. The simulation result is an operation graph execution matrix. Operation graph execution matrix analysis can find soft spots in the selected shared plan. The identified soft spots can then be used to improve the shared plan. After correcting the coordination model, a simulation with the same parameters is repeated and the results are compared. Operation graph execution matrix generation can be omitted from the simulation, in which case the only simulation result is the total execution time. This improves simulation performance and resource consumption.

The class diagram shown in Fig. 2 represents the core of the simulator. The main class is AgentSystem which represents the whole multi-agent system. It contains a list of nodes (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 is 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 is used for preconditions in the operation graph.
3.2. **Graph Definition**

In order to run the simulation the operation graph needs to be defined. Let us use the simple graph from Fig. 3 as an example. In this example there are three operations \( t_1, t_2 \) and \( t_3 \). Operations \( t_1 \) and \( t_2 \) are to be executed at the node \( S_1 \) by agent \( A_1 \) while operation \( t_3 \) is to be executed at node \( S_2 \) by agent \( A_2 \).

The program needs to create an agent system and individual nodes (the following program lines 1–3). In line 1, an agent system is created. A node with name \( S_1 \) is created and added to the agent system in line 2.

```java
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 an agent with the a specified name, at a specified node (line 4). The agent name must be unique to the agent system and the specified node must have been created before. For example, an agent named \( A_1 \) is created at the node \( S_1 \) in line 4.
Furthermore, the operations need to be created (lines 6–8), distributed to the agents (lines 9–11) and connected (lines 12–15). All operations are implemented as the NormalOperation class. The constructor of the operation has two parameters: the operation name and the name of node at which the operation will be executed (line 6). In order to assign an operation to an agent, the addOperation method must be called. In line 9, operation t1 is assigned 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. Operation t1 is connected to operation t2 and t3 in lines 14 and 15, respectively. These connections represent the sending of start signals between the connected operations. They are specified as the destination address where the operation sends the result of its execution. The operation which receives the start signal must know that it is supposed to receive such a signal before it is executed. For example, in line 12 it is specified that operation t2 should receive a start signal.

```
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 to start the simulation (line 18). If we want to be able to see how the simulation progressed during execution, an execution logger must be created and registered in the agent system. This is done in lines 16 and 17, respectively. During simulation execution all relevant data is logged in the execution logger. In order to print this data, the method serializeToString converts it to a string (line 19). The structure of logged data is shown in the class diagram in Fig. 4, while part of the collected data from this example is shown in the object diagram in Fig. 5. 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());
```
3.3. **Agent System Simulation**

3.3.1. **Simulation Execution**

Results of the simulation are stored in a structure described by the class diagram in Fig. 4. The main class is `SimulationExecutionLogger` which represents the whole simulation. It contains a list of elements executed at a specific time (class `InTimeExecution`). Each element has a time attribute. Each `InTimeExecution` object contains node executions (class `NodeExecution`). At each node in the system, there can only be one node execution at a time. Each node execution has a list of agent executions (class `AgentExecution`). An agent execution represents the execution of one agent at a specified time. In the specified time, the agent can execute an infinite number of actions and only one operation. Actions do not consume node processor time while operations do. This is the reason why only one operation can be executed at the node by one agent at a specified time.

The printout of simulation execution is the following:

<table>
<thead>
<tr>
<th>0</th>
<th>S1</th>
<th>A1</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S2</td>
<td>A2</td>
<td>B</td>
</tr>
<tr>
<td>1</td>
<td>S1</td>
<td>A1</td>
<td>t1</td>
</tr>
<tr>
<td>2</td>
<td>S1</td>
<td>A1</td>
<td>t2</td>
</tr>
<tr>
<td>3</td>
<td>S1</td>
<td>A1</td>
<td>D</td>
</tr>
<tr>
<td>3</td>
<td>S2</td>
<td>A2</td>
<td>t3</td>
</tr>
<tr>
<td>4</td>
<td>S2</td>
<td>A2</td>
<td>D</td>
</tr>
</tbody>
</table>

Columns are separated by the symbol `|`. Individual columns represent: time period, node, agent, operation and actions. The first row indicates that in time period of 0Δt agent A1 was executing operation B as node S1. Since there is nothing else in that row agent A1 did not execute any actions in that time period. There are two
special operations: B — agent birth and D — agent death. Each agent consumes
node processor when it is created or destroyed. Actions are separated by the sym-
"  
[126x671]bol ; (see row at time 2∆t). In this example we can see three actions: ‘Cr->start@t2”,
“CSI->start@t2” and “CSR->start@t3@A2@S2”. Possible actions and their notation are de-
scribed below:

• CSI->iv@tn — sending an internal message (between operations executed by the
same agent) to operation tn and setting input variable iv;

• CSL->iv@tn@an — sending a local message (between agents executing on the same
node) to operation tn executed by agent an and setting input variable iv;

• CSR->iv@tn@an@sn — sending a remote message (between agents executing on dif-
ferent nodes) to operation tn executed by agent an on node sn and setting input
variable iv;

• CR->iv@tn — receiving message in this agent on this node for operation tn and
setting input variable iv;

• TL->sn — the agent starts migrating towards node sn;

• TA — the agent has arrived at the node.

In our example, the first action is denoted as ”Cr->start@t2” and represents receiv-
ing a message for operation t2 and setting input variable start. The second action
”CSI->start@t2” represents sending an internal message to operation t2 and setting
input variable start. The third action is CSR->start@t3@A2@S2 and represents sending a
remote message to operation t3 executed by agent A2 on node S2. As we can see,
the order of actions is not important.

The part of simulation execution progress which is stored by the logger is shown
in the object diagram in Fig. 5. As shown, the SimulationExecutionLogger object has
references to InTimeExecution objects, which represent the 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∆t has a reference to one AgentExecution, which represents the execution of agent
A1. This agent is executing one operation (reference to the object OperationExecution).
Each OperationExecution object must have a reference to the operation. In this example,
it is the NormalOperation object with its name attribute set to t2. This agent executes
three actions in the same time period, indicated with references to ActionExecution.
These actions represent sending and receiving messages.

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. If such is the case,
the simulation ends. If not, the handling of events in the queue is repeated. Each
event has a reference to a certain object in the simulator, eg. a node, an agent, an
operation, etc. Handling such events is represented as an incoming asynchronous
message in sequence diagrams. If an object is planning 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 begins with the execution (Fig. 6).
A *Start Node* message dequeues the first agent from the agent queue at the node in question and schedules agent execution by sending a *Schedule Agent* message to it. The agent then fetches the first operation from its operation queue and starts its execution of the operation by sending a *Start Operation* message. Operation execution does the work and then sends an *Operation Finished* message to the agent upon completion. The agent then removes the operation from the operation queue and sends an *Agent Finished Operation* message to the node. The node then moves the agent to the end of the agent queue and schedules the next agent at the node. Since there is only one agent in our example, the same one 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 operation t₁. The agent then 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 one. In our example, these operations are t₂ and t₃. Operation t₂ is executed by the same agent as operation t₁ so the agent needs to send internal message to these operations. More precisely, the agent sends a *Receive Message* to operation t₂ and a *Send Internal Message* to operation t₁ as a confirmation that the message was sent. When an operation receives a *Receive Message*, it sets its input data with the value specified in the received message. After setting all the input data to the desired value, the operation can be executed. If the first operation in the agent queue cannot be executed, the agent sends an *Agent Can Not Execute Operation* message to the node after which the node schedules the next agent in the queue. In our example, after sending internal messages (Fig. 6)
the agent also must also send a message to the operation \( t_2 \). Since this operation is executed by an agent at another node, a remote message must be sent. To do this agent \( A_1 \) sends a *Send Remote Message* to the node which in turn calls a *sendMessage* method in the network object. The network object is responsible for delivering messages through the network. It calls the *sendData* method responsible for sending network messages (details are described in Sec. 3.4). After calling this method, the operation \( t_1 \) receives confirmation that the message was sent in the form of a *Send Remote Message*. After a certain time (depending on the size of the message, the network topology, network elements and link capacities) the message arrives at the destination and a *Receive Message* is delivered to operation \( t_2 \). After all messages are delivered operation \( t_1 \) is finished.

Fig. 7 shows a scenario where an agent migrates from node \( S_1 \) to node \( S_2 \). 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 in order to execute the subsequent operation. If so, the agent sends a Request Agent Migration message to the node where it is currently located (node S1). Node S1 calls the migrateAgentTo method in the network object after which the network confirms the start of migration by responding with an Agent Leave Node message. The node removes the agent from the agent queue and schedules the next agent at that node. In order to migrate an agent to another node, the network needs to calculate the size of the agent and send it to the network. The same mechanism used for sending agents is also used for sending messages. After the agent arrives at the destination node (node S2) the network sends an Arriving Agent message to this node. Node S2 enqueues the arrived agent in the agent queue and starts the node by sending a Start Node message to itself if it is not already running.

3.4. Network Simulation

The N in a MAN represents the physical network which agents use while migrating and communicating. The core of the simulated network is a component which is common to all network elements. 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 logically connect entities in the network and do not introduce any link delay. There are three implementations of a component: the link, switch and processing node entities.

Fig. 8 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 in accordance with classical queuing theory. A link can be divided into two components: a queue \((TQ_i)\) and a service station \((P_i)\) [7]. The queue is used to store processing requests which cannot be processed at that particular time since the service station is already processing some other request. In the network model, a processing request contains 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: 

\[
t_{si} = \frac{b_i}{C},
\]

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. Furthermore, we assume 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 modeled 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 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 (Fig. 9). The network object finds the host object (i.e., the representation of the network host) of the source node (an agent node) and calls the `send` method of host $S_1$. The host sends data to logical link by calling the `sendToLogicalLink` method in the connector $c_1$. The connector forwards data to the logical link $ll_1$ by calling the `sendToConnector` method. Since the logical link has two connectors it needs to know to which connector to send the data to. This is the reason why the source connector is a parameter of the calling method. Logical link $ll_1$ forwards data to connector $c_2$ by calling the `receiveDataFromLogicalLink` method. The connector knows from which logical link the data is coming from and calls the `receive` method in the link. The connector sends the reference to itself as a parameter. The link calls its `send` method which generates a `Link Event` message. This event represents the delay on the link. When such an event is handled by the simulator, the time is advanced to the time the data arrives at the other end of the link. After this, the `finishSending` method is called in order to send data to another logical link. This procedure is repeated between different components in the network until the data arrives at the destination host.

4. The Multi–Agent Remote Maintenance Shell

Software maintenance is, by IEEE definition, the modification of a software product after delivery. The main purposes of software maintenance are to correct faults, improve performance and/or other attributes, adapt the product to a changing environment and/or improve product maintainability [16]. Software management is also

![Fig. 9. Sequence Diagram of Data Sending](image-url)
a very demanding task in distributed systems since such systems are composed of a large number of computers located at geographically different areas. Service management operations, such as software migration, installation, starting or stopping become difficult when they have to be performed on tens or hundreds of computers. In the New Generation Network (NGN), which is characterized by the integration of traditional telecommunication systems and the Internet, this problem becomes even more important since it integrates different types of networks, programming technologies and provides an environment for new innovative personalized services. The NGN also supports mobility of terminals, users and services. In such environments, it is necessary to provide an advanced service management framework which can support such a dynamic and flexible service provisioning process. This process requires service deployment, configuration, control, upgrading, provisioning, monitoring, accounting, billing and self-management. Agents differ from other paradigms because they enable the development of software that is intelligent and can, thus, adapt to changing conditions.

4.1. Remote Software Operations
Remote software operations are started on the client side (management station) while most of the work is done on the server side (remote system). Software under maintenance can be found in different states in accordance with which maintenance actions must be defined (Fig. 10). The software migration operation includes all the actions necessary to transfer software from the management station to a remote system. Before migration, the software is in the initiated state. Installation data, specific to that particular piece of software, must be transferred together with the software itself. After the transfer is completed, a delivery report must be sent. After migration, the software is switched from the initiated to the inactive state and is ready for installation. The software installation operation includes all the

![Diagram](Fig. 10. Remote maintenance operations)
actions needed for software installation on the remote system. After installation, the software state is changed to the active/ready state. In this state, the software is active and ready for execution.

Four other operations are also defined:

- The **software starting** operation is composed of actions which change the software state from ready to running. In the running state, the software is being executed. In this state, software can be stopped, but must be restarted to reach the running state again. Possible errors and faults generated during the starting operation can be dangerous. In case of malfunctions, errors should be detected, the software execution stopped, and finally, a report should be sent to the user who initiated the starting operation;

- The **software stopping** operation can be activated only from the running state, in which case the software returns to the ready state. The software can be stopped in case of problems with the operating system or hardware (memory, processor). Furthermore, the user can initiate software stopping. After restarting, the software returns to the running state;

- The **software uninstallation** operation can be started only from the ready state, and after uninstallation, the software returns to the inactive state. Before uninstallation, it is important to determine the relationships between the software which is to be uninstalled and other applications installed on the same system. Some software units could be used by other applications and, thus, their removal could affect regular operation. If such relationships exist, the common part must be retained;

- The **software tracing** operation (passive logging) includes actions for collecting trace data during execution. Tracing can be initiated from the ready state and performed in the running state, or it can dynamically be turned on during software execution. The main idea of tracing is to collect input and output data from maintained software in order to analyze its correctness. Comparing log files from remote and home systems can be useful for debugging. It is important to underline that the first execution of new software should always be started with tracing turned on.

The following software operations will be considered: software migration, installation, setting execution parameters, starting, stopping, uninstallation and tracing. Each remote software operation is represented as an elementary task with its input and output parameters. A user request for remote software operations is represented by a directed acyclic graph $G = (T, L)$ where $T$ denotes a list of elementary tasks, and $L$ represents a set of directed edges which define precedence relations between tasks [13]. Each elementary task from the list $T$ corresponds to one software operation and is defined by $t_i = \{i, s_{ti}\}$. Element $i$ represents the elementary task number and $s_{ti}$ its service type. Each $s_i$ is defined by $s_i = \{I_i, O_i\}$ where $I_i$ represents a set of input data $I_i = \{i_1, i_2, \ldots, i_{ni}\}$ and $O_i$ a set of output data $O_i = \{o_1, o_2, \ldots, o_{no}\}$. A set of directed edges $L$ is defined by $L = \{l_1, l_2, \ldots, l_i, \ldots, l_{nl}\}$. Each $l_i$ is defined
by \( t_i = \{ t_{io}, o_i, t_{ii}, i_i \} \) where \( t_{io} \) is a task number of the output parameter \( o_i \) and \( t_{ii} \) is a task number of the input parameter \( i_i \). Input parameters which a task may receive in the process of creation are not presented in the task graph. The actual number of parameters (input/output) can only be determined at execution time.

For example, the install operation \( (s_2) \) has four inputs (Table 1). The first is the host name \((i_1)\) which represents the host where the task is to be executed. The second is the software name \((i_2)\). It is the name of the software that should be installed. Next is the start signal \((i_3)\) which is a trigger for starting the operation. Finally, the report address \((i_4)\) is the address of the agent and task to which it is sent. Report \((o_1)\) triggers another task to start its execution. As preconditions, software must be migrated. If this software is a version then testbed (version and

<table>
<thead>
<tr>
<th>Operation</th>
<th>Inputs</th>
<th>Outputs</th>
<th>Preconditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>migrate ((s_1))</td>
<td>host name ((i_1)), software name ((i_2)), start signal ((i_3)), report address ((i_4))</td>
<td>report ((o_1))</td>
<td>if it is version then testbed must be migrated</td>
</tr>
<tr>
<td>install ((s_1))</td>
<td>host name ((i_1)), software name ((i_2)), start signal ((i_3)), report address ((i_4))</td>
<td>report ((o_1))</td>
<td>software migrated if it is version then testbed must be installed</td>
</tr>
<tr>
<td>set execution parameters ((s_1))</td>
<td>host name ((i_1)), software name ((i_2)), execution mode ((i_3)), mode parameters ((i_4)), start signal ((i_3)), report address ((i_4))</td>
<td>report ((o_1))</td>
<td>testbed installed one version installed for normal or testing mode 2 or more versions installed for parallel or selective mode</td>
</tr>
<tr>
<td>start ((s_1))</td>
<td>host name ((i_1)), software name ((i_2)), start signal ((i_3)), report address ((i_4))</td>
<td>report ((o_1))</td>
<td>execution parameters set</td>
</tr>
<tr>
<td>stop ((s_1))</td>
<td>host name ((i_1)), software name ((i_2)), start signal ((i_3)), report address ((i_4)), trace delivery address ((i_5))</td>
<td>report ((o_1))</td>
<td>trace ((o_2)), software started</td>
</tr>
<tr>
<td>uninstall ((s_1))</td>
<td>host name ((i_1)), tested or version name ((i_2)), start signal ((i_3)), report address ((i_4))</td>
<td>report ((o_1))</td>
<td>software installed software stopped if it is tested then it must not have versions installed</td>
</tr>
<tr>
<td>trace start ((s_1))</td>
<td>host name ((i_1)), software name ((i_2)), interactive trace ((i_3)), start signal ((i_4)), report address ((i_5))</td>
<td>report ((o_1))</td>
<td>software installed software started</td>
</tr>
<tr>
<td>trace stop ((s_1))</td>
<td>host name ((i_1)), software name ((i_2)), start signal ((i_3)), report address ((i_4)), trace data ((o_2))</td>
<td>report ((o_1))</td>
<td>trace started</td>
</tr>
<tr>
<td>plugging ((s_1))</td>
<td>user data ((i_1))</td>
<td>report ((o_1))</td>
<td>report ((o_1))</td>
</tr>
</tbody>
</table>

Table 1. Software Operation Tasks
Tesbed will be explained in detail in Sec. 4.2) should also be installed.

Table 1 includes two tasks which are not software operations: the planning task ($s_9$) and the report task ($s_{10}$). The planning task is responsible for obtaining getti

4.2. **MA–RMS architecture**

Agents are capable of autonomous actions which allow them to perform operations on network nodes by themselves. For this purpose, we have designed the MA–RMS.

The MA–RMS is an agent–based framework user for remote control over software on remote locations. There is no need for the administrator to be physically present at the actual location of the system being maintained. With centralized location management, it is possible to work on several remote systems at the same time. This saves both time and money since operations can be performed simultaneously from an office.

The basic MA–RMS concept is shown in Fig. 11. It consists of two main components: the MA–RMS Console and the MA–RMS Maintenance Environment (also called the MA–RMS Core). The MA–RMS Console has two components: a Management Console Agent and an MA–RMS GUI. An MA–RMS user uses the MA–RMS GUI to define operations which have to be performed at remote locations. It can also be used to track changes on remote systems. After the user starts execution of the defined tasks, they are dispatched to the Management Console Agent. This agent is a centralized agent used to supervise the process of software maintenance. Based on differences between the current state of the remote systems and the desired state of the systems, it generates a set of operations which need to be executed in order to get the system into the desired state. After generating the operations, they are distributed to a set of multi–operation agents according to a sub–team–coordination plan. This plan defines how to distribute operations among agents and

![Fig. 11. the basic MA–RMS concept](image_url)
Verification of the mobile agent network simulator

is used to reduce network traffic and node load. Deciding which plan to use depends on network topology and node types.

There is also an automated version of the MA–RMS Console, referred to as AutoRMS. This version provides interfaces which allow external applications to schedule maintenance operations at the remote location. The application has to define the operations to be performed, software components, and a list of remote locations.

The Maintenance Environment must be preinstalled on remote network nodes in order for them to be managed. It also handles a local database which stores data regarding the installed software. It consists of four agents, where each agent is responsible for a certain set of functionalities. This increases parallelism and reduces the complexity of agents. The agents and their functionalities are described below:

- The Management Database Agent stores information regarding the installed softwares, their status and execution parameters. It registers and de-registers MA–RMS Consoles and notifies them when software status changes;
- The Cooperation Layer Agent is used for software migration. It is responsible for taking software from multi-operation agents and storing it in a local file repository;
- The Installation Dock Agent supports software installation and un-installation. During the software installation process all the necessary components are fetched from a local repository and installed within the Maintenance Environment according to the installation scripts. Furthermore, information regarding the software is sent to the Management Database Agent;
- The Application Handler Agent is responsible for software starting and stopping as well as trace activation and deactivation.

Fig. 11 shows only one MA–RMS Console and one Maintenance Environment but in reality there can be any number of them.

Each software component must be adopted to the MA–RMS execution model before it can be deployed on a remote network node. MA–RMS software consists of two components: the Application Testbed and the Application Version. The Application Testbed is an interface between the Application Version and the Maintenance Environment. It conveys all data from the outer environment to the Application Version and vice versa. It implements three APIs:

- The ResourceManager API which represents the input/output layer and provides a connection to system resources. Communication between software and the environment is handled by the Resource Manager;
- The Version Handler, which is responsible for starting and stopping software versions;
- The Trace Support API, which is used for collecting, modeling and delivering trace data to the Maintenance Environment;

The Application Version provides the actual functionality of the application.
The MA–RMS is organized as a team-oriented hybrid knowledge-based concept. It includes a master and team agents with knowledge shared between them. The Management Console agent is the master agent who has the same knowledge as master agents in the CKC: its own capabilities and team capabilities. Team agents have knowledge such as that possessed by agents in the DKC: its own capabilities and situation-specific knowledge. After starting the team, the master agent no longer cares about unexpected events during execution. Namely, team agents should know how to handle such events using their situation-specific knowledge. Slave agents in the MA–RMS are implemented as multi-operation agents.

4.3. Multi-Operation Agents

Software operations and all management actions in the MA–RMS are performed by mobile agents. They are responsible for software migration, installation, starting, stopping un-installation, running and tracing. Instead of sending messages between the management station and the remote system, tasks are decomposed into operations which are distributed among multi-operation agents. Agents are equipped with the required knowledge (e.g. communication protocol), data (e.g. software packages which have to be installed) and access rights to the remote system. Agents then migrate to the remote location and perform operations locally.

There are two ways of performing maintenance operations. The first is by using one universal agent capable of executing all possible operations the system supports. In complex systems, these agents can become too heavy and complex due to a large number of maintenance actions. Alternatively, a set of specialized cooperating and communicating agents, responsible for only a subset of possible operations, could be used. In the MA–RMS, we have adopted a hybrid principle combining the two.

Multi-operation agents (MOA) are generic carriers that can handle multiple operations. In general, these agents are only carriers, while the logic behind performing the actual operations is loaded during the process of distributing operations among multi-operation agents. This is performed by the Management Console. This approach is used since it is more flexible, allows easier system modeling and provides better control over the sequence in which operations will be executed. If needed, it is also possible to generate operations in such a way that we create specialized agents, universal agents or any combinations of the two.

5. Laboratory Measurements

The described MAN simulator was used in experiments which simulate the execution of operations in the MA–RMS system. Conducting this experiment was necessary to be able to compare with the results obtained by the actual multi-agent system and, thus, validate the results obtained by the simulator. If the results from this experiment were comparable with those obtained by the simulator, this would prove that the simulator could successfully be used to model the behavior of multi-agent systems based on MAN.
5.1. Laboratory configuration

Fig. 12 shows the configuration of the laboratory where the experiments were performed. Nine PCs were used: eight were used for hosting the MA–RMS servers and one hosted the ScenarioExecutionAgent and the AutoRMS station. Configuration of the PCs can be seen at Table 2.

The various testing scenarios differed with respect to three parameters: the number of operations to be performed (ranging from one to eight software installation operations), the number of PC on which these operations were to be performed (ranging between one to eight PCs) and the network bandwidth (512 Kbit/s, 1 Mbit/s and 10 Mbit/s networks) making a total of 192 measurements. Such a large number of measurements required some automation to reduce the time needed to perform them and the possibility of errors which can be caused by human interaction. Automation of the first two parameters was performed by the ScenarioExecutionAgent. It was responsible for scheduling the installation operations and keeping track of the results of the experiments. The network bandwidth parameter was

<table>
<thead>
<tr>
<th>Configuration type</th>
<th>Configuration value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC Model</td>
<td>Dell OptiPlex 170L</td>
</tr>
<tr>
<td>Processor</td>
<td>Intel Celeron 2.66 GHz</td>
</tr>
<tr>
<td>Physical Memory</td>
<td>512MB</td>
</tr>
<tr>
<td>Operating System</td>
<td>Windows XP Professional</td>
</tr>
<tr>
<td>Java Version</td>
<td>Sun JDK 1.5.0_09_b01</td>
</tr>
<tr>
<td>Jade Version</td>
<td>3.3</td>
</tr>
</tbody>
</table>
changed manually since we didn’t find any way to control it using an agent. To simulate different networks we used network bandwidth limiters on all the PCs included in the experiment.

A single scenario measurement was performed in the following way:

- The ScenarioExecutionAgent would read the scenario parameters from the XML configuration file (the number and the location of the software, and the number and the IP address of the PCs where the MA-RMS servers were installed);
- After gathering the parameters of the experiment, the agent would generate an installation request and send it to the AutoRMS station;
- The AutoRMS station would receive and process the request. After processing the request, the station would generate the operations needed to perform the installation request (every installation request consists of several sub-operations);
- The corresponding operations would be scheduled for execution by the Multi-OperationAgents (MOA) according to the scheduling algorithm used;
- The MOA would migrate to the MA-RMS server/s and perform operations;
- Upon completion of the scenario, the AutoRMS would send a notification message to the ScenarioExecutionAgent with the time needed to perform it;
- If there were additional scenarios left, this process would be repeated (There was a total of 192 scenarios).

The AutoRMS station was responsible for measuring the time needed to perform a scenario. Time measurement was started before scheduling the operations for execution and stopped upon operation completion.

5.2. Simulator parameters measurement

Before simulating the multi-agent system using the simulator, we measured the necessary simulation parameters on the real system. The parameters measured were as follows:

- The time needed to perform a single migrate operation;
- The time needed to perform a single installation and configuration operation;
- Traffic generated while migrating the MOA;
- Traffic generated while migrating the MOA with one migrate operation;
- Traffic generated while migrating the MOA with one installation and configuration operation.

To measure the time needed to perform operations, a modification of the MOA was required. Agents were modified with a timer that calculated the elapsed time. The timer was initiated before the operation was scheduled for execution by the agent and stopped after the agent received notification from the MA-RMS server that its request was completed.

Traffic generated by the migration process was measured using the Ethereal Network Protocol Analyzer. During a scenario, this tool would capture the network
traffic on the PCs’ network interface. The traffic generated was in the form of a Java
RMI (Java Remote Network Invocation) stream, used by the Jade agent platform
to migrate agents between two PCs. The packet analyzer was then used to calculate
the stream size.

The traffic parameters required to run the simulator were calculated as follows:

\[ S_{mo} = S_{amo} - S_a \]  
\[ S_m = S_a + N \cdot S_{mo} \]  
\[ S_{io} = S_{aio} - S_a \]  
\[ S_i = S_a + N \cdot S_{io} \]

where \( S_{amo} \) is the traffic generated by the MOA with one migrate operation, \( S_a \)
is the traffic generated by the MOA without any operations, \( S_{mo} \) is the size of the
migrate operation, \( S_m \) is the size of the MOA with multiple migration operations, \( N \)
is the number of software components, \( S_{aio} \) is the traffic generated by the MOA with
one install operation, \( S_{io} \) is the size of the installation operation and \( S_i \) represents
the size of the MOA with multiple installation operations. The time needed to
migrate a MOA depends on the size of the agent and is calculated as follows:

\[ t = \frac{S}{B} \]

where \( t \) represents the time needed to migrate MOA, \( S \) is the agent size and \( B \)
represents the network bandwidth. All values are in bytes. The values obtained for
the size of the agent and the time needed to perform operations are shown in Table
3 where \( t_m \) represents the time needed to perform the migrate testbed operation,
\( t_{mv} \) is the migrate version operation, \( t_{it} \) is the install testbed operation, \( t_{iv} \) is the
install version operation and \( t_e \) is the time needed to configure the application.

5.3. Comparison of results

In this section, the simulation results obtained by the MAN simulator (with and
without network components) are compared with the results obtained by the MA–
RMS. In Fig. 13, 14 and 15 the x-axis represents the number of PCs on which the

<table>
<thead>
<tr>
<th>Table 3. Measured parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter name</td>
</tr>
<tr>
<td>----------------</td>
</tr>
<tr>
<td>( S_a )</td>
</tr>
<tr>
<td>( S_{amo} )</td>
</tr>
<tr>
<td>( S_{aio} )</td>
</tr>
<tr>
<td>( t_m )</td>
</tr>
</tbody>
</table>
software is to be installed in the MA–RMS Maintenance Environment, the y–axis represents the number of software installation requests and the z–axis shows the time needed to complete a single scenario. Accuracy of the simulator is shown in Fig. 16 and 17 and is calculated by the following formula

\[ RE = \left( \frac{t_{rms} - t_{man}}{t_{rms}} \right) \times 100 \]

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. Two versions of the simulator were analysed:

Fig. 13. Total execution time for 512 kbit/s

Fig. 14. Total execution time for 1 Mbit/s
without the network components and with the network components.

The results show that the total execution times of both the MAN simulator and the MA–RMS increase linearly with the number of software installation requests and Maintenance Environments. It also shows that the MAN simulator with network components shows better simulation results when compared with those obtained by the MA–RMS. The RE’s are 7.9% (for 512 kbit/s network bandwidth), 7.6% (for 1 Mbit/s network bandwidth) and 10.5% (for 10 Mbit/s network bandwidth).

Fig. 15. Total execution time for 10 Mbit/s

Fig. 16. RE for MAN without network components
The MAN simulator without network components has $RE$ of 16.6\% (for 512 kbit/s network bandwidth), 18.8\% (for 1 Mbit/s network bandwidth) and 16.9\% (for 10 Mbit/s network bandwidth).

The only anomaly in the results occurs for scenarios with few software components and small number of Maintenance Environments. In these scenarios, the difference increases up to the a maximum value of 81\% for the worst case scenario (i.e. the scenario with one software component, one Maintenance Environment and a network bandwidth of 10 Mbit/s). The cause of this anomaly is still unknown. It could be caused by a component(s) of the MA–RMS, the agent platform, or it could be caused by the some initialization process. Since the total execution time in these scenarios is quite small its effect can have a significant influence on the results. The accuracy of the simulator increases with the number of software components and Maintenance Environments.

The accuracy of the simulator is even better if we negate the influence of assumptions that were made by the simulator. The simulator does not take into consideration the influence of other PCs in the network and the software installed on them, which can generate traffic on the network interface. Measurements performed in the laboratory indicated that on average, these components generate 80 KB of traffic per minute (not taking into consideration traffic generated by the agent platform while communicating with other components in the agent system) which makes up for 3\% of the total traffic generated by the MA–RMS.

6. Conclusion and Future Work

In this paper, we presented the Mobile Agent Network simulator, a tool for simulating multi–agent systems. The agents in the simulator are based on the MAN formal
Verification of the mobile agent network simulator

model and organised as a team-oriented multi-agent system. The team consists of a master agent and a team of agents, where knowledge of the system is shared between the master and the team. Shared-plans described in the paper enable team formation according to task complexity and the characteristics of the distributed environment where they are to be performed. A detailed description of the simulator is given, including details regarding the simulator core, graph definition and simulation execution. The simulator is verified by comparison with the Multi-Agent Remote Maintenance Shell (MA-RMS) system, a team-oriented knowledge-based system responsible for distributed software management. The laboratory configuration used is described, along with various testing scenarios which differ with respect to the following three parameters: the number of operations to be performed, the number of PCs (nodes) on which these operations will be performed, and the network bandwidth. Two versions of the MAN simulator were compared: one without network components and one with them. The analysis shows that the RE of the MAN simulator without network components is 17.43% on average, while the RE of the MAN simulator with network components is 8.66% on average. As the number of software components and remote locations increases, so does the RE of the MAN simulator (it is below 3% for scenarios with eight software components and remote locations).

In the future we plan to integrate the simulator into the MASON framework in order to improve RE.

Acknowledgements

This work was carried out within the research project 036-0362027-1639 "Content Delivery and Mobility of Users and Services in New Generation Networks", supported by the Ministry of Science, Education and Sports of the Republic of Croatia.

References

Mario Kusek et al.


