AAMAS 2010
TORONTO

The 9th International Conference on
Autonomous Agents and Multiagent Systems
May 10-14, 2010
Toronto, Canada

Workshop 20

The First International Workshop on Infrastructure and Tools for Multiagent Systems

ITMAS 2010

Editors:
Wiebe van der Hoek
Gal A. Kaminka
Yves Lespérance
Michael Luck
Sandip Sen

FIRST INTERNATIONAL WORKSHOP ON INFRASTRUCTURES AND TOOLS FOR MULTIAGENT SYSTEMS

ITMAS2010

Workshop Notes

Vicent Botti
Ana Garcia-Fornés
Jomi F. Hübner
Andrea Omicini
Juan A. Rodriguez-Aguilar

May, 10 2010

at AAMAS 2010
Toronto, Canada

Workshop Organization

Programme Chairs

Vicent Botti
Ana Garcia-Fornes
Jomi F. Hubner
Andrea Omicini
Juan A. Rodriguez-Aguilar

Programme Committee

Juan M. Alberola
Makoto Amamiya
Matteo Baldoni
Fabio Bellifemine
Juan A. Botia
Frances Brazier
Nadia Erdogan
Agustin Espinosa
Kamalakar Karlapalem
Yasuhiko Kitamura
Abder Koukam
Michal Laclavik
Pavlos Moraitis
Sascha Ossowski
Marcin Paprzycki
Michal Pechoucek
Murat Sensoy
Michael Shumacher
Carles Sierra
Jose Such
Pavel Vrba
Danny Weyns

External Reviewers

Antonio Barella
Estefania Argente
Nancy Ruiz
Nikolaos Spanoudakis
Ruben Ortiz

Prologue

Vicent Botti[1], Ana Garcia-Fornes[1], Jomi F. Hübner[2], Andrea Omicini[3], and Juan A. Rodriguez-Aguilar[4]

[1] Departament de Sistemes Informàtics i Computació, Universidat Politècnica de València Camí de Vera s/n. 46022 València, Spain
{vbotti,agarcia}@dsic.upv.es
[2] Department of Automation and Systems Engineering Federal University of Santa Catarina PO Box 476 Florianópolis, SC 88040-900 Brasil
jomi@das.ufsc.br
[3] Department of Electronics, Informatics and Systems (DEIS), Alma Mater Studiorum-Università di Bologna Viale Risorgimento, 2 40136 Bologna, Italy
andrea.omicini@unibo.it
[4] IIIA-CSIC, Campus de la UAB, E-08193 Bellaterra, Catalonia, Spain
jar@iiia.csic.es

INTRODUCTION

ITMAS aims at bringing together leading researchers from both academia and industry to discuss issues on the design and implementation of infrastructures and tools for Multiagent Systems. When developing applications based on Multiagent Systems, developers and users demand infrastructures and tools which support essential features in Multiagent Systems (such as agent organizations, mobility, etc.) and facilitate the system design, management, execution and evaluation. Agent infrastructures are usually built using other technologies such as grid systems, service-oriented architectures, P2P networks, etc. In this sense, the integration and interoperability of such technologies in Multiagent Systems is also a challenging issue in the area of both tools and infrastructures for Multiagent Systems. A long term goal is the industrial development of infrastructures for building highly scalable applications comprising pre-existing agents that must be organized or orchestrated.

In order for Multiagent Systems to be included in real domains such as industry, infrastructures and tools for Multiagent Systems should provide efficiency, scalability, security, management, monitorization and other features related to building real applications.

Table of Contents

Invited Talk: A Middleware Architecture for Dynamic Agent Organizations

Danny Weyns

Katholieke Universiteit Leuven
Departement Computerwetenschappen
Celestijnenlaan 200 A
B-3001 Leuven, Belgium
danny.weyns@cs.kuleuven.be

ABSTRACT

Middleware is the software layer that lies between the operating system and the application components. Middleware provides high-level abstractions to support the coordination of distributed software components. Since multi-agent systems are particularly useful for problem domains characterized an inherent distribution of resources, it is clear that any real-world multi-agent system application should deal with the distribution concern. The dynamic interactions and collaborations among agents are usually structured and managed by means of roles and organizations. In existing approaches agents typically have a dual responsibility: on the one hand playing roles within the organization, on the other hand managing the life-cycle of the organization itself, e.g. setting up the organization and managing organization dynamics. Engineering realistic multi-agent systems in which agents encapsulate this dual responsibility is a complex task. In this talk, I present MACODO: Middleware Architecture for COntext-driven Dynamic agent Organizations. The MACODO middleware offers the life-cycle management of dynamic organizations as a reusable service separated from the agents, which makes it easier to understand, design and manage dynamic organizations in multi-agent systems. To conclude, I put forward a number of research challenges in the area of middleware for multi-agent systems.

A Multi-agent Simulation Framework on Small Hadoop Clouds

Kamalakar Karlapalem
Centre for Data Engineering
International Institute of Information Technology
Hyderabad, India
kamal@iiit.ac.in

Prashant Sethia
Centre for Data Engineering
International Institute of Information Technology
Hyderabad, India
prashant.sethia@research.iiit.ac.in

ABSTRACT

In this paper, we explore the benefits and possibilities about the implementation of multi-agents simulation framework on a Hadoop cloud. Scalability, fault tolerance and failure recovery has always been a challenge for a distributed systems application developer . The highly efficient fault tolerant nature of Hadoop, flexibility to include more systems on the fly, efficient load balancing and the platform-independent Java are useful features for development of any distributed simulation. In this paper, we propose a framework for agent simulation environment built on Hadoop cloud. Specifically, we show how agents can be represented, how agent do their computation and communication, and how agents are mapped to data nodes. Further, we demonstrate that even if some of the systems fail in the distributed setup, Hadoop automatically rebalances the work load on remaining systems and the simulation continues. We present some performance results on this environment for few example scenarios.

Keywords

Design, Experimentation, Reliability, Cloud Computing, Fault-tolerance, Failure-resilience

1. INTRODUCTION

Multi-agent simulation is an important research field in today's scenario and analyzing emergent behaviors in such simulations largely depend on the number of agents involved. More the number of agents involved, closer is the result obtained to the real world. Due to large number of agents, time involved in such simulations becomes huge and so we resort to a distributed computing solution. In order to run such simulations, we require a fault-tolerant, fast and easily extensible agent-based simulation framework which can handle a large number of processors. Further, separating implementation of an agent-based framework from the code which tackles hardware failures, will allow a framework developer to concentrate more on synchronization of processes and their run-time optimization rather than their failures.

Hadoop is a promising option in this respect. It takes care of non-functional requirements, like scalability, fault-tolerance, load-balancing, and leaves the framework developer with the problem of framing a solution for the functional requirements of a multi-agent simulation framework. If some systems in the distributed environment fail, simulation does not stop. Hadoop autmomatically rebalances the work load on remaining systems and continues to run the simulation. Further, Hadoop facilitates dynamic addition of new nodes in a running simulation. In this paper, we present a design of an agent-based simulation framework implemented on hadoop cloud. To the best of our knowledge, no multi-agent simulation framework provides the capabilities of rebalancing work load on systems' failures or dynamic addition of new nodes. Our framework, implemented on Hadoop, provides these new features.

2. RELATED WORK

Developing tools for multi-agent simulations has always been an active area of research, with emphasis being laid on different aspects - architecture, scalability, efficiency, fault-tolerance, effectiveness in modelling. A number of frameworks have been developed - Netlogo [11], JADE [12], ZASE [13], DMASF [14], MASON [15] - to name a few. The proposed framework developed on Hadoop provides three major advancements to the current state of art - (i) Dynamic addition of new computing nodes while the simulation is running; (ii) Handling node failures without affecting the ongoing simulation by redistributing the failed tasks on working systems; (iii) Allowing simulations to run on machines running different operating systems. Further, the framework incorporates several optimization techniques [clustering of frequently communicating agents (for reducing inter-processor communication) and caching of results (for improving performance)] that are run on Hadoop cloud.

3. HADOOP ARCHITECTURE AND MAP-REDUCE MODEL

Hadoop is an Apache project which develops open-source software for reliable and scalable distributed computing. It maintains a distributed file system, Hadoop Distributed File System (HDFS) for data storage and processing. Hadoop uses classic *Map-Reduce* programming paradigm to process data. This paradigm easily fits a large number of problems. Hadoop consists of a single master system (known as *namenode*) along with several slave systems (known as *datanodes*). For failure resilience purposes, it has a *secondary namenode* which replicates the data of *namenode* at regular intervals.

3.1 Hadoop Distributed File System (HDFS)

HDFS is a block-structured file system: individual files are broken into blocks of a fixed size (default size is 64MB). These blocks are stored across a cluster of one or more machines (*datanodes*). A file can be made of several blocks, and they are not necessarily stored on the same machine; the target machines which hold each block are chosen randomly on a block-by-block basis. Thus, access to a file may require the cooperation of multiple machines. If several machines must be involved in the serving of a file, then a file could

be rendered unavailable by the loss of any one of those machines. HDFS solves this problem by replicating each block across a number of machines (3, by default). The metadata information consists of division of the files into blocks and the distribution of these blocks on different *datanodes*. This metadata information is stored on *namenode*. Furthermore, the metadata structures (e.g., the names of files and directories) can be modified by a large number of clients concurrently. It is important that this information is never desynchronized. It is for this reason that all the metadata information is handled by a single machine (*namenode*).

3.2 Map-Reduce Paradigm

The *MapReduce* paradigm transforms a list of *(key, value)* pairs into a list of values. The transformation is done using two functions: *Map* and *Reduce*. *Map* function takes an input *(key, value)* pair and produces a set of intermediate *(key, value)* pairs.

```
map(key1,value1) -> list<(key2,value2)>
```

That is, for an input it returns a list containing zero or more *(key, value)* pairs. The *Map* output can have a different *key* from the input and it can even have multiple entries with the same *key*. The *MapReduce* framework sorts the *Map* output according to intermediate *key* and groups together all intermediate *values* associated with the same intermediate *key*. If the amount of intermediate data is too large to fit in memory, an external sort is used. The *Reduce* function accepts an intermediate *key* and a set of corresponding *values* for that *key*. Typically just zero or one output *value* is produced per *Reduce* invocation.

```
reduce(key2, list<value2>) -> list<value3>
```

The intermediate values are supplied to the *Reduce* function via an iterator. This allows handling lists of values that are too large to fit in memory. The *MapReduce* framework calls the *Reduce* function once for each unique *key* in sorted order. Due to this the final output list generated by the framework is sorted according to the *key* of *Reduce* function.

For example, consider the standard problem of counting the number of occurrences of each word in a large collection of documents. This problem can be solved by using the *Map* and *Reduce* function in following form:

```
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
      EmitIntermediate(w, "1");

reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
      result += ParseInt(v);
  Emit(AsString(result));
```

The *Map* function emits each word along with an associated count of occurrences (just '1' in this simple example). The *Reduce* function sums together all counts emitted for a particular word.

3.3 MapReduce Job Execution by Hadoop

In Hadoop terminology, by 'job' we mean execution of a Mapper and Reducer across a data set and by 'task' we mean an execution of a *Mapper* or a *Reducer* on a slice of data. Hadoop distributes the *Map* invocations across multiple machines by automatically partitioning the input data into a set of M independent splits. These input splits are processed in parallel by M *Mapper* tasks on different machines. Hadoop invokes a *RecordReader* method on the input split to read one record per line until the entire input split has been consumed. Each invocation of the *RecordReader* leads to another call to the *Map* function of the *Mapper*. *Reduce* invocations are distributed by partitioning the intermediate key space into R pieces using a partitioning function. One more thing to mention here is that some problems require a series of *MapReduce* steps to accomplish their goals.

```
Map1 -> Reduce1 -> Map2 -> Reduce2 -> Map3...
```

Hadoop supports this by allowing us to chain *MapReduce* jobs by writing multiple driver methods, one for each job, using *ChainMapper* classes.

The *namenode* is responsible for scheduling various *MapReduce* tasks on different *datanodes*. First the *Map* tasks are scheduled and then the *Reduce* tasks. It gets periodic updates from the *datanodes* about the work-load on each. It computes the average time taken by *MapReduce* tasks on each *datanode* and then does the distribution of tasks in a way that the faster *datanodes* get more number of tasks to execute and the slower ones get less number of tasks to execute.

4. MULTI-AGENT SIMULATION FRAMEWORK ON HADOOP

Each agent has three essential properties: a unique *identifier*, a *time-stamp* associated with the state of that agent and *type* of agent, where identifiers and time-stamps (current time) are generated by the framework itself, whereas agent-type is provided by the user. User can specify additional properties. State of an agent at a particular timestamp refers to the set of its property-values at that timestamp. Likewise, an update in the state of an agent refers to changes in these property-values. User needs to provide an *update-agent-state* function for each type of agent, which is a programmatic model for incorporating agent strategy.

Hadoop requires problems to be solved using *MapReduce* model. This constraint is required in order to make the scheduling of *MapReduce* tasks (done by Hadoop) independent of the problem being solved. Further, by default Hadoop invokes one *Map* task for each input file. Therefore, we model a multi-agent simulation as a series of *MapReduce* jobs with each job representing one iteration in the simulation and we model each agent as a separate input file to these jobs. This leads a particular job to invoke multiple *Map* tasks, one for each agent, executing in parallel. The function for updating the state of each agent, *update-agent-state*, is written as *Map* task. *Reduce* tasks are responsible for writing the data back into the file associated with each agent.

Each agent-state is modelled as a separate flat file on HDFS with file name being the agent identifier. This file contains t most recent states of the agent, where t is a user specified parameter (specified through *getIterationParameters()* in *CreateEnviron* class desribed later), each state being distinguished by a timestamp. This file is the input for

map task initiated to update the state of an agent (one map task for each input file/agent). Current state of an agent is the one having most recent timestamp. A separate message file is associated with each agent that stores recent messages received from other agents.

One *MapReduce* task is invoked for each agent.

The framework implements two classes *Agent* and *AgentAPI*. Class *Agent* contains two classes: *Map* and *Reduce* corresponding to the *MapReduce* task.

Framework Supplied Code Classes

```
public class Agent {
   public static class Map extends MapReduceBase
        implements Mapper {
        public void map(key, value,
          OutputCollector<key, value>);
   }
   public static class Reduce extends MapReduceBase
        implements Reducer {
        public void reduce(key, Iterator<value>,
          OutputCollector<key, value>);
   }
   public static void main(String [ ] args) {
        //Create the simulation world.
        CreateEnviron ce = new CreateEnviron();
        iterparams = ce.getIterationParameters();
        ce.createWorld();
        //Configure map-reduce jobs.
        //Invoke map-reduce task.
   }
}
public class AgentAPI {
   public void createAgent(map <object,
        object> agent_data);
   map<String,String> getAgents(map<object,
        object> filters);
   void sendMessage(String from_agent_identifier,
        String to_agent_identifier, String message);
}
```

Each iteration in the simulation corresponds to one *MapReduce* job invoked with one *MapReduce* task corresponding to every single agent. In the *map* method of class *Map*, data is read from associated input file of the agent into a Java map<object, object>, say *agent_data*, mapping agent properties to their values. Agent state is updated by execution of user-supplied *Update* method (in the *AgentUserCode* class described later). *Timestamp* is also updated to the current time. All the properties are concatenated as a string, and it is passed as a *value* in the *(key, value)* tuple to the *Reducer*, *key* being the agent identifier. In the reduce method of *Reduce* class, the input properties-concatenated string is written in the corresponding agent flat-file. *Main()* function creates the simulation world with the help of class *CreateEnviron* for which the code is supplied by user. The method *getIterationParameters()* gets the user specified inputs such as number of iterations the simulation is intended to run and the number of most recent timestamps for which the agent data is retained in the flat files. Method *createWorld()* is also supplied by user. It creates agents and initializes the world.

The framework supplies code for the *AgentAPI* class. Method *createAgent(map <object,object>)* creates a flat file in HDFS with name as agent identifier and writes the initial state of agent. Method *getAgents(map <object,object> filters)* returns all those agents which satisfy the given set of filters. Several keywords help in this respect, for example, if all the agents are required, then user can pass agent-identifier as *'ALL'* in filters map. *sendMessage()* function sends a message to another agent. It writes the current timestamp and identifiers of the two agents involved in communication in message files associated with them. Finally the user needs to supply the code for strategy of different types of agents.

Classes for which code is supplied by user

```
public class CreateEnviron {
   public map<object,object> getIterationParameters();
   void createWorld() {
        //map <object, object> agent_data initialized.
        AgentAPI crobj=new AgentAPI();
        crobj.createAgent(agent_data);
        //And so on for any number of agents.
    }
}
public class AgentUserCode {
   map<object, object> Update(map<object,
        object> agent_data) {
        switch (agent_data["type"]):
          case "type1":
          //User code for update.
          case "type2":
          ....
    }
   void Shape(String agent_identifier) {
        switch (agent_identifier):
          case "type1":
          //User code for rendering shape.
          case "type2":
          ....
   }
}
```

5. HANDLING FAILURES

System failures are a common case when number of systems involved are large. Information about the *namenode* and *secondary namenode* is present on all the *datanodes*. The *namenode* sends a heart-beat message to *datanodes* at regular intervals (by default 600 seconds; can be configured by user). Each *datanode* sends an acknowledgement message along with the information regarding the status of various tasks running on it. This information includes the number of completed *MapReduce* tasks (for the current *MapReduce* job) after the last heart-beat message received and the total time taken to complete them. It also includes number of active *MapReduce* tasks and number of *MapReduce* tasks in queue, waiting to be scheduled.

When a particular *Map* (or *Reduce*) task fails (and in cases when a slower *Map* task is becoming a bottleneck for rest of the processes), Hadoop spawns a new process to carry out its job, and may also use idle processes to do its task (the ones which have completed their *Map/Reduce task*). When one of the several processes spawned to complete the failed task finishes it, rest of them are aborted (*Speculative execution*). Thus, the simulation enters into next iteration only when all *Map* tasks in the current iteration are completed. Hence *Map* tasks is used for updating an agent's state in a particular iteration. We address the common failure cases and how they are being handled.

5.1 Namenode Failure

If the *datanodes* do not receive heartbeat message from the *namenode* for more than two time intervals (1200 seconds), then the *namenode* is considered to be failed. The *namenode* data has already been replicated on *secondary namenode* at regular intervals. Therefore, failure of *namenode* does not cause any loss of data. *Datanodes* (on detecting namenode failure) at once declare the *secondary namenode* as the new *namenode*. All the responsibilities of *namenode* like job scheduling are now taken up by this new *namenode*. Further, the *datanode* which is physically nearest to the new *namenode* (for faster *namenode* data replication) is selected as the new *secondary namenode*. A special case occurs when the *secondary namenode* turns out to be failed at the instant when *namenode* is detected as failed. In this case, the simulation is aborted.

5.2 Secondary Namenode Failure

The *namenode* detects *secondary namenode* failure if it does not receive any acknowledgement for the heart-beat message. Since, *secondary namenode* only had replica of *namenode* data, therefore its failure is handled simply by electing a new *secondary namenode* from the current *datanodes* (*datanode* physically nearest to the *namenode* is selected).

5.3 Datanode Failure

Namenode detects a *datanode* failure if it does not receive an acknowledgement of the heart-beat message from the *datanode*. Data of each node is replicated on three other nodes in the distributed system. As such, when a *datanode* fails its data can be recovered easily using its replica. However, it might be running several *MapReduce* tasks when it failed. These *MapReduce* tasks need to be rescheduled on some other datanode. Two cases arise for a failed *MapReduce* task: (i) failure occurred while running the *Map* task; (ii) failure occurred while running the *Reduce* task. When the *datanode* fails while running its *Map* task, the entire *MapReduce* task needs to rescheduled on some other datanode and the complete task needs to be done again. If a *datanode* failure occurs while running a *Reduce* task, then optimally the *Map* task should not be redone. For achieving this, output of *Map* task is replicated (as soon as it is finished) on those *datanodes* which contain the data replicas for the failed *datanode*. Thus, if a *datanode* fails when it is running *Reduce* task, then only the *Reduce* task is rescheduled and redone on some other node.

Thus, even if some of the machines fail in the Hadoop cluster, the simulation does not stop and carries on.

6. DYNAMIC ADDITION OF NEW NODES

Namenode maintains a file containing the details of IP addresses of different machines (*datanodes*). It sends heartbeat messages to the systems mentioned in this file. If a new system needs to be added in the Hadoop cluster, then information about it simply needs to be added in this file. When the *namenode* finds a new entry in this file, it immediately grants access to HDFS to the new *datanode* and invokes *MapReduce* tasks on this system, rebalancing the total work-load. Since, heart-beat messages are sent every 600 seconds, therefore the newly added *datanode* can be idle at most for this period. Further, the simulation need not be stopped for achieving this addition. This feature of dynamically adding new nodes is not provided by any other multi-agent simulation framework to the best of our knowledge.

7. IMPLEMENTATION ISSUES

The above model faces run-time challenges which needs to be addressed. When number of agents becomes large (of the order 10^7 agents on 100 machines), overhead due to generation of large number of *MapReduce* tasks becomes huge. Further, the way in which the agents are distributed on different datanodes may affect the run-time adversely. We present below the challenges faced with above model and workarounds for the same.

7.1 Agent Communication

Agent simulation requires communication between different agents. In the present model, agent communication occurs by fetching the state of other agent from their corresponding agent files which reside on different datanodes. This can be a time consuming factor in such simulations. Hence, the number of accesses to files residing on remote datanodes need to be reduced. This is achieved by placing the agents which communicate with each other frequently on the same data node. We framed the following algorithm for achieving this purpose.

7.1.1 Agent Clustering algorithm based on agent-communication

Given K sites, agents a_1, a_2, ..., a_N, and communication statistics between these agents, the problem is to redistribute the agents on these sites in such a manner that communication between agents on same site (intra-site communication) is maximized and that between agents on different sites (inter-site communication) is minimized. Order of the solution needs to be *O(N)* as the number of agents involved is very large.

ALGORITHM : Greedy Agent-Redistribution.

1. Distribute the N agents randomly on K sites, and carry out one iteration of the simulation.
2. For every agent a_i, on each site j,
   (a) Compute the list of agents with which a_i communicated using the message file associated with a_i. Call a_i as the representative agent.
   (b) Map each agent a_k in a_i*'s* list to a_i only if *id-value*(a_i) <= *id-value*(a_k). Denote this map table as R_j. Also, map agent a_i to itself.
3. Combine map-tables R_j*'s* from different sites into a single map table R using the following update rule :

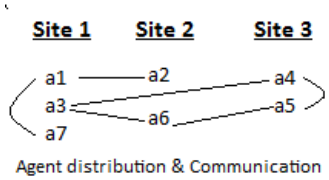
```
if (Rm(ak) < Rn(ak))
    then, R(ak)=Rm(ak)
else,
    R(ak)=Rn(ak)
```

4. Let $R(a_k) = a_l$. Then, do $R(a_k) = R(a_l)$. Do this update for all the entries in table-map R.
5. Form groups of agents in such a way that, all agents a_i in the same group have the same value for $R(a_i)$.

Step 3 in the algorithm is required, because same agent a_i may communicate with several agents on different sites and hence having different values for $R_j(a_i)$. Next, consider a case when agent a_i is occuring on two sites m and n. On

site m, a_i is a representative agent and having $R_m(a_i) = a_i$. On site n, it is not a representative agent and has $R_n(a_i)$ lower than a_i. In such a case, $R(a_i) = R_n(a_i)$. All the elements which initially mapped to a_i on site m have to be re-mapped to $R_n(a_i)$. This justifies Step 4 of the above algorithm.

As an example, let the distribution of agents and communication links between them be as shown in figure below.



Agent distribution & Communication

Step 2 (corresponding to the algorithm) : $R_1(a_1) = a_1$; $R_1(a_3) = a_3$; $R_1(a_7) = a_1$; $R_2(a_2) = a_1$; $\mathbf{R_2(a_6) = a_3}$; $\mathbf{R_3(a_6) = a_5}$; $R_3(a_4) = a_3$; $R_3(a_5) = a_4$

Step 3 : $R(a_1) = a_1$; $R(a_3) = a_3$; $R(a_7) = a_1$; $R(a_2) = a_1$; $\mathbf{R(a_6) = a_3}$; $R(a_4) = a_3$; $\mathbf{R(a_5) = a_4}$

Step 4 : $R(a_1) = a_1$; $R(a_3) = a_3$; $R(a_7) = a_1$; $R(a_2) = a_1$; $R(a_6) = a_3$; $R(a_4) = a_3$; $\mathbf{R(a_5) = a_3}$

Step 5:-

Group 1 : a_1, a_2, a_7

Group 2 : a_3, a_4, a_5, a_6

7.1.2 Execution of Greedy Agent-Redistribution Algorithm on Hadoop

Above algorithm has *O(M/S)* complexity, where M is the number of unique communication links between different agents and S is the number of sites. Since we already have a Hadoop cloud setup, we use this cloud to run this algorithm too (apart from the agent simulation code) and compute the clusters. So, we reframed the algorithm into a *Chained MapReduce* model and executed it on Hadoop. Execution is carried out in two chained *MapReduce* jobs. Output of first *MapReduce* job becomes the input to the *Map* phase of second *MapReduce* job. Finally, clusters of agents are obtained as output from the second *MapReduce* job. Step 1 in the above algorithm corresponds to *MAP-1*, Step 2 corresponds to *REDUCE-1*. Further, Step 3 is executed as *MAP-2* and Step 4 as *REDUCE-2*.

In MAP-1 phase:.

Input - Agent and the list of agents, it communicated with. This is represented as a single line of numbers : x_1, x_2,..., and x_k, where x_1 represents the identifier of current agent under consideration, and the following numbers are identifiers of agents who communicated with the current agent. The input consists of several such lines, one for each agent in the simulation. The input split and load balancing is done by hadoop itself.

Output - (*Key, Value*) pairs (x_i, x_1) if $x_1 <= x_i$.

In REDUCE-1 phase:.

Input - The output from *MAP-1*.

Output - Different values corresponding to the same key are brought together in a list by Hadoop. Let Val_{min} denote minimum of these values. Reduce this list of values to a single value, Val_{min}. Therefore, output of this phase is the reduced set of *(Key, Value)* pairs. Further, if a pair with same value for *Key* and *Value* occurs (e.g. (2,2)) then (2, Val_{min}) is written in a separate file. This is useful for phase MAP-2. Refer the file containing the latter tuples as *Representative_Maps*.

In MAP-2 phase:.

Input - *(Key, Value)* pairs from *REDUCE-1* output and the file *Representative_Maps*.

Output - For each *Key*, the corresponding *Value* is mapped to *R(Value)* using the *Representative_Maps*, which contains *(Value, R(Value))* pairs. Finally, the pair is reversed. Therefore, the output of the phase is *(R(Value), Key)*.

In REDUCE-2 phase:.

Input - *(Key, Value)* pairs from *MAP-2*.

Output - Values corresponding to same key are grouped together in a list by Hadoop itself. Therefore the final output of the phase is *(Key, List of Values)*. e.g. : If *(*k_1*,* v_1*), (*k_1*,* v_2*), (*k_1*,* v_3*)* were present in the output of *MAP-2*. Then the corresponding output pair will be *(*k_1*, [*v_1*,* v_2*,* v_3*])*.

These are the required clusters.

7.1.3 Allocating Sites to the Agents

Finally the clusters obtained are allocated on different sites such that all agents belonging to the same cluster occur together on same site as far as possible. We follow the below procedure for site allocation.

ALGORITHM : Agent-Allocation.

```
site = 1
for each cluster c
    if (size(c) + allocation(site) < capacity(site))
    then,
       Allocate this site to all agents in cluster c.
    else,
       Allocate this site to maximum possible number
       of agents in cluster c.
       site = site + 1
       Reduce cluster c to unallocated agents.
       Repeat the loop for this c.
    Update allocation(site) appropriately.
```

where, *size(c)* denote the size of cluster *c*, *allocation(site)* gives the current number of agents being allocated to this *site*, and *capacity(site)* gives the maximum number of agents which this *site* can hold.

7.2 Small-Files-Problem

Every file, directory and block in HDFS is represented as an object in the namenode's memory, each of which occupies about 150 bytes. So if we have 10 million agents running then we need about 3GB (assuming each file has one block) of memory for the namenode. Furthermore, HDFS is not geared up to efficiently accessing small files: it is primarily designed for streaming access of large files. Reading through small files normally causes lots of seeks and lots of hopping from datanode to datanode to retrieve each small file, all of which is inefficient. Moreover, *map* tasks usually process a block of input at a time. If the file is very small and there are lots of them, then each map task processes very little input, and there are a lot more map tasks running, each of which imposes extra book-keeping overhead.

An alternative for storing agent data in small flat files, is to store them in an Hbase table.

7.2.1 Hbase Model for Representing Agents

In Hbase, data is logically organized into tables, rows and columns. HBase can be reduced to a 5-tuple model *(Row-Key, Column-Family, Column-Key, Time-stamp, Value)*. Each *row-key* maps to a set of *column-families*, which in turn contains a set of *column-keys*. For each *column-key*, there are several versions of the value stored, distinguished by different *time-stamps*. The keys are typically strings, the *timestamp* is a long and the *value* is an uninterpreted array of bytes. The *column-key* is always preceded by its family and is represented like: *family:key*. Therefore, in this model, to retrieve a single value, the user needs to specify a *row-key*, a *column-key* and a *timestamp*.

When using Hbase, each agent is represented as a row. *Row-keys* are generated by framework. Each row has column familes corresponding to each data type in Java. Attributes of agents are modelled as *column-keys*, with each key belonging to the corresponding data-type column family. *Timestamp* is also associated with each *column-key*. Agent identifier is constant throughout, so one can configure to store only one instance of this attribute. An example is shown below.

RowKey	Col.Family	Col.Key	TimeStamp	Value
1	int	x	t1	40
			t2	50
		y	t1	20
			t2	20
		z	t1	30
			t2	25
	String	agent_id	t2	1232
		agent_type	t1	Good
			t2	Bad
2	int	x	t1	40
			t2	40
....				
....				

Java APIs for Hbase enables one to query it directly. Framework code thus becomes simpler because of this. Also, Hbase can be configured to maintain values in the table for a fixed number of timestamps and hence the framework developer need not code for maintaining flat files.

In Hbase, physically, tables are broken up into row ranges called *regions*. Each row range contains rows from *start-key* (inclusive) to *end-key* (exclusive). A set of regions, sorted appropriately, forms an entire table. Row range is identified by the table name and *start-key*. A separate *MapReduce* task is invoked for a single Hbase region. Now, the problem of optimizing communication between agents reduces to bringing frequently communicating agents in the same region or in regions on same datanode. For this, the developer just needs to change the *row-key* of that agent who is frequently communicating to agents in other regions, and set the value of *row-key* such that the value lies within the row-range of the new destined *region*. Physical re-location of the agent tuple is done by Hbase itself.

7.3 Queries in Agent-State Updates

An agent, to decide its next state, needs to know about the state of other agents. For achieving this we implemented the method *getAgents(filters)* in class *AgentAPI*. Execution of this method is significantly time consuming as it needs to access a large number of files and most of them residing on different datanodes. For overcoming this time consumption, we cache results of recent queries in the HDFS.

It is common that same type of agents issue similar queries. So, if another agent issues a query whose result has already been computed and cached, then we return the cached result to it. Another improvement in this respect would be to find out intersection of the filters specified by different agents, and compute and cache the result of this set (intersection set) of filters. This cached result is then used to find out result for the superset query filters. For example, for the queries $x > 60$, $x > 40$ and $x = 90$, the superset would be $x > 40$. Another example would be, for the queries $x < 40$ *and* $y < 50$, $x > 30$ *and* $y < 60$ *and* $z > 40$, the superset would be $y < 60$. Hbase is really useful in this respect as it allows us to store various attributes along with their data types in a more structured manner. This makes the semantics of queries more clear.

The cached result files are physically replicated on each datanode with the help of *DistributedCache* class of Hadoop. This provides a rapid access of the cached results to the datanodes. Name of the cache file indicates the query, along with the iteration number and timestamp during which it was created. Cache files older than a predefined threshold are regularly deleted.

Adding to the solution for the above problem, we built an index of agents based on the frequent query fields (attributes), and updated this index at regular intervals. This allowed a faster lookup for the potential candidate agents for the query.

8. RESULTS

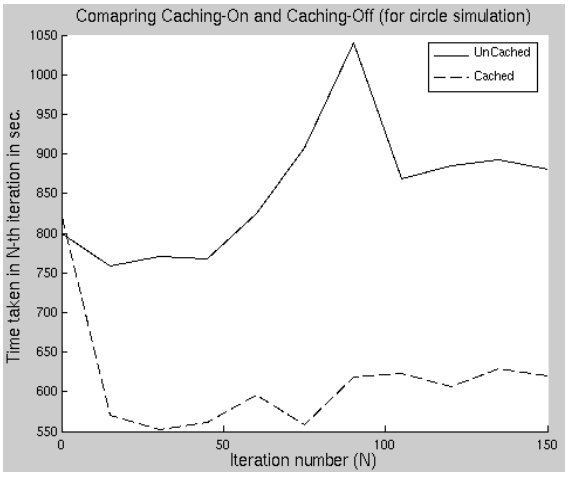
In our experiments, we took various multi-agent simulation problems of diverse nature, so as to test thoroughly the overhead of execution of the two optimization algorithms - clustering of agents and caching of intermediate results - running on top of Hadoop framework. We had set up a Hadoop cloud with three Unix machines, each having 1.67 GHz processor with 1GB RAM. Our experiments involved 20,000 agents distributed on these three machines and interacting with each other for 150 iterations. Some of the interesting results obtained are presented here.

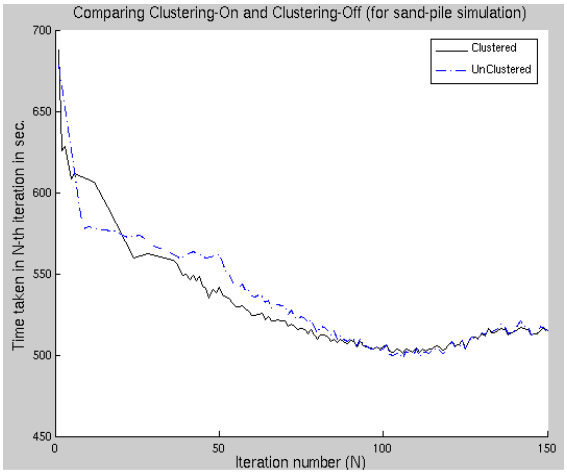
(i) Circle Simulation.

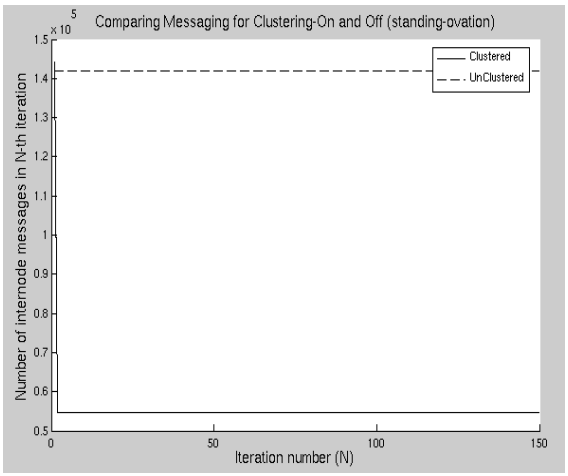
In this problem, agents are scattered randomly on a 2-D plane. Their goal is to position themselves in a circle, and then move around it. The strategy involed computation of arithmetic mean of locations of all the agents. A frequent query is executed to compute this mean to get the locations of all agents. Accessing the agent files on different systems, everytime (once in each agent update function) an agent requested, was avoided by caching locations of all the agents once and then using this cached value for future reference. This cache was updated in every iteration.

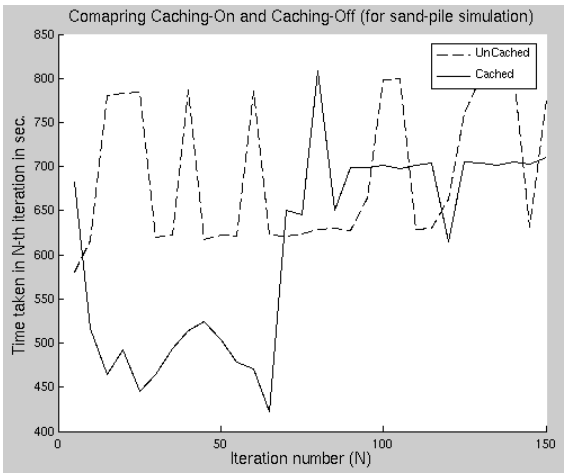
(ii) Standing Ovation Problem.

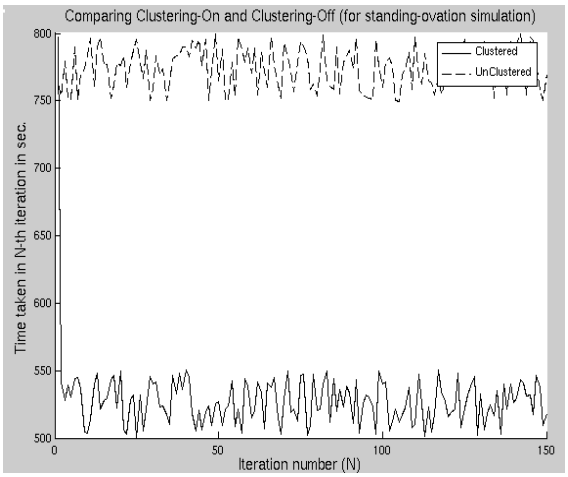
The basic SOP can be stated as: A brilliant economics lecture ends and the audience begins to applaud. The applause builds and tentatively, a few audience members may or may not decide to stand. Does a standing ovation ensue or does the enthusiasm fizzle? The authors present a simulation model for the above problem in [7]. We modelled the auditorium as a 100 X 200 grid, and agents were randomly assigned a seat. Agents in this simulation communicated with almost a constant set of neighbouring agents. Hence

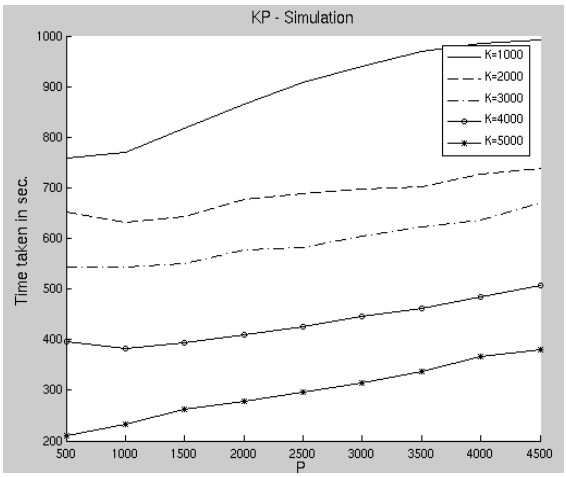

Comapring Caching-On and Caching-Off (for circle simulation)
UnCached
Cached
Time taken in N-th iteration in sec.
Iteration number (N)

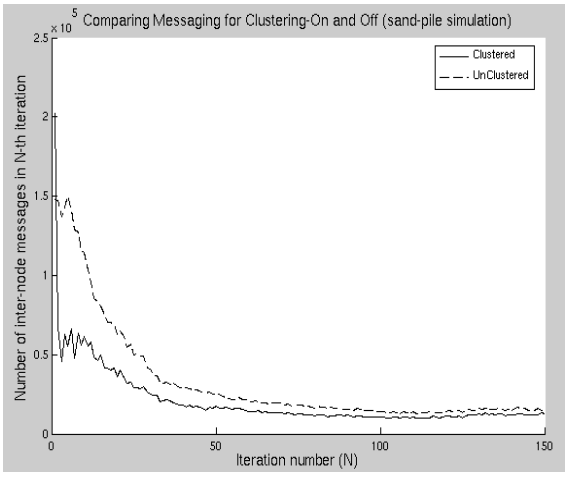

Comparing Clustering-On and Clustering-Off (for sand-pile simulation)
Clustered
UnClustered
Time taken in N-th iteration in sec.
Iteration number (N)

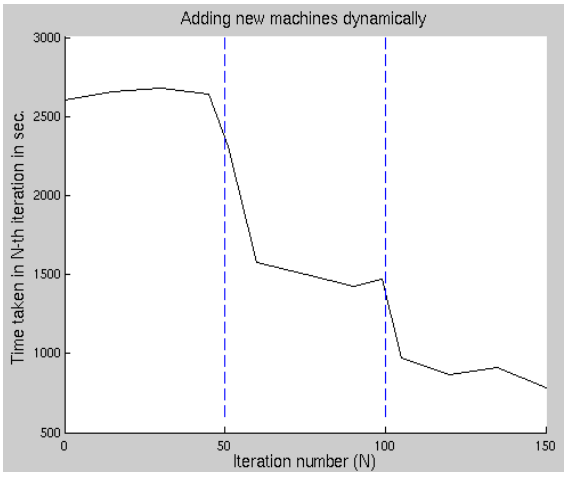

Comparing Messaging for Clustering-On and Off (standing-ovation)
Clustered
UnClustered
Number of internode messages in N-th iteration
Iteration number (N)


Comapring Caching-On and Caching-Off (for sand-pile simulation)
UnCached
Cached
Time taken in N-th iteration in sec.
Iteration number (N)


Comparing Clustering-On and Clustering-Off (for standing-ovation simulation)
Clustered
UnClustered
Time taken in N-th iteration in sec.
Iteration number (N)


KP - Simulation
K=1000
K=2000
K=3000
K=4000
K=5000
Time taken in sec.
P


Comparing Messaging for Clustering-On and Off (sand-pile simulation)
Clustered
UnClustered
Number of inter-node messages in N-th iteration
Iteration number (N)


Adding new machines dynamically
Time taken in N-th iteration in sec.
Iteration number (N)

the clustering algorithm proposed earlier showed marked reduction in number of inter-site messages and showed major improvements in run-time. Time for each iteration was almost halved.

(iii) Sand-pile Simulation.

In this problem, grain particles are dropped from a height into a beaker through a funnel, and they finally settle down in the beaker after colliding with each other and with the walls of beaker and funnel. A detailed description and solution of the problem is given in [8] and for visualization of the problem refer to the link [9].

The queries generated in this problem were not generic enough to give good results on caching intermediate results. Also, the set of agents with which a particular agent interacted, changed too frequently. Hence, movement of agents from one machine to another was also frequent. The results show that even if the inherent nature of a problem did not match the clustering and caching optimizations, execution overhead of such algorithms did not worsen the runtime.

(iv) KP Simulation.

This simulation is done to test messaging efficiency of the framework. Agents are divided into groups with K agents in each group. P is the number of messages sent by each agent on receiving P messages from other agents in the group. Results obtained for different values of K and P are shown.

(v) Dynamic Nodes Addition.

In this simulation, we tested the ability of Hadoop to redistribute agents when new datanodes are dynamically added to the Hadoop cluster. We ran sand-pile simulation with Hadoop cloud consisting of only one datanode. After fifty iterations, we added another datanode. Finally we added a third datanode after 100 iterations. Results obtained show that run-time decreased to half between iteration numbers 50-100 and to almost one-third in the last set of fifty iterations.

9. CONCLUSION

Cloud computing is a recent advancement in the field of solving larger problems. Multi-agent simulations when scaled up to a large number of agents require a potential framework to run them. Hadoop provides a novel framework for running applications involving thousands of nodes and petabytes of data. It allows a developer to focus on agent model and their optimization without getting involved in fault tolerance issues. Extensibility of hardware on which framework is running is made easy by Hadoop, by allowing dynamic addition of new nodes and by allowing heterogeneity between operating systems which the different nodes are running. Therefore, it provides a strong backbone for implementing large scale agent-based simulation framework.

Using cached results is a major optimization in the framework. Developing better heuristics for caching results and to determine appropriate cache sites for faster access of the results are some of the challenging tasks and work is going on in this direction. A faster lookup for agents is achieved by indexing them on frequently queried agent attributes.

Hadoop (in version 0.21.0) is planning to provide an API in which the users can embed their own algorithm for distribution of files on datanodes in the hadoop framework. This will allow us to try several strategies for optimizing inter-node communication.

10. REFERENCES

1. Jeffrey Dean and Sanjay Ghemawat - MapReduce: Simplified Data Processing on Large Clusters. Proceedings of the 6th conference on Symposium on Opearting Systems Design and Implementation - Volume 6, 2004.
2. Steven F. Railsback, Steven L. Lytinen, Stephen K. Jackson - Agent-based Simulation Platforms: Review and Development Recommendations - Society for Computer Simulation International, 2006.
3. Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber - Bigtable: A Distributed Storage System for Structured Data. Seventh Symposium on Operating System Design and Implementation, Seattle, WA, 2006.
4. Hadoop wiki page : [http://wiki.apache.org/hadoop]
5. Some information about Hadoop 0.21 version release : [http://issues.apache.org/jira/browse/HDFS-385]
6. Nuannuan Zong, Feng Gui, Malek Adjouadi - A New Clustering Algorithm of Large Datasets with O(N) Computational Complexity : Proceedings of the 5th International Conference on Intelligent Systems Design and Applications, 2005.
7. John H. Miller, Scott E. Page - The Standing Ovation Problem : Computational modeling in the social sciences, 2004.
8. Laurent Breton, Jean Daniel Zucker, Eric Clement - A multi-agent based simulation of sand piles in a static equilibrium : MABS, Boston, 2001.
9. Visualization of sand piles problem : [http://grmat.imi.pcz.pl]
10. Cloud-computing Wikipedia-page: [http://en.wikipedia.org/wiki/Cloud_computing]
11. Seth Tisue - NetLogo: Design and implementation of a multi-agent modeling environment - International Conference on Complex Systems, Boston, 2004.
12. F Bellifemine, A Poggi, G Rimassa - JADE : A FIPA compliant agent Framework. - PAAM (1999)
13. Gaku Yamamoto, Hideki Tai, Hideyuki Mizuta - A Platform for Massive Agent-Based Simulation and Its Evaluation - AAMAS, 2007.
14. IVA Rao, M Jain, K Karlapalem - Towards Simulating Billions of Agents in Thousands of Seconds - AAMAS, 2007.
15. Luke S, Cioffi-Revilla C, Panait L, Sullivan K, Balan G. - MASON: a multiagent simulation environment - Society for Computer Simulation International, 2005.

An agent-based signal processing in-node environment for real-time human activity monitoring based on wireless body sensor networks

F. Aiello[1], F.L. Bellifemine[2], G. Fortino[1], R. Gravina[1,3], A. Guerrieri[1]

[1] Department of Electronics, Informatics and Systems (DEIS), University of Calabria, Rende (CS), Italy

[2] Telecom Italia, Turin, Italy

[3] Telecom WSN Lab, Berkeley, CA, USA

faiello@si.deis.unical.it, fabioluigi.bellifemine@telecomitalia.it, g.fortino@unical.it, {rgravina,aguerrieri}@deis.unical.it

ABSTRACT

Nowadays wireless body sensor networks (WBSNs) have great potential to enable a broad variety of assisted living applications such as human biophysical/biochemical control and activity monitoring for health care, e-fitness and emergency detection, and emotional recognition for social networking, security and highly-interactive games. It is therefore important to define design methodologies and programming frameworks which enable rapid prototyping of WBSN applications. Several effective application development frameworks have been already proposed for WBSNs based on TinyOS-based sensor platforms, e.g. CodeBlue, SPINE, and Titan. In this paper we present an application of MAPS, an agent framework for wireless sensor networks based on the Java-programmable Sun SPOT sensor platform, for the development of a real-time WBSN-based system for human activity monitoring. The agent-oriented programming abstractions provided by MAPS allow effective and rapid prototyping of the sensor software. In particular, the architecture of the developed system is a typical star-based WBSN composed of a coordinator node and two sensor nodes located respectively on the waist and the thigh of the monitored assisted living. The coordinator is based on an ad-hoc enhancement of the Java-based SPINE coordinator and allows configuring the sensors, receiving their data, and recognizing pre-defined human activities. On the other hand, each sensor node runs a MAPS-based agent that performs sensing of the 3-axial accelerometer sensor, computation of significant features on the acquired data, feature aggregation and transmission to the coordinator. Finally, the experimentation phase of the prototype is also described; it allows evaluating the obtainable monitoring performances and activity recognition accuracy.

Categories and Subject Descriptors

D.2.11 [**Software Architectures**]: Domain-specific architectures. I.2.11 [**Artificial Intelligence**]: Distributed Artificial Intelligence – *Multiagent systems*. C.3 [**Special-purpose and application-based-systems**]: Real-time and embedded systems.

General Terms

Algorithms, Design, Experimentation.

Keywords

Wireless body sensor networks, software agents, real-time human activity monitoring.

1. INTRODUCTION

Wireless Sensor Networks (WSNs) are currently emerging as one of the most disruptive technologies enabling and supporting next generation ubiquitous and pervasive computing scenarios [18]. WSNs are capable of supporting a broad array of high-impact applications in several domains such as disaster/crime prevention, military, environment, logistics, health care, and building/home automation. WSNs applied to the human body are usually called Wireless Body Sensor Networks (WBSNs) [21]. WBSNs are conveying notable attention as real-world applications of WBSNs aim at improving the quality of life of human beings by enabling continuous and real-time non-invasive medical assistance at low cost. Health-care applications where WBSNs could be greatly useful include early detection or prevention of diseases, elderly assistance at home, e-fitness, rehabilitation after surgeries, motion and gestures detection, cognitive and emotional recognition, medical assistance in disaster events, etc.

However, programming WBSN applications is a complex task due to the hard constraints of the wearable devices in terms of limited resources (computing power, memory, and communications) and to the lack of proper and effective software abstractions. To deal with these issues, several software frameworks have been developed such as CodeBlue [12], Titan [10], and SPINE [7]. They aims at decreasing development time and improving interoperability among signal processing intensive applications based on WBSNs. In particular, they basically rely on a star-based network architecture, which is organized into a coordinator node and a set of sensor nodes. Moreover, they are developed in TinyOS at the sensor node side and in Java at the coordinator node.

However, apart from the adoption of effective frameworks, we believe that the exploitation of the agent-oriented programming paradigm to develop WBSN applications could provide more effectiveness as demonstrated by the application of agent technology in several key application domains [6, 11]. In this paper, we therefore propose an agent-oriented approach to develop WBSN applications based on the MAPS framework [1, 2] which enables agent-oriented programming by offering powerful abstractions which allows rapid prototyping of WSN applications on the Sun SPOT sensor platform. The proposed approach is exemplified through the implementation of an agent-based real-time human activity monitoring system. In particular, its architecture is a typical star-based WBSN composed of a

coordinator node and two sensor nodes which are located on the waist and the thigh of the monitored human being, respectively. The coordinator is based on an ad-hoc developed enhancement of the Java-based SPINE coordinator [7, 17] and allows configuring the sensing process, receiving sensed data features, and recognizing pre-defined human activities through a KNN classifier. Each sensor node executes a MAPS-based agent that performs sensing of the 3-axial accelerometer sensor, computation of significant features on the acquired data, feature aggregation and transmission to the coordinator. Finally, the experimentation phase of the developed prototype allows evaluating the obtainable monitoring performances and activity recognition accuracy.

The rest of this paper is organized as follows. Section 2 discusses related work ranging from monolithic WBSN applications to domain-specific frameworks. Section 3 describes the reference architecture for WBSN application from network and functional perspectives. Section 4 presents the proposed agent-oriented approach and the agent-based development of the real-time human activity monitoring system. In Section 5 an evaluation of the developed system is reported. Finally conclusions are drawn and on-going research delineated.

2. RELATED WORK

Most of previous research on WBSN applications has focused on proof-of-concept applications with the aim of demonstrating the feasibility of new context-aware algorithms and techniques, e.g. for the recognition of physical activity through accelerometer sensors or prompt detection of hearth diseases [16, 3, 21]. Moreover such research has considered issues related to power consumption and radio channel usage but taking scarcely into account code reusability and modularity. One of the most relevant attempts to define a general platform able to support various WBSN applications is CodeBlue [12]. CodeBlue is a framework based on TinyOS [20] specifically designed for integrating wireless medical sensor nodes and other devices that could be involved in a disaster response setting. CodeBlue allows such devices to discover each other, report events, and establish communications. CodeBlue relies on a publish/subscribe-based data routing framework in which sensors publish relevant data to a specific channel and end-user devices subscribe to channels of interest. It provides end-user devices with a query interface to retrieve data from previously discovered sensor nodes. While it is possible to select sensor types or physical node identifiers as data sources, configure the data rate and define in-node threshold-based filters to avoid unnecessary data to be transmitted, more sophisticated in-node processing on the sensor data is not supported. A different approach is proposed by Titan [10], which is also implemented in TinyOS. Titan is a middleware for distributed signal processing in WSNs that supports implementation and execution of context recognition algorithms in dynamic WSN environments. Titan represents data processing by a data flow from sensors to recognition result. The data is processed by tasks which implement elementary computations. Tasks and their data flow interconnections define a task network, which runs on the sensor network as a whole. Tasks are mapped onto each sensor node according to the sensors and the processing resources it provides. Titan dynamically reprograms the WSN to exchange context recognition algorithms, handle defective nodes, variations in available processing power, or broken communication links. The architecture of Titan is composed of several software components, which enhance modularity. Although CodeBlue and Titan raise the programming abstraction level by offering general-purpose platforms and middlewares for effectively developing signal processing applications in WBSNs, they are sometimes too general for providing efficient solutions in specific application domains. Thus, domain-specific frameworks [3, 30] have been proposed which are positioned in the middle between application-specific code and middleware approaches. They specifically address and standardize the core challenges of WSN design within a particular application domain. While providing high efficiency, such frameworks allow for a more effective development of customized applications with little or no additional hardware configuration and with the provision of high-level programming abstractions tailored for the reference application domain. An example of such approach is represented by the SPINE framework [7, 17]. SPINE provides libraries of protocols, utilities and processing functions, and a lightweight Java API that can be used by local and remote applications to manage the sensor nodes or issue service requests. By providing these abstractions and libraries, that are common to most signal processing algorithms used in WBSNs for sensor data analysis and classification, SPINE also provides flexibility in the allocation of tasks among the WBSN nodes and allows the exploitation of implementation tradeoffs. Currently SPINE is implemented for several sensor platforms based on TinyOS [20] and Z-Stack [22] by using the programming paradigms offered by such platforms (event and component-based programming in TinyOS and C programming in Z-Stack) and is being effectively applied to the development of applications in the health care domain [8]. In this paper we propose an agent-oriented approach which borrows the basic features characterizing the domain specific framework approach, particularly SPINE, with the aim to provide more programming effectiveness as demonstrated by the application of agent technology in several key application domains [6, 11].

3. A REFERENCE SYSTEM FOR WBSN-BASED SIGNAL PROCESSING

The network architecture of the reference WBSN system for signal processing is organized into multiple sensor nodes and one coordinator node (see Figure 1). The coordinator manages the network, collects, stores and analyzes the data received from the sensor nodes, and also can act as a gateway to connect the WBSN with wide area networks (e.g. Internet) for remote data access. Sensor nodes measure local physical parameters and send raw or pre-processed data to the coordinator. In this system, sensor nodes only communicate with the coordinator according to the star network topology. However, the system could be easily extended to support also direct and multi-hop communications among sensor nodes. In the reference architecture a sensor node is associated with a single coordinator; a possible extension is to allow sensor nodes to be associated and communicate with multiple coordinators. A scenario where such architecture could be used is when a human wearing sensor nodes moves across locations; in this case such sensors should connect to a different coordinator at each different location. The software architecture of the system consists of two main components, implemented, respectively, on the coordinator (e.g. a PC or a smart-phone) and on the WBSN sensor nodes. Figure 2 shows a schema of the architecture from a functional point of view.

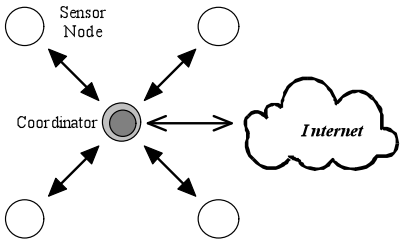


Figure 1: Reference system network architecture

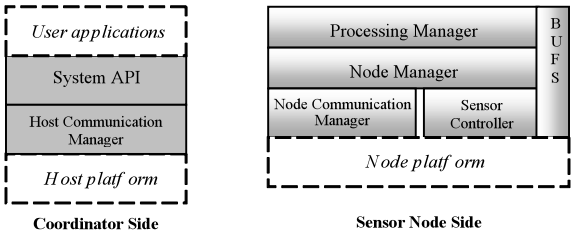


Figure 2: Software architecture layers of the system from the functional perspective

At the coordinator side, an interface to the BSN which is placed between user applications and the hardware and software host platform is made available. User applications manage the WBSN through a system API. The top level of the software architecture at the coordinator allows registered applications being notified of the following events generated by the WBSN: discovery of new nodes, sensor data communication, node alarms, and system messages such as low battery warnings. Commands issued by the user application and network-generated events are both coded in lower-level messages and decoded in higher-level information by the *Host Communication Manager* according to a specific over-the-air protocol. This component handles packets generation and retrieval and is interfaced with specific software components of the host platform to access the physical radio module to transmit/receive packets to/from the WBSN.

At the sensor node side, the software framework provides abstractions of hardware resources such as sensors and the radio, a default set of ready-to-use common signal processing functions and, most importantly, a flexible and modular architecture to be customized and extended to support new physical platforms and sensors and introduce new signal processing services. In particular, the *Node Communication Manager* acts as the counterpart of the *Host Communication Manager*. The *Sensor Controller* manages and abstracts the sensors on the node platform, providing a standard interface to the diverse sensor drivers. It is responsible of sampling the sensors and storing the sensed data in properly defined *Buffers*. The *Node Manager* is the central component, responsible for recognizing the remote requests and dispatching them to the proper components. Finally, the *Processing Manager* consists of a dispatcher for the actual processing services and a standard interface for user-defined services integration. It is worth noting that the SPINE framework [7, 17] is designed according to such an architecture.

4. AGENT-BASED SIGNAL-PROCESSING IN-NODE ENVIRONMENT

In this section we propose an agent-based approach to design and implement the system architecture described in section 3; in particular, the system is specialized for real-time human activity monitoring based on sensor nodes equipped with a 3-axial accelerometer. First, an overview of MAPS (Mobile Agent Platform for Sun SPOTs) architecture and programming model is provided; then, the MAPS-based development of the system is detailed.

4.1 An overview of MAPS

MAPS is a novel Java-based framework for wireless sensor networks based on Sun SPOT technology [1, 2, 14] which enables agent-oriented programming of WSN applications. Other few agent-oriented frameworks [6, 9, 4] for WSNs have been proposed; however, they are all based on TinyOS but one, the AFME framework [15], which is being ported to Sun SPOT. MAPS has been appositely defined for resource-constrained sensor nodes according to the following requirements:

- Component-based lightweight agent server architecture to avoid heavy concurrency models and, therefore, exploit cooperative concurrency to execute agents;
- Lightweight agent architecture to efficiently execute and migrate agents;
- Minimal core services involving agent migration, sensor resources capability access (actuators, input signalers, flash memory, and battery), agent naming, agent communication, and timing.
- Plug-in-based architecture extensions through which any other service should be defined in terms of one or more dynamically installable components implemented as single or cooperating (mobile) agent/s.
- Java as programming language for the agent system and (mobile) agents.

The MAPS architecture, which is shown in Figure 3, consists of the following basic components:

- The mobile agent (MA) is the high-level components of each agent-based application. It is implemented as a single-threaded Isolate according to the Java Sun SPOT libraries.
- The mobile agent execution engine (MAEE) supports the execution of agents by means of an event-based scheduler enabling lightweight concurrency. It handles each event emitted by or to be delivered at an MA through decoupling event queues. The MAEE interacts with the other core components to fulfill service requests (message transmission, sensor reading, timer setting, etc) issued by MAs.
- The mobile agent migration manager (MAMM) supports the migration of agents from one sensor node to another. In particular, the MAMM is based on the feature of Isolate (de)hibernation provided by the Sun SPOT environment [19] and is therefore able to stop and hibernate an MA, serialize it into a byte array and transmit it to the target sensor node. On the migration target sensor node, the MAMM can receive a message containing a serialized MA, deserialize, dehibernate and resume it. The agent serialization format includes data and execution state whereas the code should already reside at

the destination node (this is a current limitation of the Sun SPOTs which do not support dynamic class loading and code migration).

- The mobile agent communication channel (MACC) enables inter-agent communications based on asynchronous messages supported by the Radiogram protocol. Messages can be unicast or broadcast.
- The mobile agent naming (MAN) provides agent naming based on proxies to support the MAMM and MACC components in their operations. The MAN also manages the (dynamic) list of the neighbor sensor nodes which is updated through a beaconing mechanism based on broadcast messages.
- The timer manager (TM) provides the timer service which allows for the management of timers to be used for timing MA operations.
- The resource manager (RM) provides access to the resources of the Sun SPOT node: sensors (3-axial accelerometer, temperature, light), switches, leds, battery, and flash memory.

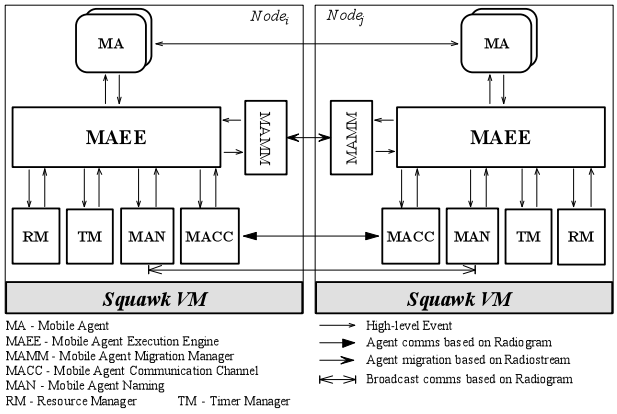


Figure 3: Architecture of MAPS.

The main programming abstractions of MAPS are Agents and Events. While events formalize interaction among components and agents, agents are the active entities whose behavior is modeled as a multi-plane state machine (MPSM). The MPSM consists of a set of planes, global variables and global functions. Each plane may represent the behavior of the MA in a specific role so also enabling role-based programming. In particular a plane is composed of local variables, local functions, and an Event-Condition-Action (ECA) ruled automaton that represents the dynamic behavior of the MA in that plane. The automaton is composed of states and mutually exclusive transitions among states. Transitions are labeled by ECA rules: E[C]/A, where E is the event name, [C] is a boolean expression based on the global and local variables, and A is the atomic action. A transition t is triggered if t originates from the current state (i.e. the state in which the ECAA is), the event with the event name E occurs and [C] holds. If the transition fires, A is executed and the state transition finally takes place. In particular, the atomic action can contain global/local variable and functions to carry out computation, and, particularly, the core primitives (see Figure 4) to request specific services. As agents interact through events, the delivery of an event at agents is asynchronous and carried out by the event dispatcher (a component of the MAEE) which inserts the event in the agent queue. Once the ECAA is idle (i.e. the handling of the last delivered event is completed), a new event is fetched out from the queue and handled. It is worth noting that the MPSM-based agent behavior programming allows exploiting the benefits deriving from three main paradigms for WSN programming: event-driven programming, state-based programming and mobile agent-based programming.

```
public void send(String sourceMA, String targetMA, Event
                 message, boolean local);

sourceMA = message sender agent
targetMA = message receiver agent
message  = event encapsulating message content and
           transmission mode (unicast or broadcast)
local    = true for local messages, false otherwise

public boolean create(String agent, String [] params,
                      String nodeID);

agent  = agent class name
params = agent creation parameters
nodeID = id of node on which the agent should be created

public boolean create(String agentID, String agent,
                      String [] params, String nodeID);

agentID = the ID of the agent to be created, in the create
method above the agentID is automatically created by the
system and returned to the creator by an event

public String setTimer(boolean periodic, long timeout,
                       Event backEvent);

periodic = true if the timer is periodic, false if one-shot
timeout  = expiration time in msec
backEvent= the event notifying the timer expiration
The String return value represents the timer ID

public void resetTimer(String timerID);

timerID = the ID of the timer to be reset

public void sense(Event backEvent);
public void actuate(Event backEvent);
public void flash(Event backEvent);
public void input(Event backEvent);

backEvent = event which includes the setting of the sense,
actuate, flash and input operations when they are invoked.
After their invocation, the operation results are inserted
in the backEvent which is then asynchronously delivered to
the invoking agent
```

Figure 4: The MAPS core primitives.

4.2 Design and implementation

The developed prototype aims at monitoring human activities in real-time by recognizing their postures (e.g. lying down, sitting and standing still) and movements (e.g. walking). The architecture of the system, shown in Figure 5, is organized into a coordinator and two sensor nodes according to the reference WBSN system described in section 3.

The coordinator side (see Figure 5) is based on the Java-based SPINE coordinator [7], developed in the context of the SPINE project [17]. In particular, the SPINE Manager is used by end-user applications (e.g. real-time activity monitoring application) for sending commands to the sensor nodes. Moreover, the SPINE Manager is responsible of capturing low-level messages and node events through the SPINE Listener, which integrate several sensor platform-specific SPINE communication modules (e.g. TinyOS, Z-Stack, etc), to notify registered applications with higher-level events and message content. A SPINE communication module is composed of a send/receive interface and some components that implement such interface according to the specific sensor platform and that formalize the high-level SPINE messages in sensor platform-

specific messages. In this work, the SPINE Listener has been enhanced with a new MAPS/Sun SPOT communication module to configure and communicate with MAPS-based sensor nodes. Such module translates high-level SPINE messages formatted according to the SPINE OTA (Over-The-Air) protocol [17] into lower-level MAPS/Sun SPOT messages through its transmitter component and vice versa through its receiver component. The latter also integrates an application-specific logic for the synchronization of the two sensors (see below and §5.2). The SPINE-based real-time activity monitoring application was thus completely reused as well as the SPINE Manager, only the SPINE Listener was modified to account for such enhancement.

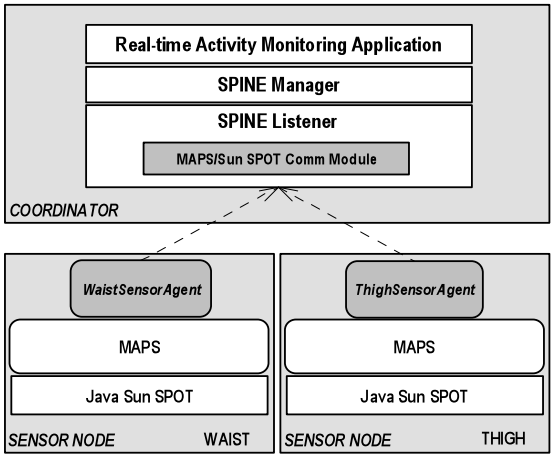
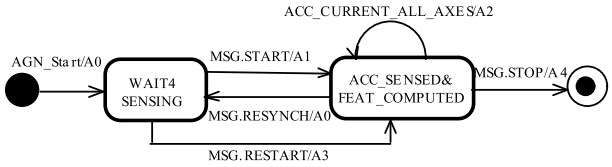


Figure 5: Architecture of the real-time activity monitoring system.

The sensor node side (see Figure 5) is based on two Java Sun SPOTs sensors respectively positioned on the waist and the thigh of the monitored person. In particular, MAPS is resident on the sensor nodes and supports the execution of the WaistSensorAgent and the ThighSensorAgent. Such sensor agents have the following similar step-wise cyclic behavior:

1. *Sensing* the 3-axial accelerometer sensor according to a given sampling time (ST);
2. *Computation* of specific features (Mean, Max and Min functions) on the acquired raw data according to the window (W) and shift (S) parameters. In particular, W is the sample size on which features are computed whereas S is the percentage of sliding on W (usually S is set to 50%);
3. Features *aggregation* and *transmission* to the coordinator;
4. Go to 1.

The agents differ in the specific computed features even though the W and S parameters are equally set. In particular, while the WaistSensorAgent computes the mean values for data sensed on the XYZ axes, the min and max values for data sensed on the X axis, the ThighSensorAgent calculates the min value for data sensed on the X axis. The behavior of the WaistSensorAgent is specified through the 1-plane reported in Figure 6 (the behavior of the ThighSensorAgent has the same structure but the computed features are different as discussed above). In particular, after an initialization action (A0) driven by the occurrence of the AGN_START event, the sensing plane goes into the WAIT4SENSING state. The MSG.START event allow starting the sensing process by the execution of action A1; in particular: (i) sensing parameters (W, S, ST), data acquisition buffers for XYZ channels of the accelerometer sensor (windowX, windowY, windowZ), and data buffers for feature calculation (windowFE4X, windowFE4Y, windowFE4Z) are initialized (see *initSensingParamsAndBuffers* function); (ii) the timer is set for timing the data acquisition according to the ST parameter (see *timerSetForSensing* function; in particular the highly precise Sun SPOT timer is used); (iii) a data acquisition is requested by submitting the ACC_CURRENT_ALL_AXES event by the sense primitive (see *doSensing* function). Once the data sample is acquired, the ACC_CURRENT_ALL_AXES event is sent back with the acquired data and the action A2 is executed; in particular: (i) the buffers are circularly filled with the proper values (see *bufferFilling* function); (ii) the sampleCounter is incremented and the nextSampleIndex is incremented module W for the next data acquisition; (iii) if S samples have been acquired, features are to be calculated, thus sampleCounter is reset, samples in the buffers are copied into the buffers for computing features, calculation of the features is carried out through the *meanMaxMin* function, and the aggregated results are sent to the base station by means of the MSG_TO_BASESTATION event appropriately constructed; (iv) the timer is reset; (v) data acquisition is finally requested. In the ACC_SENSED&FEAT_COMPUTED state the MSG.RESYNCH might be received for resynchronization purposes (see below); it brings the sensing plane into the WAIT4SENSING state. The MSG.RESTART brings the sensing plane back into the ACC_SENSED&FEAT_COMPUTED state for (reconfiguring and) continuing the sensing process. The MSG.STOP eventually terminates the sensing process.




```
Variables
byte bodyPos;
String bsAddr;
int W, S, ST;
byte timestamp, sampleCounter;
int nextSampleIndex;
IAT91_TC timer;
double [] windowX4FE, windowY4FE, windowZ4FE;
double [] windowX, windowY, windowZ;
double [] resultsX, resultsY, resultsZ;
Actions
A0: initVars();
A1: initSensingParamsAndBuffers(event);
    timerSetForSensing();
    doSensing();
A2: bufferFilling(event);
    sampleCounter++;
    nextSampleIndex=(nextSampleIndex+1)%W;
    if (sampleCounter==S){
     sampleCounter==0;
     copySensingBuffersIntoBuffersForComputingFeatures();
     computeFeatures();
     transmitFeaturesComputed();
    }
    timerReset();
    doSensing();
A3: timerDisabling();
    initVars();
    A1;
A4: timerDisabling();
Functions
initVars():
 sampleCounter=0; nextSampleIndex=0; agent.timestamp=0;
```

```
initSensingParamsAndBuffers(Event event):
 bodyPos=Byte.parseByte(event.getParam("BODY_POSITION"));
 bsAddr=event.getParam("BASESTATION_ADDRESS");
 W=Integer.parseInt(event.getParam("WINDOW_SIZE"));
 S=Integer.parseInt(event.getParam("SHIFT_SIZE"));
 ST=Integer.parseInt(event.getParam("SAMPLE_RATE_MS"));
 windowX = new double[W]; windowY = new double[W];
 windowZ = new double[W]; windowX4FE = new double[W];
 windowY4FE = new double[W]; windowZ4FE = new double[W];
timerSetForSensing():
 timer = Spot.getInstance().getAT91_TC(0);
 int cnt = (int)(ST * 1000 / 2.1368);
 timer.configure(TimerCounterBits.TC_CAPT |
                 TimerCounterBits.TC_CPCTRG |
                 TimerCounterBits.TC_CLKS_MCK128);
 timer.setRegC(cnt); timer.enableAndReset();
 timerReset();
doSensing():
 Event accel = new Event(agent.getId(), agent.getId(),
               Event.ACC_CURRENT_ALL_AXES, Event.NOW);
 agent.sense(accel);
bufferFilling(Event event):
 windowX[nextSampleIndex] = Double.parseDouble(
          event.getParam(ParamsLabel.ACC_ACCEL_X_VALUE));
 windowY[nextSampleIndex] = Double.parseDouble(
          event.getParam(ParamsLabel.ACC_ACCEL_Y_VALUE));
 windowZ[nextSampleIndex] = Double.parseDouble(
          event.getParam(ParamsLabel.ACC_ACCEL_Z_VALUE));
timerReset():
 timer.enableIrq(TimerCounterBits.TC_CPCS);
 timer.waitForIrq();
 timer.status();
timerDisabling():
 timer.disable();
 timer.shutDown();
computeFeatures():
 resultsX = meanMaxMin(windowX4FE);
 resultsY = meanMaxMin(windowY4FE);
 resultsZ = meanMaxMin(windowZ4FE);
trasmitFeaturesComputed():
 Event msgToServer = new Event(this.agent.getId(),
  Constants.MSG_TO_BASESTATION,Event.MSG_TO_BASESTATION,
  Event.NOW);
 msgToServer.setParam(ParamsLabel.AGT_BS_ADDR, bsAddr);
 msgToServer.setParam("BodyPosition", "Waist");
 msgToServer.setParam("MeanX","" + resultsX[0]);
 msgToServer.setParam("MeanY","" + resultsY[0]);
 msgToServer.setParam("MeanZ","" + resultsZ[0]);
 msgToServer.setParam("MaxY", "" + resultsY[1]);
 msgToServer.setParam("MinY", "" + resultsX[2]);
 timestamp =(timestamp+1)%128;
 msgToServer.setParam("Timestamp", ""+timestamp);
 agent.send(agent.getId(), Constants.MSG_TO_BASESTATION,
            msgToServer, false);
double [] meanMaxMin(double []): //omissis
```

Figure 6: 1-plane behavior of the WaistSensorAgent

5. PROTOTYPE EVALUATION

The evaluation of the developed prototype involves the following two aspects (which are respectively detailed in the next two subsections):

- The timing and synchronization of the real-time monitoring which respectively refer to (i) how fine grain the sensing activity at the sensor node can be and (ii) how to keep the activities of the two sensor agents synchronized.
- The recognition accuracy which shows how well the human postures/movements are recognized by the system.

5.1 Timing and synchronization

Two important issues to deal with are the timing of the sensing process in terms of admissible sampling rate and the synchronization between the operations of the two agents which is to be maintained within a maximum skew for not affecting the real-time monitoring. If such skew is overtaken, the two agents are to be re-synchronized. Indeed such two aspects are strictly correlated. In particular, as the sensor agents compute a different number of features, when the sampling rate is high, the agent computing more features (i.e. the WaistSensorAgent) takes more time to complete its operations for each S sample acquisition than the ThighSensorAgent. Re-synchronization is driven by the synchronization logic included in the developed MAPS/SunSpot comm module (see §4.2), which sends a resynchronization message (see Figure 6) as soon as it detects that the synchronization skew is greater than a given threshold. Detection is based on the skew time between the receptions of two messages sent by the agents that contain features referring to the same interval of S sample acquisition: if skew >= P * S * ST, where P is a percentage, S=0.5W, and ST is the sampling time. Thus, the evaluation aimed at analyzing the synchronization of the sensor agents and their monitoring continuity. The defined measurements are:

- The *Packet Pair Average Time* (PPAT), which is the average reception time between two consecutive pairs of synchronized packets (same logical timestamp, see timestamp variable in Figure 6) containing the computed features (see MSG_TO_BASESTATION event in Figure 6) sent by the sensor agents. PPAT should be ideally equals to ST*S, i.e. the packet pair arrives each monitoring period and so there is no de-synchronization in the average.
- The *Synchronization Packet Percentage* (SPP), which is the percentage of resynchronization packets (see RESYNCH event in Figure 6), which are sent by the coordinator for re-synchronizing the sensor agents, calculated with respect to the total number of received feature packets. SPP should be as much as possible close to 0, i.e. a few or no resynchronizations are carried out and so the monitoring can be continuous as a resynch operation usually takes 600 ms.

In particular, the experiments were carried out by fixing ST (ms) = [25, 50, 100], W (samples) = [100, 80, 40, 20, 10], and P (%) = [5, 10, 25]. Figure 7 shows the obtained results for P=25% and P=5%. As can be noticed, the system cannot support an ST=25ms because PPAT is always greater than the ideal value and SPP is too high. This lead to a non continuous monitoring due to the very frequent resynchronization (SPP>=15%). An ST=50ms can be supported for P=25% and W>=40 as SPP is maximum 8% so slightly impacting the monitoring continuity. The best results are obtained with ST=100ms, P=25% and W>=20; they guarantee monitoring continuity due to an SPP≈0% and regularity as experimented PPAT ≈ ideal PPAT for W>=20. If P=5% and W=[10, 20] or P=25% and W=10 an ST=100 ms is not a good value too because an out-of-limits skew frequently occurs. It is worth noting that even though a lower ST would allow a more accurate monitoring, the considered human activities can be well captured by an ST=100ms as demonstrated by the experimental results obtained from the carried-out real-time human activity monitoring (see §5.2).

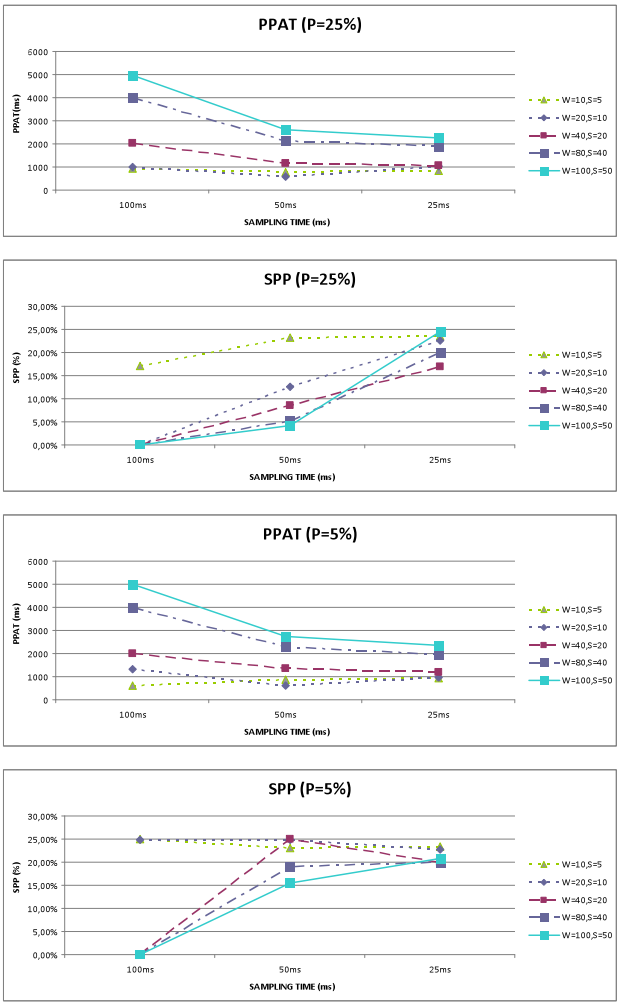


Figure 7: Analysis of the synchronization of the sensor agents: PPAT and SPP for P=25% and P=5% by varying W.

5.2 Recognition accuracy

The activity monitoring system integrates a classifier based on the K-Nearest Neighbor algorithm [5] that is capable of recognizing postures and movements defined in a training phase. The classifier was setup through a training phase and tested considering the following parameter setting: ST=100ms, W=20 (S=10), P=25%. Accordingly, the features (Min, Max and Mean) are computed on 20 sampled data every new 10 samples acquired by the sensors. The training phase used a KNN-based classifier parameterized with K=1 and the Manhattan distance which performs quite well as classes (lying down, sitting, standing still and walking) are rather separate and scarcely affected by noise. The test phase is carried out by considering the pre-defined sequence of postures/movements represented by the state machine reported in Figure 8.

Accordingly, the obtained classification accuracy results are reported in Figure 9. As can be noted after a transitory period of 5 sec from one state to another all the postures/movements are recognized with an accuracy of 100%. The state transitions more difficult to recognize are STA→SIT, WLK→STA, and SIT→LYG, whereas the transition STA→WLK is recognized as soon as it occurs. The obtained results are good and encouraging if compared with other works in the literature which use more than two sensors on the human body [13].

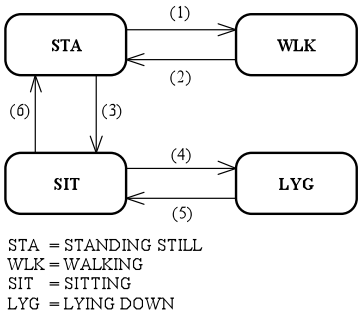


Figure 8: State machine of the pre-defined sequence of postures/movements.

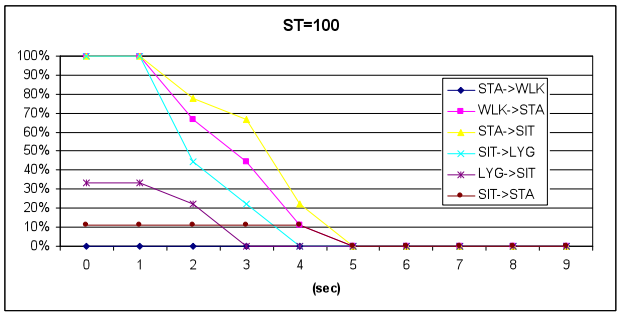


Figure 9: Percentage of mismatches vs. transitory time.

6. CONCLUSIONS

Programming WBSN applications is a complex task which requires suitable programming paradigms and frameworks coping with the WBSN specific characteristics. Several kinds of frameworks and approaches have been to date proposed. Among them, domain specific frameworks have the potential to provide both rapid development of WBSN applications and also good performances. In this paper we have proposed an agent-oriented approach, which relies on the basic features characterizing the domain specific frameworks and on the agent-oriented MAPS framework, aiming to offer more programming effectiveness while providing the required efficiency. In fact, MAPS has been purposely defined for resource-constrained environments and is based on (i) lightweight agents so avoiding conventional heavyweight agent architectures; (ii) run-time architecture formed of components efficiently handling the low-level sensor node functions and providing higher-level services to agents. In particular, by using MAPS, a WBSN application can be structured as a set of agents distributed on sensor nodes supported by a component-based agent execution engine which provides basic services such as message transmission, agent creation, timer handling, easy access to the sensor node resources, and agent migration if needed.

A complete case study concerning the development and testing of a real-time human activity monitoring system based on wireless body sensor networks has been described. It is emblematic of the effectiveness and suitability of MAPS to deal with the programming of WBSN applications. The carried out

performance evaluation of the developed prototype shows fine synchronization of the sensor nodes, continuous real-time monitoring, and good recognition accuracy, once parameters are carefully set. However, the MAPS-based development of new applications having stringent requirements (sensing rate, computing speed, message transmission latency) must be carefully analyzed case by case.

Future research efforts are devoted to: (i) introducing the fall detection, which allows arising an alarm when detecting the monitored person has fallen, in the developed agent-based system; (ii) porting MAPS onto the Sentilla JCreate pervasive computers which are compliant to Java ME CLDC 1.1; (iii) developing a full-fledged agent-based version of SPINE (named ASpine) through MAPS and the JADE framework to enable agent-oriented development of pervasive applications for ambient assisted living (such as emergency medical care) based on heterogeneous computing platforms: PCs/workstations (JADE), PDAs/smartphones (JADE Leap), and sensor nodes (MAPS).

7. ACKNOWLEDGMENTS

Authors wish to thank Roberta Giannantonio and Marco Sgroi for their precious contributions in terms of ideas and discussions, and Alessio Carbone for his on-going implementation efforts on ASpine. This work has been partially supported by CONET, the Cooperating Objects Network of Excellence, funded by the European Commission under FP7 with contract number FP7-2007-2-224053.

REFERENCES

[1] F. Aiello, G. Fortino, A. Guerrieri, “Using mobile agents as an effective technology for wireless sensor networks,” In Proc. of the 2nd IEEE/IARIA Int’l Conference on Sensor Technologies and Applications (SENSORCOMM 2008), Aug 25-31, Cap Esterel, France, 2008.

[2] F. Aiello, G. Fortino, R. Gravina, A. Guerrieri, “MAPS: a Mobile Agent Platform for Java Sun SPOTs,” In Proceedings of the 3rd International Workshop on Agent Technology for Sensor Networks (ATSN-09), jointly held with the 8th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-09), 12th May, Budapest, Hungary, 2009.

[3] Ling Bao, Stephen S. Intille, “Activity Recognition from User-Annotated Acceleration Data”, In Proc. of the 2nd Int. Conference on Pervasive Computing (PERVASIVE), pp. 1–17, 2004.

[4] M. Chen, S. Gonzalez, V. C. M. Leung, “Applications and Design Issues for Mobile Agents in Wireless Sensor Networks,” IEEE Wireless Communications (see also IEEE Personal Communications) 14 (6) (2007) 20–26.

[5] T. Cover, P. Hart, “Nearest neighbor pattern classification”, In IEEE Trans. Inform. Theory Vol. 13, pp. 21-27, January 1967.

[6] C-L Fok, G-C Roman, C Lu, “Rapid Development and Flexible Deployment of Adaptive Wireless Sensor Network Applications,” In Proc. of the 24th Int’l Conference on Distributed Computing Systems (ICDCS'05), Columbus, Ohio, June 6-10, 2005, pp. 653-662.

[7] R. Gravina, A. Guerrieri, G. Fortino, F. Bellifemine, R. Giannantonio, M. Sgroi, “Development of Body Sensor Network Applications using SPINE,” In Proc. of. IEEE Int’l Conference on Systems, Man, and Cybernetics (SMC 2008), Singapore, Oct. 12-15, 2008.

[8] S. Iyengar, F. Tempia Bonda, R. Gravina, A. Guerrieri, G. Fortino, A. Sangiovanni-Vincentelli, “A Framework for Creating Healthcare Monitoring Applications Using Wireless Body Sensor Networks”, In the Proc. of the 3rd International Conference on Body Area Networks (BodyNets’08), Tempe (AZ), USA, Mar. 13-15, 2008.

[9] Y Kwon, S. Sundresh, K. Mechitov, G. Agha, "ActorNet: An Actor Platform for Wireless Sensor Networks," In Proc. of the 5th Int’l Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS), pages 1297-1300, 2006.

[10] C. Lombriser, D. Roggen, M. Stager, G. Troster, “Titan: A Tiny Task Network for Dynamically Reconfigurable Heterogeneous Sensor Networks”, In Verteilten Systemen (KiVS 2007), Bern, Switzerland, Feb 26–Mar 2, 2007.

[11] M. Luck, P. McBurney and C. Preist, “A manifesto for agent technology: towards next generation computing,” Autonomous Agents and Multi-Agent Systems 9(3), pp. 203–252, 2004.

[12] David Malan, Thaddeus Fulford-Jones, Matt Welsh, Steve Moulton, “CodeBlue: An Ad Hoc Sensor Network Infrastructure for Emergency Medical Care”, In MobiSys 2004 Workshop on Applications of Mobile Embedded Systems (WAMES 2004), June, 2004.

[13] U. Maurer, A. Smailagic, D.P. Siewiorek, M. Deisher, “Activity recognition and monitoring using multiple sensors on different body positions”, In Proc. of the 3rd Int. Workshop on Wearable and Implantable Body Sensor Networks (BSN 2006), MIT, Boston (MA), USA.

[14] Mobile Agent Platform for Sun SPOT (MAPS), documentation and software release 1.1 at http://maps.deis.unical.it (2010).

[15] C. Muldoon, G.M.P. O'Hare, M. J. O'Grady, R. Tynan, “Agent Migration and Communication in WSNs,” in 1st International Workshop on Sensor Networks and Ambient Intelligence, held in conjunction with the 9th International Conference on Parallel and Distributed Computing, Applications and Technologies, pp. 425-430, IEEE Computer Society Press, Dec. 2008.

[16] B. Najafi, K. Aminian, A. Ionescu, F. Loew, C. J. Büla, P. Robert, “Ambulatory System for Human Motion Analysis Using a Kinematic Sensor: Monitoring of Daily Physical Activity in the Elderly”, In IEEE Transactions on Biomedical Engineering, Vol. 50, Issue 6, pp. 711–723, June 2003.

[17] Signal Processing In-Node Environment (SPINE), documentation and software at http://spine.tilab.com (2009).

[18] K. Sohraby, D. Minoli, T. Znati, “Wireless Sensor Networks: technology, protocols, and applications”, Wiley, 2007.

[19] Sun™ Small Programmable Object Technology (Sun SPOT), documentation and software at http://www.sunspotworld.com/ (2009).

[20] TinyOS, documentation and software, http://www.tinyos.net (2009).

[21] G-Z. Yang, “Body Sensor Networks”, Springer, 2006.

[22] Z-Stack (ZigBee Protocol Stack), Texas Instruments, documentation and software, http://focus.ti.com/docs/toolsw/folders/print/z-stack.html (2009).

Trust and Reputation Through Partial Identities

Jose M. Such
Departament de Sistemes Informàtics i Computació
Universitat Politècnica de València
Camí de Vera s/n, València, Spain
jsuch@dsic.upv.es

Agustin Espinosa
Departament de Sistemes Informàtics i Computació
Universitat Politècnica de València
Camí de Vera s/n, València, Spain
aespinos@dsic.upv.es

Vicent Botti
Departament de Sistemes Informàtics i Computació
Universitat Politècnica de València
Camí de Vera s/n, València, Spain
vbotti@dsic.upv.es

Ana Garcia-Fornes
Departament de Sistemes Informàtics i Computació
Universitat Politècnica de València
Camí de Vera s/n, València, Spain
agarcia@dsic.upv.es

ABSTRACT

This paper explores the relationships between the hard security concepts of identity and privacy on the one hand, and the soft security concepts of trust and reputation on the other hand. We specifically focus on two vulnerabilities that current trust and reputation systems have: the change of identity and multiple identities problems. As a result, we provide a privacy-preserving solution to these vulnerabilities which integrates the explored relationships among identity, privacy, trust and reputation.

Categories and Subject Descriptors

I.2.11 [**Artificial Intelligence**]: Distributed Artificial Intelligence—*Multiagent systems*

General Terms

Security, Management

Keywords

Trust, Reputation, Privacy, Security

1. INTRODUCTION

Security related studies in the Multiagent Systems (MAS) research field have been increasing over the last few years, as have intelligent autonomous agents and MAS based applications. This is mainly due to the fact that an understanding of the actual risk when using these sorts of applications is needed, since an agent's incorrect or inappropriate behavior may cause non-desired effects such as money and data loss.

Rasmusson and Jansson [11] first introduced the difference between two approaches to security in information systems, i.e., what they called hard and soft security. On the one hand, the term hard security is used for traditional security mechanisms like authentication, authorization, integrity, confidentiality, etc. On the other hand, the term soft security is used for social control mechanisms in general.

A major difference between these two approaches is related to how they deal with intruders in a system. Hard security mechanisms - such as identity management - aim to prevent intruders from joining the system so that the system is supposedly intruder free. Soft security mechanisms - such as trust and reputation - expect, and even accept, the presence of intruders in the system, so they attempt to identify the intruders and prevent them from harming the other actors in the system.

We strongly encourage research transversal to these two approaches to security. This is due to the fact that when relationships between these two approaches are not taken into account, some vulnerabilities can emerge which otherwise would not. The agent community in particular has not been taking the relationships between these two approaches into account.

Current Trust and Reputation systems are based on the assumption that identities are long-lived, so that ratings about a particular entity from the past are related to the same entity in the future. However, when such systems are actually used in real domains this assumption is no longer valid. For instance, an entity which has a low reputation due to its cheating behavior may be really interested in changing her identity and restarting her reputation from scratch. This is what Jøsang et al. [7] called the *change of identities* problem.

This problem has also been identified by other researchers under different names. The work of Kerr and Cohen [8] shows that Trust and Reputation Systems exhibit multiple vulnerabilities that can be exploited by attacks performed by cheating agents. Among these vulnerabilities, the *re-entry* vulnerability exactly matches the *change of identities* problem exposed by Jøsang et al. They propose a simple attack that takes advantage of this vulnerability: An agent opens an account (identity) in a marketplace, uses her account to cheat for a period, then abandons it to open another.

Kerr and Cohen [8] also point out the fact that entities could create new accounts (identity in the system) at will,

not only after abandoning their previous identity but also holding multiple identities at once. This can lead to other attacks such as what they called the *proliferation* attack. An example of this attack could be an agent that holds multiple identities in a marketplace and attempts to sell the same product through each of them.

It is worth mentioning that this is not an authenticity problem. Interactions among entities are assured, i.e, an agent holding an identity is sure of being able to interact with the agent that holds the other identity. However, there is nothing which could have prevented the agent behind that identity from holding another identity previously or holding multiple identities at once. For instance, let us take a buyer agent and a seller agent in an e-marketplace. The buyer has an identity in the e-marketplace under the name of *buy1* and the seller two identities in the e-marketplace *seller1* and *seller2*. Authentication in this case means that if *buy1* is interacting with *seller1* she is sure that she is interacting with who she wants. However, *buy1* has no idea that *seller1* and *seller2* are the same entity.

These vulnerabilities can be more or less harmful depending on the final domain of the application using trust and reputation models. For instance, in a social network like Last.fm[1] these vulnerabilities are not so important. Last.fm users can recommend music to each other. Therefore, a user who always fails to recommend good music to other users may gain a very bad reputation. If this user creates a new account in Last.fm (a new identity in Last.fm) her reputation starts from scratch, and she is able to keep on recommending bad music. The point is that other users are not seriously damaged by getting bad music recommendations. However, in systems where users can be seriously damaged (e.g. in an e-marketplace by losing money) these vulnerabilities need, at least, to be considered.

As far as we are concerned, the two vulnerabilities presented are due to the lack of a clear definition of identity and its relationship to trust and reputation. In this sense, we introduce in the next section the concept of partial identity and relate this concept to trust and reputation later on in sections 3 and 4. In section 5 we introduce what we call the *Partial Identity Unlinkability* Problem (PIUP) which includes these two vulnerabilities. As a result, a privacy preserving solution to PIUP is proposed in section 6, taking into consideration partial identities and their relation to trust and reputation.

2. IDENTITY AND PARTIAL IDENTITIES

The identity and partial identity terms are broadly used in identity management literature such as [2], [9] and [10]. However, there is a lack of clear and formal definitions of these two terms. In this section, we propose formal definitions of both identity and partial identity.

We assume that an entity can be: a legal person (a human being, a company, etc.) and a software entity (an intelligent agent, a virtual organization, etc.).

We also assume that entities are described by attributes attached to them. Attributes are defined as pairs $a = \langle name, value \rangle$. We denote the set of all attributes describing an entity e as A_e. Attributes can describe a great range of topics. For instance, entity names, biological characteristics (only for human beings), location (permanent address, geo-location at a given time), competences (diploma, skills), social characteristics (affiliation to groups, friends), and even behaviors (personality or mood).

Definition 1. A *partial identity* PI_e is any subset of attributes of an entity which sufficiently identifies this entity within a given set of entities.

$$e \in E \ \wedge \ PI_e \subseteq A_e \ \wedge \ identify(PI_e, E) = \{e\}$$

Where e is the entity which holds the partial identity, E is the set of entities taken into consideration, A_e is the set of attributes that describes e and $identify$ is defined as follows $identify$: $\mathcal{P}(\mathbb{A}) \times \mathcal{P}(\mathbb{E}) \rightarrow \mathcal{P}(\mathbb{E})$ such that $identify(A, E) = \{e \mid e \in E \ \wedge \ \forall \ a \in A, \ a \in A_e\}$ where $\mathbb{A}$ is the set of all of the attributes describing entities and $\mathbb{E}$ is the set of all of the existing entities.

Definition 2. The *identity* I_e of an entity e is the union of all of the attributes from all of the partial identities of e.

$$I_e = \bigcup_{j=1}^{N} PI_e^j$$

Where PI_e^j is a partial identity of the entity e, and N is the number of all of the partial identities that e holds.

An identity of an entity is composed of many partial identities. Moreover, an identity potentially identifies an entity within any set of entities, while a partial identity *may* not. For instance, let a human being be registered with a given profile in the Last.fm social network. This profile is a partial identity because it does sufficiently identify the human being among all of the different entities registered in Last.fm. However, this profile may not sufficiently identify this human being among all of the possible entities.

Although each partial identity usually represents the entity in a specific context or role, the same partial indentity can represent the entity in *different contexts*. For instance, a driver license represents an entity in the context of operating a motorized vehicle but it also represents an entity in the context of accessing a disco only for adults.

In order for the reader to better understand the identity and partial identity concepts, Figure 1 shows the identity and some of the partial identities of an individual person called Bob. Four partial identities are shown regarding four contexts: government, work, health care and social networking (Last.fm). For the sake of clarity, we only show some attributes that make up each of the partial identities represented. It is easily observed that the name and address of Bob are shared by three partial identities but are not used in the partial identity he uses in Last.fm.

Following the previous definitions, we consider a real identity as a partial identity of an entity with respect to the set of all of the legal persons. Formally:

Definition 3. A *real identity* RI_e is any subset of attributes of an entity which sufficiently identifies this entity within the set of all of the legal persons.

$$e \in E \ \wedge \ RI_e \subseteq A_e \ \wedge \ identify(RI_e, \mathbb{L}) = \{e\}$$

Where e is the entity which holds the identity, $\mathbb{L}$ is the set of all of the legal persons, A_e is the set of attributes that describes e and $identify$ is the same function previously defined. As described later on in section 6, we use real identities for accountability concerns such as law enforcement.

[1]Last.fm `http://www.last.fm`

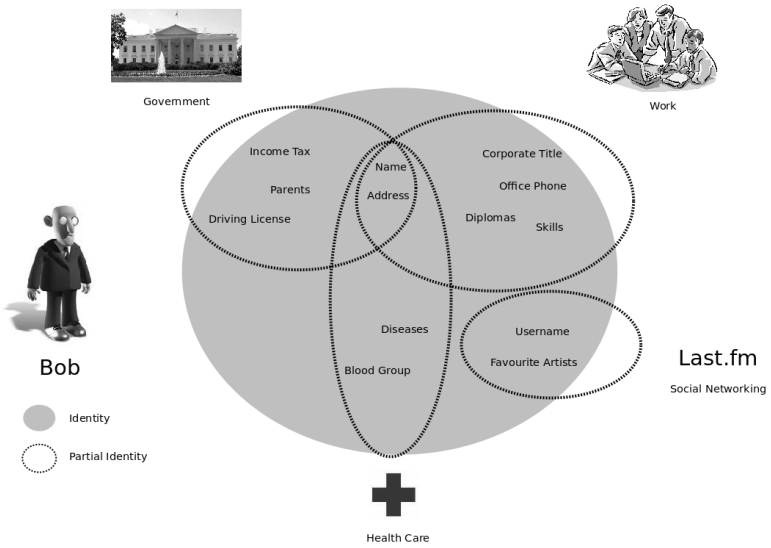


Figure 1: Identity and Partial Identities of Bob

For this reason, real identities are restricted to legal persons (human beings, companies, ect.) that can be considered in front of the law. A real identity would be for example: *Bob Andrew Miller, born in Los Angeles, CA, USA on July 7, 1975.*

A broad misunderstanding regarding identities is related to the concept of identifier and how it relates to an identity or a partial identity. The identity term is usually used instead of that of identifier.

Definition 4. An *identifier* id_e is an attribute which sufficiently identifies an entity within a set of entities.

$$e \in E \ \wedge \ id_e \in A_e \ \wedge \ identify(\{id_e\}, E) = \{e\}$$

Where e is the entity which holds the identity, E is a set of entities, A_e is the set of attributes that describes e and $identify$ is the same function previously defined.

Examples of identifiers include names, driving license numbers, usernames, nicknames, e-mail addresses, agent identifiers (AID), Universal Resource Identifier (URI), etc.

An identifier can also be defined as a partial identity PI_e so that $|PI_e| = 1$. However, partial identities are usually composed of not only identifiers but also more attributes describing the entity thoroughly in a given context. For instance, a partial identity in Last.fm (a Last.fm account) includes a username (identifier), an e-mail address and favorite artists.

3. TRUSTING ENTITIES THROUGH PARTIAL IDENTITIES

In this section, we propose the building of trust relationships through partial identities. In this sense, we first introduce the concept of trust.

According to Gambetta [5], trust is "*the subjective probability by which an individual, A, expects that another individual, B, performs a given action on which its welfare depends*".

Most of the trust models proposed by the agent community are based on Gambetta's definition and treat trust as a probability. Different grounding theories are used to build these models. Although most of them are based on Game Theory (for a survey refer to [15]) there are other probabilistic approaches like [17], in which Sierra and Debenham use Information Theory.

Agent community has also developed *cognitive* models which treat trust as more than a probability. For instance, Castelfranchi and Falcone [1] define trust as "*a mental state, a complex attitude of an agent x towards another agent y about the behaviour/action relevant for the result (goal) g*".

Both probabilistic and cognitive models share that trust is established from a *trustor* (the one who trusts) to a *trustee* (the one who is trusted). Thus, we focus on trust as a directed relationship between two entities. In this sense, a primary requirement is that the trustor is able to recognize the trustee when interacting with each other.

In the real world, an individual can recognize other individuals by means of identity documents such as a passport. However, inter-personal meetings are also carried out without the needing for such documents. For instance, let Alice be an individual who always goes to the same supermarket. Alice always chooses the same checkout to pay for the items she buys. The checkout she chooses is the one where the fastest cashier (according to Alice's criterion) in the supermarket is (the cashier is at a different checkout each time). Alice recognizes the face of the supermarket cashier that she trusts as being the fastest in the supermarket. In this example, no identity document is needed because the trustor recognizes the face of the trustee from past interactions.

In the digital world there is no physical contact, all of the interactions between entities are carried out through online networks and most of them across the Internet. The increase in global connectivity is increasing the number of entities taking part in the digital world and also the number of interactions they carry out. In this scenario, recognizing an entity in an interaction usually means authenticating it using technologies like Kerberos[2], OpenID[3], and so on. En-

[2]Kerberos `http://web.mit.edu/Kerberos/`
[3]OpenID `http://openid.net/`

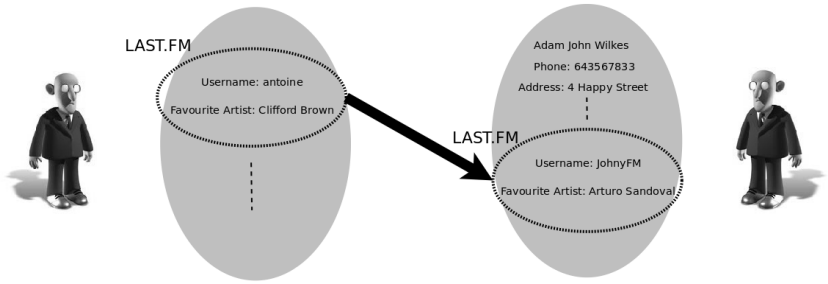


Figure 2: Trust Through Partial Identities

tities are authenticated using such technologies according to a partial identity that they hold.

We consider trust relationships to be established between two entities through some of their partial identities. Moreover, these partial identities represent part of the context where the trust relationship is established. We assume that the context $\mathcal{C}$ of a trust relationship can be described by means of pairs $a = \langle name, value \rangle$ of attributes. Thus, the partial identity of the trustor ($PI_{trustor}$) and the partial identity of the trustee ($PI_{trustee}$) are part of such attributes that describe the context $\mathcal{C}$. Formally:

$$PI_{trustor} \cup PI_{trustee} \subseteq \mathcal{C}$$

Partial identities are key parts when building trust relationships. There are attributes of a partial identity of an entity that clearly describe important features of an entity. For instance, a corporate title (such as chief executive officer) is an attribute which is part of the partial identity of an employee of a company. When this employee is interacting with other entities in a business context, his corporate title is an important attribute that the rest of the entities in that context will consider valuable when trusting him.

Figure 2 shows an example of a trust relationship established between two entities through partial identities. There are two entities: the real identity of the entity on the left is not known while the entity on the right is Adam John Wilkes. Both entities have a partial identity when they interact in the Last.fm social network. In this sense, the entity with the username antoine is trusting (represented as a directed arrow) the entity with the username JohnyFM (Adam John Wilkes). This trust relationship is contextualized in Last.fm. Moreover the favorite artist of both partial identities plays a crucial role in the trust relationship. In this sense, both antoine and JohnyFM have trumpet players (Clifford Brown and Arturo Sandoval) as favorite artists. Therefore, music recommendations from JohnyFM to antoine may be relevant to antoine.

4. REPUTATION THROUGH PARTIAL IDENTITIES

In the previous section, we stated how trust relationships can be built through partial identities. In this section, we state how partial identities relate to reputation.

We understand reputation in the same way as Sabater et al. in their Repage Model [16]. In this sense, reputation is a social evaluation of a target entity attitude towards socially desirable behavior which is circulating in the society (and can be agreed on or not by each one of the entities in the society).

Reputation, just like trust, is known to be context dependent [15]. For instance, a lawyer can have a great reputation defending digital criminals while having a bad reputation making cakes.

Unlike trust, reputation also relates to anonymity. The anonymity concept is defined by Pfitzmann and Hansen in [9] as: *"Anonymity of a subject means that the subject is not identifiable within a set of subjects"*. Reputation, as a social evaluation circulating in the society, is *anonymously* assigned to an entity. Therefore, the social evaluation any entity has about other entities remains private (whenever she does not communicate her social evaluation to others in a non-anonymous fashion).

The anonymous nature of reputation is sometimes not taken into account, which leads to some problems. For instance, the eBay reputation system is not anonymous which leads to an average 99% of positive ratings[13]. This is due to the fact that entities in eBay do not negatively rate other entities for fear of retaliations which could damage their own reputation and welfare.

We consider reputation as an anonymous social evaluation of an entity in a given context through one of its partial identities. In this sense, the partial identity of the entity reputed is needed when defining the context of a reputation.

Assuming that the context C of a reputation is described by $a = \langle name, value \rangle$ pairs of attributes, e is the entity evaluated and PI_e is her partial identity:

$$PI_e \subseteq \mathcal{C}$$

Therefore, if an entity has a reputation in a given context, all of the entities interacting with this entity in the same context can be aware of her reputation through her partial identity.

5. THE PARTIAL IDENTITY UNLINKABILITY PROBLEM

After the definition of the partial identity concept and its relationships to trust and reputation has been given, we are now in a position to define what we call the *partial identity unlinkability problem* (PIUP).

In section 1, we described two vulnerabilities affecting trust and reputation systems: the multiple identities and the change of identities problems. As far as we are concerned, these two vulnerabilities are closely related to the *unlinkability* concept described by Pfitzmann and Hansen in [9]. They define *unlinkability* as *"Unlinkability of two or*

more items of interest (IOIs, e.g., subjects, messages, actions, ...) from an attacker's perspective means that within the system (comprising these and possibly other items), the attacker cannot sufficiently distinguish whether these IOIs are related or not".

We use this definition of *unlinkability* made by Pfitzmann and Hansen and our definition of partial identity to formulate the PIUP:

Definition 5. The *partial identity unlinkability* problem (PIUP) states the impossibility that an entity, which is taking part in a system, is able to sufficiently distinguish whether two partial identities in that system are related or not.

It is easily observed that the *change of identities* problem is an instantiation of PIUP, i.e., an entity with an identity by which she is known to have a bad reputation, acquires another identity with a fresh new reputation so that other entities are unable to relate the entity to its former reputation. In a similar way, if an entity does not trust another entity, the latter can change her identity. Therefore, the former entity is unable to notice that the same entity which he used to trust (distrust) is behind the new identity, so the trust relationship is restarted.

Regarding multiple identities, a similar instantiation can be made, so that an entity holds several identities and has different reputations with each of them. Thus, another entity is unable to relate the different reputations that the entity has because it is unaware of all of the identities the entity has. PIUP relates to trust in the same way when multiple identities are considered. An entity can believe that she is trusting multiple entities in a given system (such as a specific marketplace), but she may be trusting the same entity with different identities without being aware of it.

5.1 The Straightforward Solution

PIUP is obviously solved by forcing the entities taking part in a system to use their real identity. Historically, a real identity has been used to uniquely identify persons [10].

If an entity is not allowed to change its identity, then trust and reputation assessments of this identity cannot be removed. Although the changing of real identities has always been possible as a way of erasing reputation, these changes are not cost-free and do not completely erase the reputation. For instance, there are some companies that change their name in order to erase their previous reputation. However, a link with the previous reputation can be made (e.g. looking at its employees in order to find employees of the former company).

Due to the impossibility of completely erasing reputation, new online services are emerging related to the management of the online reputation of an entity with a real identity. For instance, ReputationDefender[4] and Mamba IQ[5] provide services to report the online reputation of an entity with a real identity (individuals or companies). These services usually find information related to an entity searching in blogs, social networks, and audio and video pages. These services also give the entities advice on improving their online reputation.

However, the solution of forcing entities to use their real world identities exposes a great disadvantage: privacy loss. Fisher-Hübner and Hedbom in [3] define *privacy* as "*the right to informational self-determination, i.e. the right of individuals to determine for themselves when, how, to what extent and for what purposes information about them is communicated to others*".

Nowadays, in the era of global connectivity (everything is inter-connected anytime and everywhere) privacy is a great concern regarding identity management in the digital world. While in the real world everyone decides (at least implicitly) what to tell other people about themselves (after considering the situational context and the role each person plays), in the digital world users have more or less lost effective control over their personal data. Users are therfore exposed to constant personal data collection and processing without being aware of it.

6. A PRIVACY PRESERVING SOLUTION FOR PIUP

After the definition of PIUP and the privacy issues of the straightforward solution, we provide a privacy preserving solution to PIUP so that trust and reputation systems can be used without PIUP and preserving users' privacy.

The architecture we propose is based on supporting the building of trust and reputation through partial identities. Entities are forced to use only one partial identity in a given system. Therefore, an entity cannot get rid of trust and reputation assessments in that system. The real identities of the entities taking part in that system are not disclosed, except under special circumstances such as law enforcement. What is more, an entity is able to control what attributes of her partial identity are disclosed to other entities in the system (whenever the resulting attributes are still a partial identity, i.e., the resulting attributes can sufficiently identify this entity among the rest of the entities in the system, as explained in section 2).

We have defined some requirements to be fulfilled by our proposal for the solution of PIUP while preserving privacy. These requirements relate to functionality, privacy and security.

Regarding functional requirements, the following are established:

- *The building of trust and reputation.* The proposed solution must support the building of trust and reputation relationships among the entities in the system.
- *Trust and Reputation Model Independence.* The proposed solution must be independent of the trust and reputation models that the entities in the system are using.
- *Heterogeneous Trust and Reputation Models.* The proposed solution must allow different entities to use different trust and reputation models in the same system.

Regarding privacy requirements, the following are established:

- *Unlinkability of partial identities to real identities.* The proposed solution must ensure that the real identities of the members of the system are unlinkable to their partial identities by any other member of the system (except for special circumstances such as law enforcement).

[4] `http://www.reputationdefender.com/`
[5] `http://www.mambaiq.com`

- *Entity control over partial identity attributes.* The proposed solution must allow entities to have control over the attributes of their partial identities which are disclosed to other entities in a concrete system.

Finally, the security requirements established are:

- *PIUP avoidance.* The proposed solution must avoid the PIUP problem by forcing entities to use only one partial identity in a given system
- *Entity accountability.* Entities must be liable for their acts without affecting the privacy requirements explained before.
- *Authentication of Partial Identities.* Entities in a given system must be able to recognize to each other as a prerequirement for the establishment of trust and reputation among them.

Our proposed solution fulfills all of these requirements by defining a two-layer architecture for trust and reputation systems. There are two layers that make up the architecture: the identity management layer and the trust and reputation model layer. The identity management layer is in charge of providing the entities taking part in a trust and reputation system with partial identity management. It fulfills the requirements stated regarding privacy and security. This layer acts as a foundation for building trust and reputation models. The trust and reputation model layer is in charge of providing the actual trust and reputation models being deployed in the system. Trust and reputation are established through partial identities, following the definitions and relationships among partial identities, trust and reputation proposed in sections 2, 3 and 4.

We assume that entities communicate to each other following a secure connection (such as TLS), so that the data they exchange in their interactions is provided with basic security features such as integrity and confidentiality.

6.1 Identity Management Layer

The technical systems supporting the process of management of partial identities are known as Identity Management Systems (IMS) [10]. User-centric privacy-enhancing IMS are supposed to enable a user to control the nature and amount of personal information disclosed [2]. These systems are aimed at firstly providing the controlled pseudonymity of the users; and secondly, the reliability of the users.

Controlled pseudonymity implies unlinkability between the partial identity and the real identity of the entity behind the partial identity. Controlled pseudonymity also implies that partial identities of the same entity are unlinkable if they are used in different contexts.

The reliability of the users implies that at first, there is unlinkability between partial identities and the real identity of the entities behind them, but under special circumstances the issuer of the partial identity can make a partial identity and the real identity of an entity linkable.

Our solution to PIUP assumes that a user-centric privacy-enhancing IMS is running at the infrastructure level. In order to assure the feasibility of the solution we provide, we especially assume that the underlying infrastructure is Higgins[2], but any other user-centric privacy-enhancing IMS would be valid, such as CardSpace[3] and Bandit[4].

Higgins is an open source identity framework composed of three main parts described bellow.

- The *Higgins Selector* provides a simple way to manage partial identities and choose which partial identity to be used in a given context.
- *Identity Services* is composed of two kinds of services: Identity Providers (IdPs), that issue partial identities and validate these identities to the RPs; and Relying Parties (RPs), that are a set of APIs that allows services to check the identity of the entities that interact with them.
- *Identity Attribute Services (IdAS)* include services that allow an entity to determine the access control rights of every other entity when accessing each attribute of each partial identity she holds.

We propose a *fixed scheme* for using Higgins in order for our proposed architecture to fulfill the requirements previously stated relating to security and privacy. This fixed scheme for Higgins is the identity management layer for our proposed two-layer architecture. In this scheme, there are two main parties:

Higgins IdP. We assume only one intermediary, a third party trusted by all of the entities and the trust and reputation system they are taking part in. This intermediary is a Higgins IdP (or a federation of Higgins IdPs). In this sense, there is only one Higgins IdP in each trust and reputation system. The IdP provides partial identities to the entities taking part in the specific system. Entities must register using a real identity which the IdP will not reveal to others. The IdP is also in charge of forcing one entity to only hold a partial identity in this specific system.

Entities. Entities, which are in a given trust and reputation system, select and manage their own partial identities using the Higgins Selector. Moreover, entities also act as RPs that validate the partial identities of other entities through the IdP. Entities use Higgins IdAS to access attributes of other entities' partial identities. Entities also use the Higgins IdAS to set access control policies to their own partial identity attributes.

This fixed scheme provides the overall two-layer architecture with the fulfillment of the following privacy and security requirements:

- *Unlinkability of partial identities to real identities.* The Higgins IdP acts as an independent third party that must be trusted by the entities taking part in the trust and reputation system. Although entities register to using a real identity, the Higgins IdP does not make real identities publicly available. Therefore, the rest of the entities in the trust and reputation system are not able to link a partial identity used in the system to the corresponding real identity.
- *Entity control over partial identity attributes* is achieved by means of the Higgins IdAS service. The IdAS service allows entities to determine the access control rights over each attribute of a partial identity they

[2]Higgins Open Source Identity Framework `http://www.eclipse.org/higgins/`

[3]CardSpace `http://msdn.microsoft.com/CardSpace`

[4]Bandit `http://www.bandit-project.org/`

hold. Entities are able to choose to hide some of the attributes of a partial identity in a system as long as the resulting set of attributes is still a partial identity, i.e., it sufficiently identifies the entity among the set of entities in that system.

- *PIUP avoidance.* Only one Higgins IdP is allowed to issue partial identities in a trust and reputation system. The Higgins IdP avoids that a previously registered entity (using a real identity) is able to obtain a new partial identity. Therefore, there is no chance for an entity in a trust and reputation system to not be able to sufficiently distinguish whether two partial identities in that system are related or not.

- *Entity accountability.* Under special circumstances, such as law enforcement, the Higgins IdP can disclose the real identity of a misbehaving entity. Therefore, accountability is assured and entities can be punished if necessary. This leads entities to be liable for their acts and they will take this into consideration before misbehaving.

- *Authentication of Partial Identities.* Entities use the Higgins RP in order to authenticate the partial identities of the other entities taking part in the trust and reputation system. Therefore, entities are allowed to recognize to each other from interaction to interaction.

The solution exposed needs a trusted third party (the Higgins IdP) so that it is inherently centralized. However, in scenarios where PIUP cannot cause important issues such as money loss, user-centric privacy-enhancing IMS can also be used without a centralizing point. This can be achieved in scenarios where partial identities issued by different IdPs can co-exist in order to improve the overall privacy of the system by hiding real identities. Therefore, this centralized solution is only needed in scenarios such as electronic markets where PIUP is actually a serious problem which can lead to money loss.

Finally, the identity management layer can be classified as a hard security approach. This means that the solution tries to combat intruders (in this case entities that take advantage of PIUP) so that there are none in the system.

6.2 Trust and Reputation Model Layer

On the top of the identity management layer, we find the trust and reputation model layer. This layer is the one which implements the actual trust and reputation models being used in the system.

Trust and reputation models in this layer are based on the definitions of identity and partial identity and their relationship with trust and reputation detailed in sections 2, 3 and 4. In this sense, partial identities act as a foundation for the establishment of trust and reputation among the entities taking part in the system. Therefore, this layer fulfills *The building of trust and reputation* functional requirement.

The concept of partial identity is totally independent from the trust and reputation model being used. Therefore, *Trust and Reputation Model Independence* is also assured. In this sense, a privacy preserving solution to PIUP is provided without the needing of re-designing the trust and reputation models. However, as explained in sections 3 and 4, partial identities are part of the context in which trust and reputation take place. Therefore, trust and reputation models must be aware of partial identities in order to extract the information they need to compute trust and reputation.

In this sense, partial identities can be used by trust and reputation systems for identifying an interaction partner from interaction to interaction and building trust based on past interactions with her. For instance, Wang and Singh propose a formal model for trust in [18]. This model is based on past experiences (successful or not) which are converted into trust, distrust and an statistical measure of the certainty of both trust and distrust. They recognize entities from interaction to interaction by using an identifier for each entity. Therefore, the only adaptation needed by this model is to use partial identities as sets of only one attribute: the identifier for each entity.

Another example of a trust and reputation model which can be built using our two-layer architecture is the REGRET system [14] developed by Sabater and Sierra. This model takes into account not only past experiences but also other sources of information to assess trust and reputation. Concretely, REGRET uses the role that an entity is playing in an institutional structure as a mechanism to assign default reputation to the entities. In this sense, the role of the entities can be extracted from their partial identity (whenever entities decide to make it accessible to other entities).

These two examples clearly show that our solution could be used by existing trust and reputation models without requiring a new design of the model itself. Therefore, our solution fulfills the *Trust and Reputation Model Independence* functional requirement.

The trust and reputation model layer also fulfills the *Heterogeneous Trust and Reputation Systems* functional requirement. In this sense, there is nothing that prevents different entities from using different trust and reputation models in the same trust and reputation system. Entities are not forced to use a concrete particular trust and reputation model in a system. They could choose the trust and reputation model they prefer for a given system. Indeed, this fact opens the possibility of having multiple vendors of trust and reputation models to be used for different entities in the same system.

Finally, as explained previously, the identity management layer is centralized. It needs a trusted third party (the Higgins IdP). However, the trust and reputation model layer allows both centralized and distributed trust and reputation models. Furthermore, the identity management layer can be classified as a hard security mechanism while the trust and reputation model layer can be classified as a soft security mechanism, i.e., the identity management layer avoids the presence of entities taking advantage of PIUP, while the trust and reputation model layer acts as a social control mechanism that isolates misbehaving entities.

7. RELATED WORK

Rehák and Pěchouček [12] relate trust and identity by modeling trust context and identity representation. They mainly focus on scenarios with scarce resources such as sensor networks, in which an underlying identity infrastructure cannot be assumed. Jennings and Finkelstein [6] propose a unified identity for social software in business processes. They propose building this unified identity by mining data from different social silos. Once this unified identity is built, it can be used as a foundation for trust and reputation. These two approaches obviate privacy concerns related to

identity attributes.

Friedman and Resnick [4] propose a mechanism for preventing name changes in a social arena. They assume an intermediary, trusted by all of the entities in the specific social arena without revealing one's real identity. However, they simplify the concept of partial identity solely to identifiers. Other attributes that can play a crucial role when assessing the trust and reputation of an entity are omitted. Moreover, they do not consider that real identities should be revealed in special situations such as law enforcement.

8. CONCLUSIONS

In this paper, we propose formalized definitions of partial identities and their relationship to trust and reputation. Partial identities are a key concept for identifying entities. Moreover, they play a crucial role in trust and reputation, modeling part of the context where trust and reputation take place. In this sense, both trust and reputation are established through partial identities.

We also define the *partial identity unklinkability* problem (PIUP) based on partial identities. PIUP can be more or less harmful depending on the final domain of the application using trust and reputation models. In domains where users can be seriously harmed (e.g. in an e-marketplace by losing money) PIUP needs, at least, to be considered.

We finally propose a privacy preserving solution to PIUP which takes into account privacy concerns. It allows the building of trust and reputation through partial identities while preventing entities from getting rid of trust and reputation assessments in a given system. The real identities of the entities in a system are not disclosed except under special circumstances such as law enforcement.

9. ACKNOWLEDGMENTS

This work has been partially supported by CONSOLIDER-INGENIO 2010 under grant CSD2007-00022, and projects TIN2008-04446 and PROMETEO/2008/051. Jose M. Such has received a grant from Conselleria d'Empresa, Universitat i Ciència de la Generalitat Valenciana (BFPI06/096).

10. REFERENCES

[1] C. Castelfranchi and R. Falcone. Principles of trust for mas: Cognitive anatomy, social importance, and quantification. In *ICMAS '98: Proceedings of the 3rd International Conference on Multi Agent Systems*, page 72, Washington, DC, USA, 1998.

[2] S. Clauβ, D. Kesdogan, and T. Kölsch. Privacy enhancing identity management: protection against re-identification and profiling. In *DIM '05: Proceedings of the 2005 workshop on Digital identity management*, pages 84–93, New York, NY, USA, 2005. ACM.

[3] S. Fischer-Hübner and H. Hedbom. Benefits of privacy-enhancing identity management. *Asia-Pacific Business Review*, 10(4):36–52, 2008.

[4] E. J. Friedman and P. Resnick. The social cost of cheap pseudonyms. *Journal of Economics and Management Strategy*, 10:173–199, 1998.

[5] D. Gambetta, editor. *Trust: Making and Breaking Cooperative Relations*. Basil Blackwell, 1990.

[6] B. Jennings and A. Finkelstein. Digital identity and reputation in the context of a bounded social ecosystem. In *Business Process Management Workshops*, pages 687–697. Springer, 2008.

[7] A. Jøsang, R. Ismail, and C. Boyd. A survey of trust and reputation systems for online service provision. *Decis. Support Syst.*, 43(2):618–644, 2007.

[8] R. Kerr and R. Cohen. Smart cheaters do prosper: defeating trust and reputation systems. In *Proc. of The 8th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2009)*, pages 993–1000, 2009.

[9] A. Pfitzmann and M. Hansen. Anonymity, unlinkability, undetectability, unobservability, pseudonymity, and identity management – a consolidated proposal for terminology, Feb. 2008. v0.31.

[10] K. Rannenberg, D. Royer, and A. Deuker, editors. *The Future of Identity in the Information Society: Challenges and Opportunities*. Springer Publishing Company, Incorporated, 2009.

[11] L. Rasmusson and S. Jansson. Simulated social control for secure internet commerce. In *NSPW '96: Proceedings of the 1996 workshop on New security paradigms*, pages 18–25, New York, NY, USA, 1996. ACM.

[12] M. Rehák and M. Pěchouček. Trust modeling with context representation and generalized identities. In *CIA '07: Proceedings of the 11th international workshop on Cooperative Information Agents XI*, pages 298–312, Berlin, Heidelberg, 2007.

[13] P. Resnick and R. Zeckhauser. Trust among strangers in Internet transactions: Empirical analysis of eBay's reputation system. In M. R. Baye, editor, *The Economics of the Internet and E-Commerce*, volume 11 of *Advances in Applied Microeconomics*, pages 127–157. Elsevier Science, 2002.

[14] J. Sabater and C. Sierra. Social regret, a reputation model based on social relations. *SIGecom Exch.*, 3(1):44–56, 2002.

[15] J. Sabater and C. Sierra. Review on computational trust and reputation models. *Artificial Intelligence Review*, 24(1):33–60, 2005.

[16] J. Sabater-Mir, M. Paolucci, and R. Conte. Repage: REPutation and imAGE among limited autonomous partners. *JASSS - Journal of Artificial Societies and Social Simulation*, 9(2), 2006.

[17] C. Sierra and J. Debenham. An information-based model for trust. In *Proc. of the fourth int. joint conf. on Autonomous agents and multiagent systems (AAMAS 2005)*, pages 497–504, New York, NY, USA, 2005. ACM.

[18] Y. Wang and M. P. Singh. Formal trust model for multiagent systems. In *IJCAI'07: Proceedings of the 20th international joint conference on Artifical intelligence*, pages 1551–1556, San Francisco, CA, USA, 2007. Morgan Kaufmann Publishers Inc.

Dispatching Agents in Electronic Institutions

Hector G. Ceballos
Tecnologico de Monterrey
Ave. E. Garza Sada 2501
Monterrey, Mexico
ceballos@itesm.mx

Pablo Noriega
IIIA-CSIC
UAB Campus
Bellaterra, Spain
pablo@iiia.csic.es

Francisco J. Cantu
Tecnologico de Monterrey
Ave. E. Garza Sada 2501
Monterrey, Mexico
fcantu@itesm.mx

ABSTRACT

In Electronic Institutions [1], agents may be prevented from achieving their goals if other participants are not present in a given scene. In order to overcome this situation we propose the addition of an institutional agent in charge of dispatching agents to scenes through a participation request protocol. We further propose to endow this agent with the capability of instantiating new agents, thus providing grounds for a self-optimization of the system. Advantages of our proposal are illustrated with the implementation of an information auditing process.

Categories and Subject Descriptors

I.2.11 [**Distributed Artificial Intelligence**]: Multiagent systems

Keywords

Multiagent systems development environments. Electronic institutions. EIDE.

1. INTRODUCTION

The Electronic Institutions framework [1] developed in the Artificial Intelligence Institute of the Spanish National Scientific Research Council, (IIIA-CSIC), is a means to design and implement regulated open multiagent systems. The current framework is the outcome of more than a decade of developments and has been used to implement regulated MAS in several domains. For example, to support electronic auctions, to establish supply-chain virtual organizations, to agentify hotel and hospital management systems, to support participative experimentation, to model policy-making or to simulate human activity in archaeological sites. The framework has also served as a model-building environment for the discussion of topics like agent-based simulation, machine readable normative languages or autonomic computing. But in spite of this considerable variety of modes of use there have been very few published references to the underlying technology and, in particular, seldom any discussion of its expressive limitations and ways to circumvent them [2, 4].

This paper is one such discussion. We address the problem of deadlocks induced by improper institutional support in the follow-through of processes whose "most natural" representation may be as goal-directed workflows. In fact, we frame that problem in slightly more general terms: as the breakdowns produced by stalling or absent agents; and we advance a solution whose schema —an institutional agent with particular functionalities— may be reused *mutatis mutandis* to address similar problems and may also be coded as a standard functionality in the framework infrastructure. We believe that our solution should facilitate the adoption of the electronic institutions metaphor for the design of conventional MAS.

The structure of the paper is straightforward. The next subsections provide terminological and conceptual background. In Sec. 2 we present our proposal and in Sec 3 we describe a case study on information auditing to illustrate our proposal and present simple experimental results to back our claims. We finish with a brief discussion and comments on future work.

1.1 Electronic Institutions

For the purpose of this paper the EI framework may be described in terms of a conceptual model, a computational model and a software platform, EIDE, to specify and run electronic institutions [1, 5].

The conceptual model for electronic institutions assumes that the electronic institution determines a virtual space where agents interact subject to explicit conventions, so that institutional interactions and their effects *count as facts* in the real world. Because of this virtuality, it is assumed that all interactions within the electronic institution are *speech acts* expressed as illocutionary formulae. The electronic institution defines an *open MAS* in the sense that (i) it makes no assumption about the architecture and goals of participating agents (who may be human or software entities); and (ii) agents may enter and leave the institution at will, as long as the regimented conventions of the institution are met. Participating agents are subject to *role-based regulations* whose specification is given in terms of illocutions, norms and protocols. There are two classes of agents, internal and external. Internal agents act on behalf of the institution itself who is responsible for their behavior. External agents act on their own behalf and their internal state is inaccessible to the institution. Interactions are organized as repetitive activities called *scenes*. *Scenes* establish interaction protocols describing agent group meetings as transition diagrams whose arcs are labeled by valid illocutions. The *performative structure* captures the relationships among scenes describing those transitions agents playing certain role can make. Finally, *normative rules* describe the obligations an agent contracts while it participates in the institution. Agents may move from one scene to another, they may be active in more than one scene at a given time and they may perform different roles in different scenes.

The computational model for EIs defines a social (in-

stitutional) software layer between an agent communication platform (e.g. JADE) and participating agents. All *institutional communications* among agents are mediated by this platform. That institutional middleware is composed by three types of *infrastructure "agents"*: (i) An *institution manager* who centralizes valid communications and keeps track of the *state* of the institution, which is a data structure that contains the current values of all variables involved in the enactment of the institution. (ii) There are *scene* and *transition managers* for each, scene and transition, who handle the activation and persistence of scenes and transitions, and give access and exit to participants according to the local conventions; these managers mediate between the institution manager and the agent governors and keep track of the state of the institution as it applies to their particular context. (iii) One *governor* is attached to each agent and filters all communications between that agent and the institution; in particular, it directs valid illocutions to the corresponding scene managers and the institutional manager. The governor keeps a copy of the evolving state of the institution in order to apply regimented conventions on all speech acts its agent utters, and communicates to other infrastructure agents only those speech acts that comply with those conventions; thus the governor enables a change of the institutional state if and only if it admits a valid illocution from its agent.

The Electronic Institutions Development Environment, EIDE [5], consists of a graphical specification language, ISLANDER, whose output is an XML specification of an institution; a middleware AMELI [6], that takes an XML specification and enacts a runtime version of the institution with agents who run on a FIPA-compatible agent communication platform; a debugging and monitoring tool, SIMDEI, that registers all communications to and fro AMELI and displays and traces the evolution of the institutional state; finally an agent-shell builder, ABuilder, that from the XML specification produces an agent "skeleton" for each agent role. The skeleton satisfies all the (uninstantiated) navigation and communication requirements of the specification thus leaving the agent programmer to deal only with the implementation of the agent's decision-making logic at communication points.

The way these ideas are made operational in EIDE makes it possible to build complex regulated MAS. However, that operationalization corresponds more naturally to some types of MAS functionalities than to others. In this paper we propose one mechanism to deal with a type of situation that is currently difficult to handle in EIDE. Namely, the deadlock caused by the unavailability of an internal agent in a scene. Such deadlocks may happen in processes (performative structures) that require dedicated internal agents to follow-through subprocesses that involve single agents. For instance, the supervision of a patient's treatment through a medical protocol or, as exemplified in this paper, the auditing of the dossier of an individual as part of the academic evaluation process of a university. In general, this type of individual follow-through processes tends to appear whenever institutional interactions are organized in terms of individual agents' goals (e.g. each dossier has one agent who "pushes" the dossier through the steps of a pre-established workflow). Because in EIDE all agents that are present in a given scene share the state of the scene, when there is a process that involves private attention from the institution to one agent, an internal agent that enforces institutional conventions may be needed in all the scenes involved in the process. Moreover, currently there is no standard way of creating new internal agents in a running EI and furthermore, the current version of EIDE does not have the functionality of forcing an agent to act at any point, in particular, thus, it cannot force an agent to move from one scene to another, nor to terminate an agent. Consequently, deadlocks may arise when all available internal agents are already busy and also when no available internal agent enters the stalled scene.

To overcome this hurdle, the current solution in EIDE is to instantiate a new (sub) performative structure that corresponds to the private process each time that process needs to happen. In this solution, all the required internal agents are created automatically for every scene or transition in the process. However, the creation of substructures is expensive and in most cases involve having agents active in multiple scenes simultaneously, situation that is rather complex to program. Furthermore, this mechanism does not necessarily solve the deadlock induced by an available agent who stalls.

In this paper we propose another way of addressing that problem. It consists in the definition of a new internal *Dispatcher agent*, that keeps track of the need of internal agents (of those roles that may become necessary) and when requested by a scene manager, dispatches those that are available to the scenes that may need them, spawning new ones whenever necessary.

1.2 Agent Platform Services

Multi-agent frameworks have solved the problem of managing agent participation in several ways. For instance, FIPA has proposed a low-level solution for peer-to-peer communication based on services. On the other hand, multi-agent frameworks like Moise+ [8], MadKit [7] and Electronic Institutions have proposed upper-level solutions.[1]

The FIPA organization recommends a series of low-level services that any agent platform must implement to provide peer-to-peer communication: Agent Management System (AMS), Message Transport System (MTS) and Directory Facilitator (DF). The main role of an AMS is managing agent creation, deletion and migration on the agent platform, as well as maintaining the index of all agents identifiers in the platform. The MTS enables communication between agents internally or across different agent platforms. On the other hand, the DF functions as a yellow pages service on the agent platform. The DF considers agents as service providers and publishes the service descriptions provided by them. Even when the DF is an optional component of the agent platform, it is essential for finding the location of services in the multi-agent system. JADE and FIPA-OS are examples of frameworks that fulfill such recommendation.

In MOISE+ (ORA4MAS), organizations are modeled along three dimensions: structural, functional and deontic. Agents organize themselves in groups adopting roles that determine those missions they can or must perform in order to achieve the groupal goal. Social schemata allow to organize those missions and partial goals that must be performed/achieved

[1] A thorough review of how stalling and deadlocks are addressed in other platforms is beyond the scope of this paper, here we merely point towards the grounds that are common to most approaches and two environments that use a notion of interaction context assimilable to EI scenes.

in order to reach a common goal. Similarly, in AGR (Madkit) agents organize themselves in groups adopting one role and are limited to communicate only with other agents in the same group. Nevertheless, an agent can join multiple groups simultaneously. In Electronic Institutions, once inside a scene, agents are allowed to exchange messages only with other agents in that scene.

The three approaches propose a common space or context on which agents interact in order to achieve a single common goal, or multiple individual ones. Coordination requires controlling the entrance and exit of participants as well as the minimum/maximal number of agents playing certain role. Agents are free to decide which space(s) to join as their own goals dictate. However, none of the three approaches provides a solution for those cases when there are not enough agents for starting a group or scene, or when a group or scene is stalled waiting for one or more agents to continue. Agents in the stalled group need to communicate with other agents in order to invite them to join.

Our approach proposes the existence of a Dispatcher Agent in charge of directing the invitations made by agents in stalled groups/scenes. This agent optimizes the allocation of resources instantiating new agents when necessary and negotiating with available agents their participation.

2. REQUESTING AGENT PARTICIPATION

The context established by a scene in Electronic Institutions makes it difficult for agents in the scene to reach other agents. Such confinement creates a challenge: to warrant that all required agents be present in the scene. We formalize this problematic situation and propose a solution. We propose to institute an agent in charge of dispatching agents to scenes through a participation request protocol. This agent makes use of the low-level services recommended by FIPA and is coherent with the EI model and implementable in the current EIDE version.

2.1 Missing agents in scenes

Assuming that agents decide freely to enter or not in a scene, we may find two problematic situations in which participants would not achieve their individual goals: 1) not all the agents required for the scene are available, or 2) an agent in the institution is not aware that its participation is required in a particular scene. Both conditions can appear before the creation of the scene or during its execution. In the following we will only deal with the case of ongoing scenes.

Formally, the problematic situation would be defined as follows. There is an agent A_1 pursuing a goal G_1 that is currently playing role R_1 in scene S. Scene S is in state W_1 and the achievement of G_1 requires reaching state W_n. To do so, there is a sequence of illocutions $(M_1, ..., M_n)$ that must be issued by A_1 or by some other agent $(A_2, ..., A_n)$ playing roles $(R_2, ..., R_n)$ in S. Nevertheless, at state W_j the outgoing illocution M_j, $1 \leq j \leq n$, has for sender or receiver an agent playing role R_j for which there is no agent in the scene. We assume that there exists a state W_k, $1 \leq k \leq j$, at which the entrance of agents playing role R_j is allowed and in which A_1 is capable of keeping the scene on hold.

In order to reach W_n, agent A_1 sets $meansFor(G_3, G_2)$ and $meansFor(G_2, G_1)$, where $G_3 = \{holdAt(S, W_k)\}$ and $G_2 = \{agentsPlayingRole(R_j, S, Q)\}$. $meansFor(G_2, G_1)$ denotes that goal G_1 is in stand-by until G_2 is achieved. Likewise, $holdAt(S, W)$ is a goal that is satisfied when scene S reaches state W. Similarly, $agsPlayingRole(R, S, Q)$ is satisfied once there are Q agents playing role R on scene S, where the quantifier $Q \in \{\texttt{ONE}, \texttt{ALL}, \texttt{N=n}\}$, represents only one agent, any available agent, or exactly `n` agents, respectively.

In order to achieve the goal $agsPlayingRole(R, S, Q)$, the agent A_1 must request the participation of other agents through some protocol P. Such protocol P must achieve the agreement of Q agents to participate in S with role R. Protocol P might include agent selection and negotiation, that may be performed by A_1 itself or by another agent.

2.2 A Dispatcher Agent

We introduce the notion of *Dispatcher agent* as an intermediary agent that facilitates the achievement of those $agsPlayingRole(R, S, Q)$ goals owned by other agents. Let us represent the Dispatcher agent with the symbol A_D and denote its attributions with the role R_D. This agent keeps track of all agents on the institution through the *Agents* relation. Besides, A_D maintains the three following relations: *AgClasses*, *hasType* and *canPlay*. The set of agent classes $AgClasses = \{C_1, ..., C_n\}$ represent the software implementation of any participant, denoted by a source code class. Through $hasType \subset Agents \times AgClasses$, A_D keeps track of the agent class of every agent in the institution; it is assumed that every agent belongs to a single agent class. Finally, $canPlay \subset AgClasses \times Roles$ is used to know which roles may be played by an agent according to its agent class. The *canPlay* set may be built and updated by keeping track of participants in the institution, or may be known *a priori*.

A_D is capable of creating new instances of the agent class C_i through the action $Instantiate(C_i)$, which creates and enters an agent A_i in the institution. The configuration of A_D specifies, for each agent class, a maximum number of agents it can manage, denoted $MaxAgs(C_i)$. If $MaxAgs(C_i)$ is 0, A_D cannot instantiate agents of type C_i. The function $CurrAgs(C_i)$ counts the number of tuples $(AG_j, C_i) \in hasType$.

A_D implements two primitive operations for updating these relations. $RegisterAgent(A_i, C_i)$ inserts A_i in *Agents* and introduces the tuple (A_i, C_i) in *hasType*. On the other hand, $UnregisterAgent(A_i)$ removes A_i from *Agents* and the tuple (A_i, C_i) from *hasType*.

2.3 Processing Agent Participation Requests

We can assume that the dispatcher agent becomes aware of the $agsPlayingRole(R, S, Q)$ goal owned by the agent X through a protocol P_{Req}. In this way, A_D generates an *agent participation request* $APR(X, R, S, Q)$ whose purpose is committing Q agents to participate in S with role R. Given the previous definitions, A_D must determine if it can satisfy the request with the current set of agents.

Definition 1. An $apr = APR(X, R, S, Q)$ is *satisfiable w.r.t. Agents* if an only if, there is a set $AGS = \{AG_i | hasType(AG_i, C) \wedge canPlay(C, R)\ for\ 0 \leq i \leq m\}$, such that $m \geq 1$ for a quantifier $Q \in \{\texttt{ONE}, \texttt{ALL}\}$, or $m \geq n$ for a quantifier $Q = \texttt{N=n}$. Similarly, $APR(X, R, S, Q)$ is *unsatisfiable w.r.t. Agents* if $m = 0$ for $Q \in \{\texttt{ONE}, \texttt{ALL}\}$, or if $m < n$ for $Q = \texttt{N=n}$

Given the set of agents AGS that can satisfy *apr* and using a protocol P_{Inv}, agent A_D invites every agent in AGS

to participate in S playing role R.The acceptance of AG_i is represented by $Agree(AG_i, S, R)$, while its refusal is represented by $Refuse(AG_i, S, R)$. Once all agents in AGS have given a response, A_D proceeds to select the best agents for the scene.

The set of agents accepting the invitation, denoted $AccAgs$, is partially ordered by an operator $\succeq$ that calculates how suitable is an agent AG_i of type C for playing role R in scene S, denoted $AccAgs_{\succeq}$. Hence, $AG_i \succeq AG_j$ means that AG_i is better or at least as good as AG_j for the given scenario.

The ordered set $SelAgs \subseteq AccAgs_{\succeq}$ is constituted by the first n agents of $AccAgs_{\succeq}$, where $n = 1$ for $q =$`ONE`, $n = |AGS|$ for $q =$`ALL`, and $n \leq$ `n` for $Q =$ `N=n`.

Definition 2. An $apr = APR(X, R, S, Q)$ *is satisfied*, denoted $satisfied(apr)$, if $|SelAgs(apr)| \geq 1$ for $Q \in \{$`ONE`, `ALL`$\}$ or $|SelAgs(apr)| =$ `n` for $Q =$ `N=n`.

If apr is not satisfied w.r.t. $Agents$, the introduction of new agents in the system would solve the problem. This is possible if there exists at least one agent class C_i such that $canPlay(C_i, R)$ and $MaxAgs(C_i) > 0$. If there is no C_i with these properties, A_D will have to wait for new agents for a fixed period of time τ, after which it will declare the request *unsatisfied*.

Given that there may be more than one agent class capable of playing role R, A_D can use the same partial order criteria $\succeq$ for selecting the best class for the role R required in an apr. If apr is an agent participation request and $AgClss(apr)$ a partially ordered set $\{C_i | canPlay(C_i, R)\}$ w.r.t. $\succeq$, then A_D will choose the first $C_i \in AgClss(apr)$ for which $CurrAgs(C_i) < MaxAgs(C_i)$. If no C_i satisfies this requisite, the instantiation is not performed.

A_D determines the number of agents that should be instantiated to satisfy the request, denoted $NMissing$. If $Q =$ `N=n`, $NMissing = n - |SelAgs|$, meanwhile if $Q \in \{$`ONE`, `ALL`$\}$ then $NMissing = 1$. If $NMissing > 0$, A_D can instantiate and enter in the institution a *missing agent* through the execution of the primitive $Instantiate(C_i) : A_i$ for some $C_i \in AgClss(apr)$. These primitives make use of the Agent Management System (AMS) provided by any FIPA-compliant agent platform.

Agents entering the institution are invited to scenes held in stand-by due to an unsatisfied apr. Thus, if an agent AG_i of type C_i is created by A_D in order to satisfy $apr = APR(X, R, S, Q)$ and AG_i doesn't accept the corresponding invitation to S, C_i is removed from $AgClss(apr)$. If an agent created by A_D refuses all the invitations made during its logging in the institution, its access is denied. Agents exiting from the institution produce a revision of unsatisfied agent participation requests that might require the instantiation of new agents.

We distinguish between permanent and transient participants according to their patterns of entry and exit in the institution. Let's call *permanent participants* those agents that remain in the institution continuously while it is alive. On the other hand, *transient participants* are agents that enter the institution pursuing certain goals and exit once they have reached them. Agent classes representing permanent participants are identified by the set $PermAgCls \subset AgClasses$; similarly transient participants are denoted by $TranAgCls \subseteq AgClasses$.

Now we can establish necessary conditions to determine when an unsatisfied agent participation request justifies an agent instantiation.

THEOREM 1. *An unsatisfied $apr = APR(X, R, S, Q)$ can be satisfied through the instantiation of $NMissing$ agents if there is a subset $(C_i \cup C_j) \subseteq AgClss(apr)$ such that $NMissing \leq FSlots(apr)$, where*

$$FSlots(apr) = \sum_i MaxAgs(C_i) - CurrAgs(C_i, S) + \sum_j MaxAgs(C_j) - CurrAgs(C_j)$$

for $C_i \in (AgClss(apr) \cap TranAgCls)$ and $C_j \in (AgClss(apr) \cap PermAgCls)$. $CurrAgs(C_i, S)$ returns the number of agents with type C_i currently in scene S.

PROOF. Eventually, transient agents will leave the institution releasing slots that A_D can use for creating new instances, hence in the worst case where $MaxAgs(C) = CurrAgs(C)$ and a single $C \in AgClss(apr) \cap TranAgCls$ exists, the exit of all agents of type C will make $CurrAgs(C) = 0$ allowing the instantiation of the required agents.

$CurrAgs(C, S)$ allows to consider those agents of class C that will remain in S. For permanent agents, we cannot assume that they will exit from the institution, hence we can only count with the instantiation of $MaxAgs(C) - CurrAgs(C)$ agents of type C. □

The order in which invitations are issued is important when incoming agents have a limited capacity for attending invitations. Suppose that A_D is processing two agent participation requests apr_1 and apr_2 for the same role R, where $AgClss(apr_1) = AgClss(apr_2)$, and the maximum number of invitations an agent of type $C \in AgClss(apr_1)$ can take is one. If an agent is instantiated for class C and the invitation for apr_2 is sent earlier than the invitation for apr_1, the new agent will only attend the scene in apr_2. Similar instantiations and invitations might satisfy apr_2 and left apr_1 in hold if $NMissing_{apr_1} < FSlots(apr_2)$.

On the other hand, the refusal of agents for participating in an apr might produce an empty $AgClss(apr)$ set. This condition would allow A_D to consider apr unsatisfiable discarding it from its queue. Otherwise, an unsatisfiable apr_1 might block a subsequent apr_2 if $AgClss(apr_2) \subseteq AgClss(apr_1)$.

2.4 Request and Invitation Protocols

Agent participation is negotiated through two protocols, one for requesting agent participation (P_{Req}) and another for inviting agents (P_{Inv}). Both protocols must be executed in parallel with the scene that originated the request for agent participation.

Let us use the *DAgent* name for denoting the R_D role, call *ReqAgent* the role played by an agent requesting the participation of other agents and call *InvAgent* the role that an invited agent plays. Every agent in the institution must be able to play *ReqAgent* and *InvAgent* roles, meanwhile only one agent, A_D, is allowed to play the role *DAgent*.

Figure 1 shows the sequence diagram for P_{Req} between *DAgent* and *ReqAgent*. Figure 2 depicts the automata describing the request protocol where letters on arrows represent valid sequences of illocutions taken from figure 1, as well as the nested call to P_{Inv}.

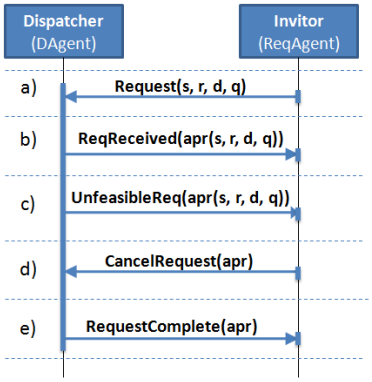


Figure 1: Sequence diagram for the request protocol.

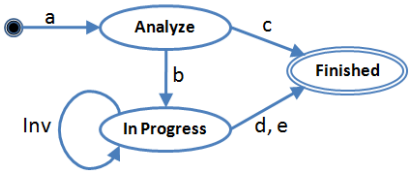


Figure 2: Automata for the request protocol.

Figure 3 shows a sequence diagram with segments of P_{Inv}. Figure 4 illustrates the automata describing the request protocol where letters on the arrows represent sequences of illocutions shown in Figure 3. P_{Inv} distinguishes between invitations to current agents and the instantiation and invitation of new agents. Figures are discussed below.

3. CASE STUDY

We used the Electronic Institution formalism for the automation of the auditing of an information repository. Several kinds of autonomous agents and human users participate in this auditing process. The process is initiated by external events and during its execution human intervention might be required. Rather than waiting for human users entry to the system, our approach enables autonomous agents to request human participation in order to achieve their goals.

A multiagent system for performing this auditing process was implemented with the tools developed in the IIIA [5]. Experiments and the results obtained are described at the end of this section.

3.1 Information Auditing

The information repository is managed by a *RepGuardian* agent that monitors changes on the repository and initiates the auditing process. The auditing process is driven by a specialized agent *Carrier*, and with the participation of other autonomous agents, *Auditor* and *Corrector*, as well as user agents representing human *experts* and information *authors*. A Carrier agent receives a notification about a record that has been added or modified in the information repository. The Carrier requests every available Auditor agent to check the internal consistency of the repository with respect to the

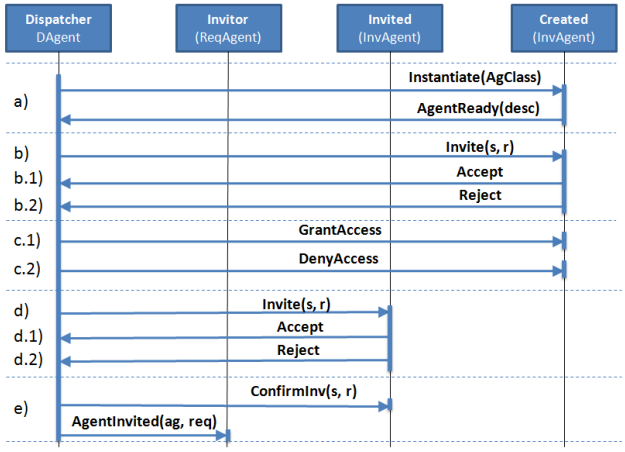


Figure 3: Sequence diagram for the invitation protocol.

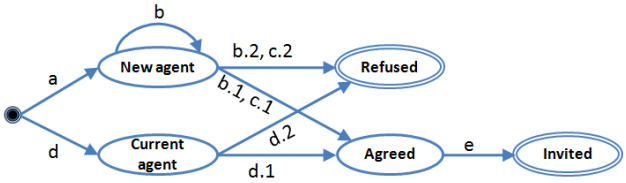


Figure 4: Automata for the invitation protocol.

auditing rules it knows. The Auditor agent responds to the Carrier whether informing that the record and the repository are consistent or returning a set of the inconsistencies detected. The type of inconsistencies are either internal inconsistencies of the record, or violations of rules defined for the entire repository; for instance, duplicity of records.

The Carrier agent chooses between sending the record to automatic correction with a Corrector agent, asking for expert assessment from a human expert, or notifying the author of the record of the possible inconsistency. The Corrector agent can apply the correction procedure or ask for expert assessment instead. In turn, the Expert user can modify the record or notify to its author. At the end, the decision made by the author is final. This decision model is depicted on Figure 5.

In this scenario we can detect some cases where human

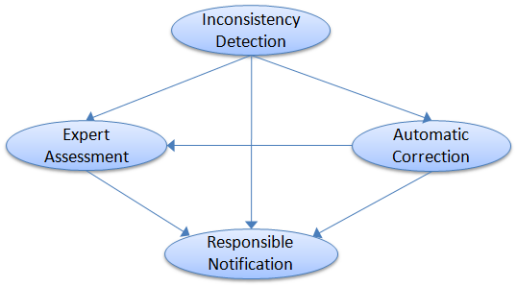


Figure 5: Decision model during information correction.

users are not present in the system when their participation is required. For instance, the user registering or modifying the record might have left the system by the time an inconsistency needs a final decision. Similarly, the expert user may not be logged in the system when an inconsistency is detected. A new auditing rule evaluated throughout the repository might require the assessment of expert users or users responsible for inconsistent records.

3.2 Implementing the auditing process

The process described above was specified with a performative structure *AuditingPS* that contains protocols for: triggering the auditing (NewInfo), detecting inconsistencies (Auditing), performing the corrections (Correction) and integrating results on the repository (Audited). *AuditingPS* and its protocols use the roles defined above: RepGuardian, Carrier, Auditor, Corrector, Expert and Author. Using the ABuilder tool [5], agent classes were generated for each role, except for Expert and Author roles which shared the same agent class, named *UserAgent*. Only the *RepGuardianAgent* was classified as permanent; the rest of the agent classes were considered transient.

It was possible to simulate the auditing of new pieces of information registered in the repository with this implementation. The simulation required to have a fixed set of user agents playing the roles of experts and authors for every possible human user that could be required in the process. Besides, every expert or author user participated in each Correction scene.

Given that human users responses to a request made when he/she was off-line might take entire days, the duration of Correction scenes was limited by a timeout after which the correction is considered to have failed. A User agent representing an expert or an author is not allowed to participate on multiple scenes simultaneously; nevertheless it can accept invitations to other scenes until reaching a given limit of invitations.

3.3 Implementing Agent Participation Request

The request for agent participation was implemented by developing: 1) an additional performative structure containing the protocols proposed in our approach, 2) the dispatcher agent, 3) an institutional service for instantiating new agents, and 4) new functionality for previously defined agent classes.

The new performative structure is assembled with four protocols that enable agents to: 1) log in, 2) request agent participation, 3) receive and answer invitations to scenes, and 4) log out. *AuditingPS* was inserted in this performative structure indicating that every agent should pass by the first three protocols before entering *AuditingPS*, hence remaining active in request and invitation protocols. Finally, after leaving *AuditingPS* they should pass by the log out scene. Protocols and roles specified in this performative structure are defined in section 2.4.

The *RepGuardian* agent implemented the functionality of the *DAgent* role with the characteristics described in section 2.2, the algorithm outlined on section 2.3 and the primitives described in both sections.

Agents developed for *AuditingPS* were augmented with the functionality of *ReqAgent* and *InvAgent* roles. A parameter on each agent class C denoted $MaxInv(C)$ was set to limit the maximum number of simultaneous invitations an agent of this class can accept.

Parameter	Low	High	Critical
Feeding rate	40 sec.	10 sec.	10 sec.
Expert revision	2-5 sec.	2-5 sec.	2-5 sec.
Author revision	20-25 sec.	20-25 sec.	20-25 sec.
MaxAgs(Carrier)	10	10	10
MaxAgs(User)	10	10	5

Table 1: Experiment configurations.

3.4 Experiments

To demonstrate the capabilities of the Dispatcher agent we prepared a test-bed with the system described above. We want to observe the capabilities of the *DAgent* for dispatching agents to scenes where human intervention is requested and for detecting unsatisfiable requests. In order to do so, we simulated different demand patterns on the system and manipulated the maximum number of agents permitted. An overloaded system is that in which information is fed faster than users are able to revise it. Thus we provoked that certain scenes stalled due to the lack of enough agents for all of them. Next we manipulated the maximal number of agents in order to generate unsatisfiable requests.

We defined a single RepGuardian and constant populations of auditor and corrector agents. One Carrier agent was instantiated for each information piece fed into the system. User agents playing the role of Expert or Author are created on demand up to a maximum of $MaxAgs(User)$. The same User agent representing a human user must participate in all the scenes where the user intervention is requested and it must wait to finish its work in a scene before proceeding to the next.

Our focus was on the Correction protocol, whose decision model is shown in Figure 5 and is explained in section 3.2. In our experiments, the power for instantiating Corrector agents was disabled, i.e. $MaxAgs(Corrector) = 0$. In consequence, the dispatcher informs the Carrier of the unfeasibility of requests for Corrector agents. Hence the Carrier requests an expert who in turn calls one author for correcting the record. In conclusion, every Correction scene is initiated by one Carrier and requires the participation of one expert and one author. All the agents remain in the scene until this finishes. User agents were limited to accept up to three invitations, i.e. $MaxInv(User) = 3$.

We prepared three system configurations. The first configuration gives us a reference of how the system would behave under low demand. In the second we have an information feeding rate higher than revision time, which we expect to generate several stalled scenes. And the last configuration has a reduced number of User agents for detecting unsatisfiability of requests. Parameters for the three configurations, labeled Low, High and Critical respectively, are shown in Table 1.

3.5 Results

Using the configurations given above we ran experiment rounds auditing 50 new information pieces in order to measure the behavior of agents and measure the performance in the Correction scene. We observed the maximal number of simultaneously stalled scenes, i.e. scenes in hold due to a request for agents, and calculated the average conclusion time for these scenes. Aditionaly, we observed the maximum

Observation	Low	High	Critical
Max. stalled scenes	2	7	10
Avg. scene time	30 sec.	61 sec.	197 sec.
Max. active Carriers	2	10	10
Max. active Users	2	10	5
Max. active Experts	1	5	4
Max. active Authors	1	6	2

Table 2: Experimental results.

number of concurrent Carrier and Users agents, as well as the maximum simultaneous number of User agents playing the Expert or Author role; recall that experts and authors use the User agent class for participating in the system. Results for the three configurations are shown in Table 2.

The first configuration showed only one Correction scene most of the time, and reached the maximum of two at some point of the simulation. The number of simultaneously stalled Correction scenes and Carrier agents is the same as long as Carrier agents are in charge of creating the Correction scene. Only one expert and one author were active in the system at the same time, authors and expert were released once the correction scene finished and were instantiated again when a new Correction scene was generated.

In the second configuration the reduction on the feeding rate produced more stalled scenes and a higher utilization of agents. The maximal number of Carrier and User agents was reached. This time the maximum number of stalled scenes didn't match the maximal number of Carriers agents because they were busy participating in other scenes of the auditing process. The average conclusion time for scenes was doubled as long as busy experts kept in hold at most two scenes meanwhile they were attending another scene. Figure 6 shows the behavior of the population of Carrier and User agents, broke down on Expert and Author roles, for this configuration.

In the third configuration, after approximately twelve successful evaluations the entire system stalled. At this point we observed the five user agents playing the role of Expert leaving no space for authors. Ten scenes were stalled, five of which had an Expert agent and the other five had a single Carrier agent waiting. In this case the dispatcher agent was not capable of determining the unsatisfiability of the requests for authors as long as it was expecting that some User agent left the institution for instantiating an agent to play the author role. The rest of the scenes made use of the five available Expert agents not allowing the instantiation of a new User agent for playing the role of Author. All these scenes finished thanks to the timeout of 200 seconds, as can be observed in the average termination time for these scenes.

This last scenario make us conclude that it was necessary to reserve agent slots for authors in order to conclude the scenes satisfactorily. Even when the participation of experts and authors is not assured in all the scenes, we should be capable of indicating it to the dispatcher agent in order to prevent the deadlock.

4. DISCUSSION

Low-level services like the Directory Facilitator only answer questions about the current set of agents in the system. On agent platforms implementing this kind of services an agent must search agents in term of the services they can provide. For Electronic Institutions such service could represent playing a role at certain type of scene. Nevertheless, the agent should be capable of negotiating the participation of other agents directly with them. In our approach this negotiation is centralized and organized in the *DAgent* which allows to detect unsatisfiable requests at some extent.

Another advantage of our approach is that populations of agents can be adjusted on line according to the current demand. This is possible thanks to the ability of transient agents for leaving the institution when they are idle and to the *DAgent*'s ability for instantiating agents when they are required.

Another way of optimizing the system performance is using a well known protocol for resource allocation, the ContractNet protocol [9]. ContractNet can be adapted for being used as agent request protocol. This can be done by narrowing the signal task announcement to agents of class $C \in AgClss(apr)$ and making an analogy between *invitation acceptance* and *task assignment*, where the task abstraction would be expressed as $playRoleIn(AG, R, S)$. The bid specification can include information about the time that would take a bidder to get to the scene. Finally, every agent AG_i chosen from $AccAgs(apr)$ receives an AWARD message for $playRoleIn(AG, R, S)$, and AG_i is added to $SelAgs(apr)$. Agents making a bid for a task of this type commits since that moment to attend to the scene if it is awarded.

5. CONCLUSIONS

We presented an approach for facilitating goal achievement by agents on an Electronic Institution. The type of situations prevented are those where a missing agent prevents the on-going execution of a protocol. Our approach consists in introducing an agent that dispatches available or new agents to those scenes.

We proposed necessary conditions for the instantiation of agents to satisfy an agent participation request. Nevertheless, experiments showed that such conditions are not sufficient when the scene requires the simultaneous participation of further agents of the same class. More work on this direction is needed. Even though, agent instantiation controlled by the *DAgent* showed its potential for optimizing dynamically the populations of agents in the system.

Advantages of our approach were illustrated in an auditing scenario with particular characteristics. For example, the interaction was not initiated by human users but by autonomous agents. As it was shown, our approach allowed the participation of just the necessary agents on each scene and avoided having idle agents in the system.

5.1 Future Work

The request protocol can be extended to deal with future agent invitation and not just current invitations. By so doing, we could prevent the deadlock of concurrent scenes. Another option would be developing an algorithm for pruning the directed cyclic graph representing an EI protocol or scene. That would produce a reduced version of the protocol when agents for certain role are missing. A pruned protocol that doesn't reach the final state would indicate an unsatisfiable scene execution. For instance, a Correction protocol on which the participation of Corrector, Expert and Author agents is pruned, could be detected *a priori* as unsuccessful.

An institutional model of public information for scenes and agents can be used to improve our proposal. Public

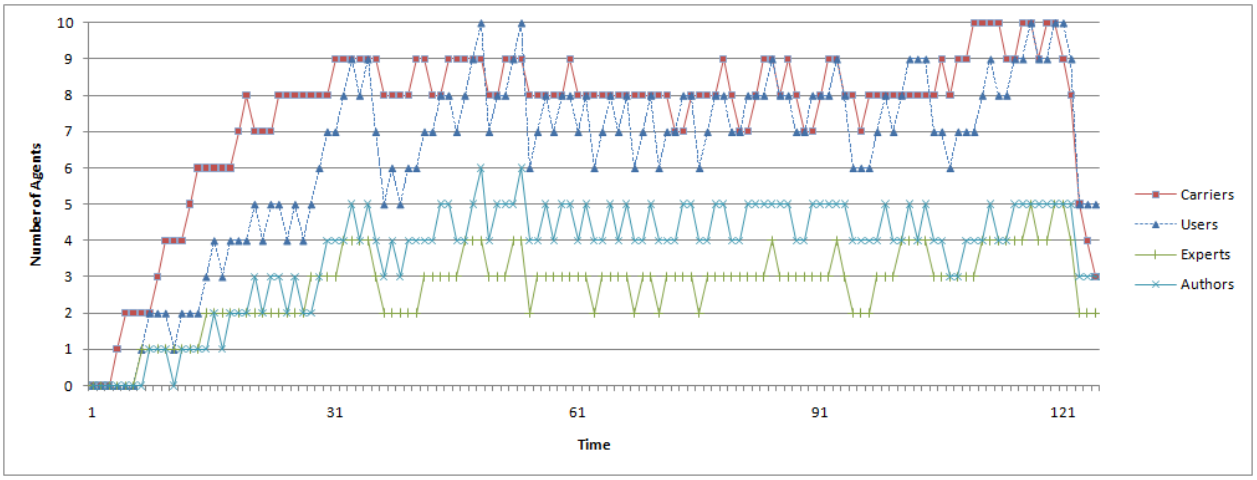


Figure 6: Agent populations on the high-demand configuration.

information of the scene and the participants may be used by the *DAgent* to narrow the announcement task, and by invited agents to calculate their bids for participating in a scene. For example, a Corrector agent that knows a rule for correcting inconsistencies of a single type, should be directed only to scenes where an inconsistency of that type is being corrected.

Agent descriptions formalized through a Description Logics [3] system would allow the generation of agent profiles describing the properties that potential participant agents should have. For instance, knowing that there are three auditing rules for the repository and that every Auditor agent can only handle one single rule, the request for auditor agents for all the type of auditing rules would generate three agent profiles, one for each rule. An instance of each profile would be enough for assuring a complete auditing of each new record.

6. ACKNOWLEDGMENTS

This paper was partially funded by the Spanish Ministry of Science and Innovation AT (CSD2007-0022, INGENIO 2010) and EVE (TIN2009-14702-C02-01), by the Generalitat de Catalunya 2009-SGR-1434, by the Mexican Council for Science and Technology and by the Tecnologico de Monterrey.

7. REFERENCES

[1] J. Arcos, M. Esteva, P. Noriega, J. Rodriguez-Aguilar, and C. Sierra. Engineering open environments with electronic institutions. *Engineering Applications of Artificial Intelligence*, (18):191–204, March 2005.

[2] J. L. Arcos, P. Noriega, J. A. Rodríguez-Aguilar, and C. Sierra. *E4Mas through Electronic Institutions.*, pages 184–202. Number 4389. Springer, Berlin / Heidelberg, 08/05/2006 2007.

[3] F. Baader. *The Description Logic Handbook: Theory, Implementation, and Applications.* Cambridge University Press, September 2007.

[4] J. Campos, M. López-Sánchez, J. A. Rodríguez-Aguilar, and M. Esteva. *Formalising situatedness and adaptation in Electronic Institutions.*, volume LNCS 5428, pages 126–139. Springer-Verlag, 2009.

[5] M. Esteva, J. A. Rodríguez-Aguilar, J. L. Arcos, C. Sierra, P. Noriega, and B. Rosell. Electronic institutions development environment. In *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems. AAMAS-08*, pages 1657–1658, Estoril, Portugal, 12/05/2008 2008. International Foundation for Autonomous Agents and Multiagent Systems, International Foundation for Autonomous Agents and Multiagent Systems.

[6] M. Esteva, J. A. Rodríguez-Aguilar, B. Rosell, and J. L. Arcos. Ameli: An agent-based middleware for electronic institutions. In *Third International Joint Conference on Autonomous Agents and Multi-agent Systems (AAMAS'04)*, New York, USA, July 19-23 2004.

[7] O. Gutknecht and J. Ferber. MadKit: Organizing heterogeneity with groups in a platform for multiple multi-agent systems. Technical Report R.R.LIRMM 9718, LIRM, December 1997.

[8] R. Kitio, O. Boissier, J. F. Hubner, and A. Ricci. Organisational artifacts and agents for open multi-agent organisations. In *Coordination, Organizations, Institutions, and Norms in Agent Systems III*, pages 171–186, 2008.

[9] R. G. Smith. The contract net protocol: High-level communication and control in a distributed problem solver. *IEEE Transactions on Computers*, C-29(12), December 1980.

Agents with cognitive capabilities for social simulation

A. Caballero*
Facultad de Informática
Universidad de Murcia
30100 Murcia
acaballero@um.es

J.A. Botía
Facultad de Informática
Universidad de Murcia
30100 Murcia
juanbot@um.es

A. Skarmeta
Facultad de Informática
Universidad de Murcia
30100 Murcia
skarmeta@um.es

ABSTRACT

Multi-Agent Based Social Simulation (MABS) is a paradigm devoted to use agents as the modeling metaphor to simulate autonomous entities in a social world composed by a number of independent and interacting entities. Such models try to reproduce real environments and situations of interest within such environments. Most MABS platforms used today (e.g. MASON, Repast, NetLogo) see agents as very simple entities. However, there are situations in which a more intelligent kind of agent is needed. For example, when a society of persons with different roles and high level behaviours must be model. In this paper, we address how to incorporate agents with cognitive skills into MABS.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous

General Terms

Experimentation

Keywords

agent based simulation, social simulation, cognitive modeling

1. INTRODUCTION

Social simulation is a research field that applies computational methods to study issues in the social sciences. The explored issues include problems in sociology, political science, economics, anthropology, geography, archeology and linguistics [16]. Nowadays, several social simulation platforms are used (e.g. MASON, Repast[1], Netlogo[2], etc). All of them are based on the concept of a very simple agent, totally reactive, without any cognitive capability. However, in a great amount of systems, developed under the artificial intelligence branch which attempt to reproduce intelligent human behaviours, the agents used in the modeling process show autonomy, proactivity, adaptability, reasoning, planing capabilities, and so on [8]. Nowadays, this is accomplished by the so called cognitive architectures. Such systems manage decision making , memory, learning, among others (e.g. SOAR[10], ICARUS [11], ACT-R[2], etc.) [15]. They are both conceptually heavy models and intensive CPU consuming approaches. This last fact makes them unfeasible for social simulations with a high number of agents in the society. Then, other approaches, using a society of agents that executes cognitive architectures, are needed.

Our application domain is Ambient Intelligence. Ambient Intelligence (AmI) is a new vision in which people are surrounded by embedded intelligent objects within an environment that is able to recognize and to respond to different individuals [13]. From an ambient intelligence simulation perspective, where it is required to reproduce intelligent behaviours simulating humans like as possible and necessary, it is very important to provide higher-level intelligent behaviours. Moreover, on the development of Ambient Intelligence [1] application and services, one important aspect is their validation by final users. AmI systems require of a subtle and intelligent interaction with final users. As such, the assessment of a correct system functionality is not trivial.

Living labs are, nowadays, the approach to choose when a small AmI system is considered. Systems of this type are those which may be find in Smart homes for example [7], when users are from one to five (i.e. a family). The main idea behind living labs is that user satisfaction is studied by reproducing artificially the real environment in which the AmI system and the user will interact (i.e. the smart home) and studying user reactions meanwhile she is living in such environment. The kind of AmI systems we are concerned with are large-scale AmI systems (i.e. those located within an intelligent building or a hospital). Obviously, it is nos feasible to artificially reproduce such huge living lab. In this case, the approach we follow to test functionality of services and applications within AmI systems is simulation. And the simulation paradigm we use is MABS (MultiAgent based Simulation). With this approach, we simulate the physical environment, the hardware of the AmI system (i.e. mainly sensors and actuators), and the users inside. The only thing which is real is the software we want to validate, which is connected to the simulation in real time.

Two simple examples of simulation scenarios we are developing in our lab are Ubik [14] and Cardinea[3]. In Ubik, we are interested on studying AmI systems on the intelligent building physical scenario, the simulation model incorporates workers on the building with different behaviour patterns depending on their role in the organization. There we investigate, as an example, the effect of intelligent mechanisms to guide people in situations like a fire by means of

*This research work is supported by the Research Projects TSI-020302-2009-43, TIN2008-06441-C02-02 and by the Fundación Seneca within the Program "Generación del Conocimiento Científico de Excelencia" (04552/GERM/06).

[1]http://repast.sourceforge.net/

[2]http://ccl.northwestern.edu/netlogo/

[3]http://cardinea.grupogesfor.es

audio or visual signals (e.g. an electronic message panel). In Cardinea, the situation is different, but the dimensions of the physical environment are similar. Here, physical hospitals with personal (care givers, doctors, assistants) and patients are simulated.

Both examples have in common that some users (agents in this case) are distinguished from the rest in terms of their skills. In the intelligent building example, some of the users may have received special instruction in order to coordinate evacuations. In the hospital example, some of the doctors may be in charge of rearranging personnel at the hospital to cope with a peak in the number of incoming patients. Both kind of users, when simulated in a MABS platform, need intelligent capabilities (i.e. coordination skills and intelligent decision making abilities in this case). These are cognitive capabilities. Such users may be modeled by means of cognitive agents, computational processes that act like certain cognitive systems or act intelligently according to a cognitive definition. Actions of cognitive agents can be the result of processes concerning to deliberative, coordination, learning, adaptation, planning, etc. capabilities of the entities. MABS based platforms like MASON do not support these cognitive capabilities. At least, mechanisms to reason and to communicate are required.

Adding cognitive capabilities to current social simulation platforms seems the intuitive way to obtain cognitive behaviours in social simulations. Several interesting scenarios, like Ubik and Cardinea, are implemented in MASON. Migration to other social simulation technologies implies losing all implementations and many interesting MASON functionalities oriented to control the simulation and to monitor the results. For that reason, the objective of the present work is to provide cognitive (and communicative) capabilities, with low computational cost, to some agents into the social simulations given by MASON.

In order to provide cognitive capabilities, it is possible to choose traditional style architectures like SOAR, ICARUS, ACT-R, or others that use BDI agents like Jason or 3APL[4]. BDI architecture is one of the best known approaches to develop cognitive agents. But, it is necessary to emphasize that it is not designed for that. As an agent-based computational model, it matches with the metaphor of agent, over which MASON is based on. Obtaining a social simulation system to support cognitive BDI agents, seems a very interesting idea. This way, [3] shows how Jason can be used to carry out BDI agent-based simulations. Jason is implemented in Java and is available as Open Source, distributed under GNU LGPL [4].

According to social scenarios defined by Ubik and Cardinea, and from a practical point of view, Jason has two important disadvantages: (1) it is not capable to support a high number of agents: each agent is running over a Java thread, and JVM imposes rigid restrictions in this way (MASON can simulate hundreds of thousands of agents), and (2) it is not possible to replicate the same simulation (in MASON the simulation can be replicated using the same random seed in different simulations). These are two very important elements for experimentation.

It is necessary to clarify that Ubik and Cardinea scenarios consider wide agent populations, with a high number of agents, but not all of them exhibit cognitive behaviours. The simulated scenarios consist of a high number of reactive agents and a reduced number of cognitive ones. This idea is discussed also by other authors like Kennedy *et al.* by means of RebeLand model [9, 5].

Then, the proposal of this paper is based on the integration of MASON and Jason to give cognitive capabilities to some agents into the current developed social simulations, with a great amount of social agents and a reduced number of cognitive ones, and where the replication of experiments is very important.

The rest of the paper is structured as follows. Section 2 highlights the main characteristics of MASON as an agent-based social simulation architecture. Section 3 presents the interesting features of Jason to social simulation and under the point of view of the integration with MASON. Following, in section 4, the paper gives some alternatives to integrate these two technologies. It proposes one of them to develop a concrete implementation to show the utilization of this approach. Section 5 discusses an implementation of an illustrative situation where a high number of simple reactive agents are guided by a reduced number of cognitive agents. The performance and behaviour of two types of agents are resumed using MASON simulation toolkits. Section 6, comments the relationships between the given approach and previous works of other authors. Finally, section 7 gives conclusions and shows some ideas for future work.

2. MASON

MASON is a single-process, discrete event simulation core and visualization library written in Java, designed to be flexible enough to be used for a wide range of simple simulations, but with a special emphasis on swarm multiagent simulations of many agents (up to millions). But, it does not support the representation of communicative and cognitive capabilities of the agents [12].

It is structured in two layers: model and visualization. In simulation model layer, it provides, among others, (1) a discrete-event scheduler, (2) a high-quality random number generator, and (3) a variety of fields which hold objects and associate them with locations. The visualization layer allows for display of fields and user control of the simulation. It separates the model features from the visualization and control functionalities. From this layer, a model defined in the previous layer can be treated as a self-contained entity.

2.1 Model layer: the environment

A MASON model is entirely contained within a single instance of a user-defined subclass of *SimState*. This class represents the environment and contains a discrete-event schedule (given by class *Schedule*). The schedule controls the simulation, providing a variety of breakpoint to agents running in the environment. Such breakpoints provide a single point to include own programming code used to customize their behaviours. Through executing these breakpoints, agents can sense the environment, other agents, or act, modifying the environment of it own status.

Two interesting simulation breakpoints are offered by *Steppable* and *Stoppable* interfaces. These interfaces can be implemented by the agents in order to run and stop into the MASON environment. *Steppable* defines the abstract method *step*, invoked when agent receives one tick from the *Schedule*. Besides, *Stoppable* defines the method *stop*, invoked when *Schedule* stops the agent.

[4]http://www.cs.uu.nl/3apl/

The order in which the *Schedule* arranges the execution of agents depends on the sequence of numbers given by the random number generator. The seed, used to generate random numbers sequence, can be specified. This way, a given MASON simulation can be replicated several times, using the same seed into different simulations. Besides, *Schedule* allows the environment to invoke the beginning of the next step of the simulation, providing the method *nextStep*.

Fields in MASON relate arbitrary objects or values of them with a location within a virtual space. Many of these fields are simple wrappers from simple 2D and 3D arrays. Also, these structure can provide sparse relationships between objects. The use of these structures is optional.

2.2 Visualization

Objects, developed in this layer, may examine model-layer objects, using an appropriated reference to the environment *SimState* given by class *GUISimState*. MASON visualizes the objects through displays: GUI windows which provide 2D and 3D views on the underlying fields. Also, it is possible to define some customized displays. It is very interesting for experimentation, because it allows for inspect to underlying model objects. It is a way to monitor the simulation step by step. For example, some real time charts can be provided, or data about simulation can be stored by a logger agent.

3. JASON

In Jason, the agent definition can be given in two complementary ways: (1) by means of its BDI representation, allowing the achievement of its cognitive capabilities, and (2) through its Java-code representation, allowing, on the one hand, its inclusion in Java-based social environments and, on the other hand, the maintenance of its own knowledge schemes, perceiving the environment and other agents, execution of the actions, among others. The cognitive representation of agents is managed by an interpreter for an extended version of AgentSpeak programming language (based on the BDI agent architecture), including also speech acts based inter-agent communication. Using Saci[5] or Jade[6] (for example), a Jason multi-agent system can be distributed over a network effortlessly [4]

Implementation of a BDI agent can include instances of certain data structures. The most relevant structures are (1) beliefs base, where agent stores all current beliefs, (2) set of events, which might trigger the execution plans, (3) plan library, where agent know-how is stored, (4) set of intentions, where agent stores their focus-of-attention, and (5) a queue of messages received from other agents [3].

However, from the point of view of social simulation, other components are needed. Jason gives an environment over which the agents run. The environment, and the agents running in it, can be controlled in two different predefined ways. Also, Jason gives the possibility to select the underlying agent architecture, that provides communication facilities among agents. Jason provides some predefined agent architecture but also permits to define a custom one.

3.1 Jason environment

The environment in Jason is a representation of the real environment in which the agents are running. In implementation terms, the life-cycle of each agent in Jason is executed over one independent thread in the JVM. For that reason, the number of agents running is limited by the characteristics of the JVM. The theoretical limit of threads in a JVM is about 1000. However, experimentally, this number is reduced to few hundreds, depending on the complexity of the reasoning processes of the agents.

The representation of the environment is supplied by *Environment* class. It is responsible (1) to maintain the state of the environment, (2) to simulate the execution of actions required by the agents, and (3) to give a symbolic representation of the state of the environment to the agents running in it. Jason environment is a passive entity. Only when agents sense the environment, its state is perceived by them. It never sends notifications to agents when its state changes.

This class has, among others, two methods to update the perception of the agents and to execute their actions. The first, *getPercepts*, is called by agents when they want to update their BDI perceptions taking into account their own knowledge representation, environment characteristics, etc. The second one, *executeAction*, is invoked by agents when they want to execute any action. Generally, actions can result from a BDI deliberative process.

3.2 Execution modes

In a MABS, several mechanisms are necessary to synchronize the reasoning cycle of agents and the actions that they take. Environment can be used to perform some kind of synchronization in this way. Jason offers two ways to manage synchronization of reasoning life-cycle of running agents: (1) asynchronous, where all agents continue the next reasoning cycle as soon as it has finished the previous one, and (2) synchronous, that defines steps to control the simulation, in the way that the next step will not begin until all agents finish the previous one.

The customization of the agent can be obtained by writing code in each breakpoint of the reasoning-cycle that synchronous execution controller gives. The synchronous controller is offered by the *TimeSteppedEnvironment* class.

In the synchronous execution mode, the end of the reasoning cycle of each agent can be captured in the method *receiveFinishedCycle*. This method is invoked by any agent when it finishes its cycle, obtaining a copy of the agent state. In terms of integration with MASON, this method can be used to carry out updating and revising processes of intentions and goals bases. Maybe, this is the most important breakpoint, using a synchronous Jason execution mode, into which cognitive features may be included, at the end of each step.

3.3 Support to communicative actions

Jason gives the possibility to send perceptions, from one agent to another, using AgentSpeak methods. It includes predefined predicates such as *.send* [4]. Communication acts are supported by the message transport mechanism implemented by agent architecture that it uses. Using agent architectures provided by Jason (Centralized, Saci, Jade), transporting of the messages is transparent to the programmer: it is given by the architecture. However, if the programmer does not use any architecture, or customizes the owner architecture, he must implement the required mechanisms to offer communication between agents, according to the communicative requirements of the system.

[5]http://www.lti.pcs.usp.br/saci/
[6]http://jade.tilab.com/

3.4 Customized agent architectures

In Jason, every agent has an architecture, responsible for the execution of actions and maintaining agent perceptions up to date. This architecture can be customized by any agent, extending the class *AgArch* supplied by Jason. Writing a new subclass is done by refine several methods, such as: *perceive*, *act*, *sendMsg*, *broadcast*, and *checkMail* [4]. These are the most interesting breakpoints, invoked by by the simulator controller, from the point of view of social simulation and possible integration with MASON.

By designing a customized agent architecture, it is possible to use only the BDI interpreter in standalone agents. Of course, communicative functionalities are not available by default. It should be added from a third party or reusing SACI or Jade.

4. SOME ALTERNATIVES TO INTEGRATE MASON AND JASON

There are several alternatives to integrate MASON and Jason in order to provide cognitive behaviours in social agent simulations. The most important ones, depending on the role played by each platform, are the following:

1. The simulation is controlled by MASON environment, where Jason environment (and the execution of cognitive agents supported by them) is executed like any other MASON agent. The tick is given by MASON environment, and each element (reactive agents and Jason environment) executes a step of its simulation.

2. The simulation is controlled by Jason environment, where MASON environment (and the agents supported by them) receives the tick simulation after all Jason agents. The tick is given by Jason environment, and each element (cognitive agents and MASON environment) executes a step of its simulation.

3. The simulation is controlled by MASON environment, without using Jason environment. All agents are MASON agents and only those with cognitive capabilities use functionalities of the BDI interpreter and, possibly, others supplied by a customized agent architecture.

Alternatives 1 and 2 require that a representation of all agents is available at both environments. There are two reasons for this requirement: (1) attributes of reactive MASON agents must be known by cognitive Jason agents, because they are used in their deliberative process, and (2) attributes of cognitive Jason agents must be mapped in non-real MASON agents because monitoring tools inspect agents running in the same environment, and Jason environment does not support a high number of agents, as it is required in this work. Also, it is interesting to keep monitoring facilities and tools given by MASON. In both cases, mapping some agents attributes into entities of the other environment seems to produce significant overload to the simulation at the end of each time step. Also, in both cases, a reduced number of cognitive agents with reasoning capabilities is used, given by Jason BDI interpreter.

However, from the point of view of implementation, the first alternative is not achievable because Jason environment, represented by class *TimeSteppedEnvironment*, does not permit to invoke the beginning of the next step of the simulation. (It does not offer any public functionality for that.) Besides, MASON environment provides also the method *nextStep* to begin the next step.

On the other hand, alternative 3 is based on an interesting yet simple idea. It considers that any cognitive agent is implemented like any other MASON agent but it is capable to commit an entire Jason reasoning-cycle using BDI interpreter. This approach does not use the Jason environment. Consequently, many functionalities delivered by Jason platform are not available in this case. Specially, communication features must be implemented by means of a customized architecture. It is not an easy task.

In this point, it should be pointed out that alternative 2 is capable to support reasoning and communicative capabilities for a reduced number of cognitive agents coexisting with thousands of reactive agents. But, programming efforts (and skills required) and overload produced in a large social simulation generate some preliminary doubts about its suitability in some scenarios. On the other hand, it seems that alternative 3 gives an inexpensive way to combine both technologies but it does not support any communicative action by default. It corresponds to the programmer to address this issue.

The following subsections comment on the implementation details of alternatives 2 and 3.

4.1 Jason controls the simulation

In this alternative, the simulation is controlled by Jason environment. The functionality of synchronous execution mode of the environment is obtained by an extension of the class *TimeSteppedEnvironment*. Cognitive agents specify their behaviours using AgentSpeak code. Reactive agents implement some interfaces like *Steppable* and *Stoppable* in order to run and stop in MASON environment.

This environment maintains (1) cognitive agents, that use a BDI interpreter and communicative primitives of Jason, (2) a reference to a MASON environment, that receives one tick when one Jason step terminates (using *stepFinished* method), and (3) a data structure to maintain mirrors of (references to) all reactive agents in MASON environment. This is necessary because cognitive agents need to know some attributes of reactive agents in all steps of the simulation. References to reactive agents must be accessible in all Jason simulation steps. In the same way, mirrors of cognitive agents, maintained in MASON environment, need to be updated from state of cognitive agents in each step. The mirrors updating process can be carried out at the end of the global step, captured by *stepFinished* method, once the MASON environment finishes the launching step.

The beginning of the reasoning cycle is captured by method *getPercepts* where the cognitive agents update their own base of perceptions (i.e. the interesting knowledge used in deliberative process in this step). From this breakpoint, an agent can sense the state of the environment and/or can request the knowledge structures in order to update its BDI percepts. This alternative requires to obtain any data coming from the reactive agents located at mirrors maintained in the Jason environment. This data is eventually required for a correct reasoning process on cognitive agents.

When BDI reasoning terminates, method *executeAction* is invoked. It is possible to put some code in this breakpoint to execute the action resulting from the BDI reasoning (i.e. conclusions of the deliberative process). Generally, conclusions of deliberative process can be (1) execution of internal

actions to change its own state or the state of the environment, or (2) messages to other agents. The receiver of such messages can be a cognitive agent (running in the same Jason environment) or a reactive one (running in MASON Environment). If the target agent is a cognitive one, it can use communicative capabilities offered by Jason. But, if the target is a reactive MASON agent, it needs an auxiliary data structure to communicate with it. This structure must be maintained in MASON environment, for example, and must be accessible by cognitive agents. In each simulation step, reactive agents sense the MASON environment and read this shared data structure.

The logic of reactive agents is encoded in the method *step* of the interface *Steppable* given by MASON. In this alternative, coordination between agents is supported by communicative capabilities given by Jason environment.

4.2 MASON controls the simulation

In this alternative, simulation is controlled by MASON environment. This environment maintains, at the same functional level, cognitive and reactive agents, without any duplicity of attributes of the agent state. All of these agents must implement some interfaces such as *Steppable* and *Stoppable* in order to run and to stop. The logic of them is encoded in the method *step* of the interface *Steppable* given by MASON. Cognitive agents do not use any Jason environment to reason and communicate. Only the BDI interpreter is used into each cognitive agent. To orchestrate the reasoning cycle and deliver some basics communicative functions, into each cognitive agent, an owner agent architecture is customized by extension of *AgArch* class of Jason. This customization defines method such as *perceive*, *act*, *sendMsg* and *checkMail*. (please, see 3.4).

In order to guarantee communication between cognitive and reactive agents (in a simple or in a complex way), the two types of agents must extend a common basic agent architecture. Reactive ones do not need to redefine its behaviour for methods *perceive* and *act*. Cognitive agents redefine these methods to customize its own reasoning cycle.

Both types of agents need to launch a reasoning cycle in each simulation step. The invocation of the new reasoning cycle can be made from the breakpoint given by the *step* method at the interface *Steppable* of MASON. The method *getTS().reasoningCycle()* given by *AgArch* class is the responsible to execute a reasoning cycle into a given agent.

Also, similar to previous alternative, cognitive agents need to update their percepts at the beginning of the reasoning cycle (supported by method *perceive*). Then, when BDI reasoning concludes, method *act* is invoked. Finally, *checkMail* breakpoint is activated to read messages that other agents have sent. In contrast, method *sendMsg* is executed when BDI interpreter finds *.send* primitive in the code of BDI cognitive agent. The functionality of the last two methods must be implemented by the customized agent architecture, according to the communicative requirements of the scenario.

4.2.1 Communication of agents without using Jason

Since any Jason environment is not used in this alternative, communicative capabilities must be under the responsibility of the programmer of the new agent architecture. Developing a complex, robust, and reliable message transporting system or reusing an existing one can be a very hard task. But, there are a number of scenarios when communicative requirements are very simple [14].

There are several available mechanisms to provide communications support (e.g. ACL, based on blackboard, tuplespaces[6], shared memory, etc.). A simple mechanism can be suitable to give communicative capabilities to several kind of scenarios (like Ubik and Cardinea), where communication between agents is very simple too. For these scenarios, this alternative proposes a shared memory based mechanism.

4.3 Discussion

This section pretends to clarify under which circumstances, one approach (either alternative 2 or 3, alternative 1 is out of the discussion) prevails in front of the other. The criteria we have used to articulate this brief discussion are the following: (1) efficiency of the simulation, (2) communicative requirements of the scenario, (3) capabilities to monitor agents, system behaviour during the social simulation, and (4) programming skills needed in order to maintain a good model productivity.

According to efficiency requirements (i.e. criteria 1), alternative 2 is clearly the worst. Mapping cognitive agents into Jason environment can produce a significant overload in the interchanged information between the two environments. At each time step, cognitive agents must update their corresponding mirror in the Jason environment data structure. This is an important reason if we consider the high number of reactive agents used in the social simulation scenarios commented before.

Following the necessities of interaction among agents (i.e. criteria 2), solution number 2 is good when the cognitive agents need advanced coordination mechanisms (e.g. decentralized planning or flexible coordination within teams of agents). Approach 3 is suitable for scenarios with simple communication requirements.

On the other hand, if the needed training is taken into account, it is considerably harder to reach a good practice on using solution 2. In such case, mastering the Jason environment and AgentSpeak language is a must. Besides, MASON programming is also required as an additional skill. However, option number 3 only requires mastering MASON and AgentSpeak. Details about Jason architecture are not needed as it is not used.

5. USING ALTERNATIVE 3

We have just discused that alternative 3 is specially suitable for a high number of agents, a few of them needed of cognitive capabilities, and with no necessity of a sophisticated communication scheme. Such features are exactly those which will be used in Ubik and Cardinea scenarios. This section will be devoted to illustrate, with a middle level complexity example, how to use such alternative.

5.1 Sheeps and Shepherds example

The example used here is based on the metaphor of the shepherds of sheeps. We have a physical space in the country, which is populated by a number of sheeps (i.e. reactive agents) belonging to different shepherds (i.e. cognitive agents). Both sheeps and shepherds continuously move in eight directions (N, NE, E, SE, S, SW, W, NW). Each sheeps move in a randomly assigned and fixed direction. It only changes its direction when a shepherd reach them and obligues it to move out to the stable (in such case, the sheep dissapear from the yard). Sheeps only obey to their corre-

sponding shepherds. Thus, they will only go to the stable when reached by its own shepherd.

As we have just mention, during the simulation of this scenario, sheeps and shepherds encounter with each other. Four types of encounters may occur: (A) a shepherd with another shepherd, (B) a sheep with another sheep, (C) a shepherd with a sheep of his flock, and (D) a shepherd with a sheep of another shepherd's flock.

Encounters of types A and B do not require any reasoning from shepherds, but C and D need it in order to determine what the shepherds does. If the encounter is of C type, the shepherd orders the sheep to go to the stable, changing the moving direction of the sheep. If the encounter is type D, the sheep is stopped and the shepherd informs the sheep's owner of the presence and location of the sheep. If the sheep is rather going to the stable than moving normally (because of a previous encounter with his shepherd), he does not stop, and the shepherd does not inform about its location.

Each shepherd can follow his own strategy to move towards the stopped sheeps of his flock (i.e. those which were previously reached by a different sheperd). For example, a shepherd can maintain a list of stopped sheeps locations so, when the list size is greater than a certain threshold T, the shepherd goes to their encounter.

This is a suitable example to show how, in a social environment, some agents could carry out cognitive processing while the others (a great majority of them), behave in a reactive way. This example includes a large number of reactive agents (sheeps) moving in given directions, and a small number of cognitive agents (shepherd) supervising them.

Each shepherd needs to carry out some deliberative processing in order to decide what action to execute. In this case, the BDI interpreter, as offered by Jason, is a suitable way to obtain the action corresponding to the percepts of the shepherd. Communicative requirements of this scenario are very simple: shepherds only need to inform, in some (not all) steps, the location of sheeps. For these reasons, approach 3, presented in section 4.2, seems useful at providing cognitive behaviours to agents in wide social simulations.

5.2 Cognitive shepherd agent

Shepherds are represented by cognitive agents, using the ideas given in section 4.2. It is necessary to provide agents with an architecture to execute a reasoning cycle and to perform basic communicative functions. Shepherds are coded in Java, extending the class *AgArch* given by Jason and redefining the methods *perceive*, *act*, *sendMsg* and *checkMail*. Strategical behaviour of the shepherds is written using AgentSpeak code, executable by the Jason BDI interpreter. But, this is not enough for agents to run within the MASON environment. For that, by extending the *ArArch* class, and implementing the interfaces *Steppable* and *Stoppable* given by MASON, the agent implementation is finished. Cognitive shepherds have to execute a reasoning cycle from the *step* method in order to perceive (the state of the environment and the other agents), to act (giving orders to the sheeps) and to communicate with other shepherds (reporting the locations of the sheeps).

The BDI code of cognitive shepherd agents, representing strategies D, is shown below:

```
//strategy D: finds a sheep of other flock
+step(_)
    : isShepherd(A) &
      foundSheep(B,S,DirX,DirY,PosX,PosY) &
      not (S == Escaping) &
      not (S == Stopped) &
      not inFlock(A,B) &
      inFlock(L,B) &
      not (A == B) &
      not (A == L) &
      not (L== B)
  <- .send(L,tell,localizacion(PosX,PosY));
     toStop(B).
```

When a shepherd agent receives the simulation tick, the method *step* is executed. From this method, the agent launches a reasoning cycle (one cycle per simulation step). First, the method *perceive* needs to update the base of perceptions adding the percept *foundSheep(B,S, DirX, DirY, PosX, PosY)*. This percept indicates that sheep B was found in the position PosX,PosY, its state is S, and its moving direction is DirX,DirY. It is worth to indicate that, when a shepherd initializes its base of perceptions, it needs to add possible new knowledge related to his flock: *inFlock(L,B)*[7]. These perceptions are shared by all shepherds in the system.

In the next phase of the reasoning cycle, the BDI interpreter executes BDI code and, eventually, produces actions. These actions must be interpreted by handlers supplied by the method *act*. The Java code in charge of adding these handlers sets up the state of sheep agents, according to the specific case.

Notice that strategy D implies sending one message to another shepherd. The predefined primitive *.send* triggers the execution of the method *sendMsg* in the sender agent. This method writes in the shared memory dedicated to guarantee the communication between shepherd agents. The receiver agents need to inspect the shared memory to receive messages. The shared memory is accessible to all agents in the MASON environment.

5.3 Evaluation of alternative 3 through experimentation

Experimental evaluations are focused on the study of the scalability of the approach, but not on the appropriateness of the strategies of the shepherds (this is not relevant in the context of this paper). In this way, two types of experiments were carried out: (1) evaluating the performance of the simulation for several sizes of populations of reactive agents, and (2) evaluating the performance for several numbers of cognitive agents.

These evaluations are based on the analysis of time variables (i.e. both real execution time and simulation time steps) when shepherds achieve that sheeps escape from the yard.

Simulation parameters are the following: d: dimension of the yard where sheeps and shepherds move; p: density of agents (including sheeps and shepherds) per unit of area of the yard; n: number of sheeps in the yard (they are obtained by using the density parameter and the area of the yard); nl: number of shepherds in the yard; and T: minimal number of stopped sheeps needed to the shepherd looking for them.

Some configurations of the simulation are compared in the next subsections. Experimentation parameters were set up according to the real requirements of Ubik and Cardinea.

[7] This percept indicates that sheep B is in the flock of shepherd L

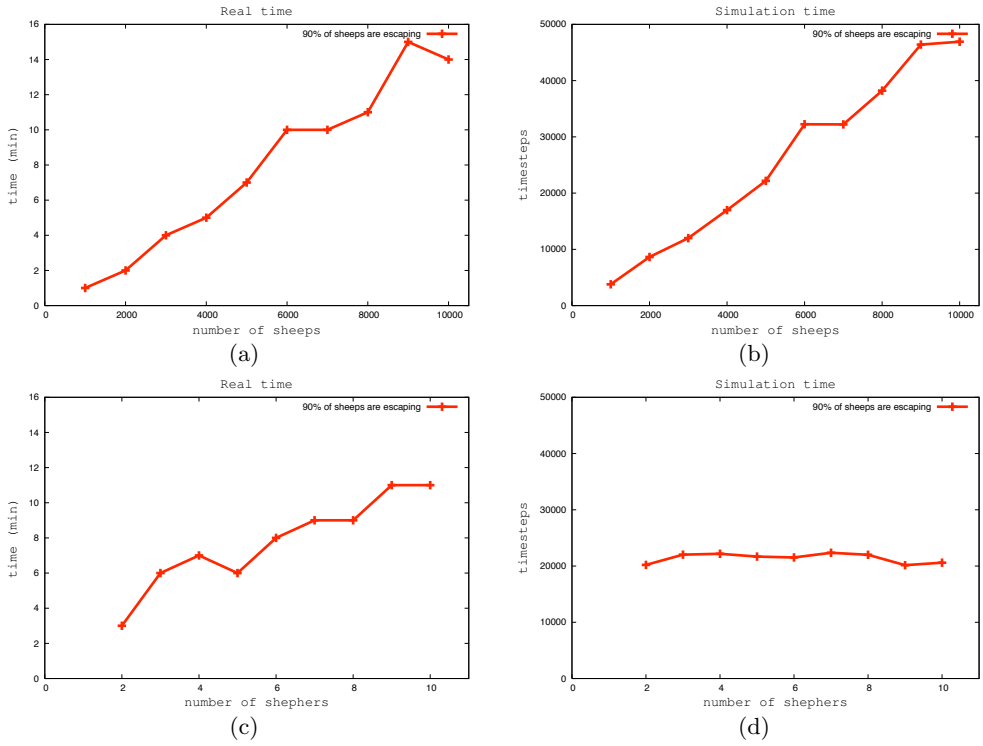


Figure 1: Real and simulation times when the 90% of sheeps escape from the yard, (a) and (b) for different numbers of sheeps $n = 1000, 2000, \ldots, 10000$, and the number of shepherds $nl = 4$; (c) and (d) for different numbers of shepherds $nl = 2, 3, \ldots, 10$, and the number of sheeps $n = 5000$.

This data was obtained keeping the density of agents in the yard $d = 0.3$, the minimal number of stopped sheeps needed to the shepherd looking for them $T = 0.02\ n$. The experiments are carried out on a simple laptop, with WindowsXP, an Intel Core 2 Duo Processor 1.66GHz and 1GB of memory.

5.3.1 Several number of reactive sheeps

Some simulations are carried out to compare the performance of the system when the number of reactive sheeps changes. Figures 1.a and 1.b show the real time and the simulation one when the 90% of sheeps were already put in the stable, for different numbers of sheeps $n = 1000, 2000, \ldots, 10000$, and $nl = 4$ shepherds.

These simulations evidence the capability of this approach to simulate large social simulations in a finite and short time. It suggest a linear relationship between times and the number of sheeps (social agents) in the simulation. Increasing of the real time seems non only related with reasoning process carried out into cognitives shepherd. It is logical that these times are increased when number of social agents and the dimensions of the yard are increased too. Simulation time is increased linearly when the number of sheeps and the size of the yard grows.

5.3.2 Varying the number of shepherds

Other experimental conditions are simulated, in order to study the performance of the approach when the number of cognitive agents is increased. The numbers of shepherds in these experiments correspond to the most usual numbers of cognitive agents in Ubik or Cardinea scenarios. 1.c and 1.d show the real time and the simulation one when the 90% of sheeps escape from the yard, for different numbers of shepherds $nl = 2, 3, \ldots, 10$, and $n = 5000$ sheeps.

These new experiments show the capability of this approach to simulate various cognitive agents into large social simulations in a finite and short time. The real execution time is increased when the number of cognitives shepherds grows, because a greater number of reasoning processes and communicative acts are carried out. Besides, the simulation time is kept around to the same value, independently of the number of shepherds in the yard. It this problem, the defined strategy of the shepherds does not improve the performance when a greater number of shepherds are in the yard. In these configurations (where the number of sheeps is the same), when the number of shepherds is increased, consequently, the number of sheeps per each shepherd is decreased. There is a greater number of cognitive agents but with a minor number of reactive ones per each of them. This is interesting to Ubik and Cardinea in order to describe the suitable number of agents performing cognitive tasks such as decision making, reasoning, coordination, planning.

In Sheeps and Shepherds scenario, coordination advantages do not reduce simulation time because a shepherd travels large distances to find his sheeps.

6. RELATED WORKS

Part of our work was to emulate cognitive behaviours into social simulations. Kennedy *et al.* suggest that some cognitive architectures such as SOAR or ACT-R are not suitable for representing a social human society [9]. All agents do

not need to perform cognitive actions. They propose three cognitive levels to classify agents according to their cognitive capabilities: simple, rule-based, and cognitive agents. These ideas are used in RebeLand [5] where agents are very simple.

Sun distinguishes between two types of cognitives architectures: software- (cognitive) and psychology-oriented. He proposes to use software-oriented ones in social simulation in order to understand human social behaviours. He illustrates his ideas by means of the architecture CLARION. Also, he points out the challenges facing cognitive social simulation. Most agent models in social simulation have been extremely simple. Using cognitive models (incorporating realistic tendencies, inclinations and capabilities of individual cognitive agents) can help to understand, in a realistic way, the interaction between individuals [15]. He suggests that more realistic and cognitive agent model, incorporating realistic tendencies, inclinations and capabilities of individual cognitive agents can serve as a more realistic basis for understanding the interaction between individuals

Bordini y Hubner [3] show how Jason can be used to simulate BDI agents. However, ideas offered in the present work suggest that Jason has limitations in the field of social simulations. In terms of implementation features, the number of agents running in Jason is limited by JVM (usually, social simulations involve a large number of agents). Also, distribution agent architectures (such as JADE or SCAI) impose a significant overload to Jason-based social simulations. Moreover, this approach requires more programming efforts than other typical agent-based simulation toolkits.

7. CONCLUSIONS AND FUTURE WORK

The integration approach presented in this paper is based on the idea of supporting cognitive behaviour of certain agents into wide social agent simulations. It proposes using Jason to provide cognitive capabilities to a reduced number of agents, running into a wide social simulation supported by MASON.

The paper compares two alternatives to combine Jason and MASON and proposes the most suitable types of scenarios for each of them, depending on communicative requirements. It uses the simplest one as an illustrative scenario where communicative requirements are reduced. It shows the guidelines to achieve cognitive behaviours in some real-scenario agents, such as Ubik or Cardinea.

Obviously, if communicative/cooperative requirements are increased, the alternative we use could not be as suitable as presented here. One solution is to improve the communication mechanism based on shared memory. It can be made better in two ways: (1) to use a more sophisticated method, like LINDA [6], where agent's communication achieves global coordination -for example, it may be interesting to considers tuple spaces (or any other type of associative memory) to ensuring mutual exclusion-, and (2) to give support to some high level human communicative social activities such as produce/hear an audible signal, write/read an informative panel, send/receive an email, etc.

Also, communicative/cooperative requirements in the simulation can be complex so, it can be necessary to adopt other alternatives (not used in the example). In this case, one of the agent architectures provided by Jason could be used at the risk of a significant overload to the simulation. On the other hand, this alternative needs an efficient mechanism to manage duplication of agent's information into environments of two different technologies. In this direction there is plenty of interesting work to be done.

8. REFERENCES

[1] E. Aarts and S. Marzano. *The New Everyday. Views on Ambient Intelligence*. 010 Publishers, Rotterdam, 2003.

[2] J. R. Anderson. Act: A simple theory of complex cognition. *American Psychologist*, 51:355–365, 1996.

[3] R. H. Bordini and J. F. Hübner. Agent-based simulation using bdi programming in jason. 2009.

[4] R. H. Bordini, J. F. Hübner, and M. Wooldridge. *Programming Multi-Agent Systems in AgentSpeak using Jason*. Addison-Wesley, 2007.

[5] C. Cioffi-Revilla and M. Rouleau. Rebeland: An agent-based model of politics, environment, and insurgency in mason. In *The annual meeting of the ISA's 50th Annual Convention Exploring the past, anticipating the future*, February 2009.

[6] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems: Concepts and Design. 3th edition*. Addison-Wesley, 2001.

[7] M. Friedewald, O. D. Costa, Y. Punie, P. Alahuhta, and S. Heinonen. Perspectives of ambient intelligence in the home environment. *Telemat. Inf.*, 22(3):221–238, 2005.

[8] N. R. Jennings and M. J. Wooldridge. *Agent Technology: Foundations, Applications and Markets*. Berlin: Springer Verlag, 1998.

[9] W. G. Kennedy, M. Rouleau, and J. K. Bassett. Multiple levels of cognitive modeling within agent-based modeling. In *Proceedings of the 18th Conference on Behavior Representation in Modeling and Simulation. Sundance, UT*, pages 143–144, April 2009.

[10] J. E. Laird. Extending the soar cognitive architecture. In *Proceeding of the 2008 conference on Artificial General Intelligence 2008*, pages 224–235, Amsterdam, The Netherlands, The Netherlands, 2008. IOS Press.

[11] P. Langley and D. Choi. A unified cognitive architecture for physical agents. In *AAAI'06: Proceedings of the 21st National Conference on Artificial Intelligence*, pages 1469–1474. AAAI Press, 2006.

[12] S. Luke, C. Cioffi-Revilla, L. Panait, K. Sullivan, and G. Balan. Mason: A multi-agent simulation environment. *Simulation*, 81(7):517–527, 2005.

[13] F. Scapolo, J. Leijten, K. Ducatel, M. Bogdanowicz, and J. C. Burgelman. Scenarios for ambient intelligence in 2010. Technical report, ISTAG, The Information Society Technology Advisory Group, 2001.

[14] E. Serrano, J. A. Botia, and J. M. Cadenas. Ubik: a multi-agent based simulator for ubiquitous computing applications. *Journal of Physical Agents*, 3(2), 2009.

[15] R. Sun. Cognitive architectures and multi-agent social simulation.

[16] S. Takahashi, D. Sallach, and J. Rouchier. *Advancing Social Simulation: the First World Congress*. Springer-Verlag: Berlin, 2007.

TRAMMAS: A Tracing Model for Multiagent Systems

Luis Búrdalo
Universidad Politécnica de Valencia
cno/ de Vera SN
46022 Valencia, Spain
lburdalo@dsic.upv.es

Andrés Terrasa
Universidad Politécnica de Valencia
cno/ de Vera SN
46022 Valencia, Spain
aterrasa@dsic.upv.es

Vicente Julián
Universidad Politécnica de Valencia
cno/ de Vera SN
46022 Valencia, Spain
vinglada@dsic.upv.es

Ana García-Fornés
Universidad Politécnica de Valencia
cno/ de Vera SN
46022 Valencia, Spain
agarcia@dsic.upv.es

ABSTRACT

Agent's flexibility and autonomy, as well as their capacity to coordinate and cooperate, are some of the features which make multiagent systems useful to work in dynamic and distributed environments. These key features are directly related with the way in which agents communicate and perceive each other, as well their environment and surrounding conditions. Traditionally, this has been accomplished by means of message exchange or by using blackboard systems. These traditional methods have the advantages of being easy to implement and well supported by multiagent platforms; however, their main disadvantage is that the amount of social knowledge in the system directly depends on every agent actively informing of what it is doing, thinking, perceiving, etc. There are domains, for example those where social knowledge depends on highly distributed pieces of data provided by many different agents, in which such traditional methods can produce a great deal of overhead, hence reducing the scalability, efficiency and flexibility of the multiagent system. Alternatively, this work proposes the use of event tracing in multiagent systems, as an specific communication mechanism to improve the amount and quality of the information that agents can perceive from both their physical and social environment, in order to fulfil their goals more efficiently. In order to do so, this work presents an abstract model of a tracing system and an architectural design of such model, which can be incorporated in a typical multiagent platform.

1. INTRODUCTION

Due to their flexible and adaptative behavior, multiagent systems are commonly applied to solve complex problems in dynamic and distributed environments. This is not only due to agents' individual features (like autonomy, reactivity or reasoning power), but also to their capability to communicate, cooperate and coordinate with other agents in the multiagent system in order to fulfil their goals. In fact, it is this social behavior, more than their individual capabilities as agents, what makes multiagent systems so powerful. Social abstractions such as teams, norms, social commitments or trust are the key to face complex situations using multiagent systems.

Mařik et al in [19] refer to the necessary knowledge, to give support to all these social abstractions as *social knowledge* and they also point that it plays an important role in increasing the efficiency in highly decentralized multiagent systems. Traditionally, these social abstractions are mostly incorporated to the multiagent system at user level; this is, from the multiagent application itself, by means of messages among agents or blackboard systems, without any specific support from the multiagent platform. These traditional methods may be easy to implement; however, each agent's social knowledge depends almost completely on the rest of the agents in the multiagent system actively informing of what they are doing, which has some major problems. First, it can lead to excesive overhead in some agents, specially in those situations where agents have to send their information to many agents because of not being able to determine which of them are really interested in receiving it. Second, it can also be difficult to trust the information provided directly by other agents using messages in open multiagent system where there is no way to know if an agent is well-meaning. This weak integration of high level social abstractions is also mentioned as an important flaw by Bordini et al in [4].

Apart from the mentioned social abstractions, another component of multiagent systems which is usually neglected or simply not considered to be important is the environment. Authors in [29] reivindicate the environment as a first class abstraction, since it provides the surrounding conditions for agents to exist, as well as an exploitable design abstraction for building multiagent system applications. The environment is also specially important in self adapting multiagent systems [28], which have to modify/adapt their structure and behavior, by adding, removing or substituting components while the system is running and without bringing it all down. In these cases, as pointed out by Dignum et al in [14], changes in the environment are the ones which trigger multiagent system reorganization and thus, this dynamic adaptation demands that systems can evaluate their own health. However, detecting these changes in the environment is not trivial.

Applications which extract information from the system at run time are already considered in the field of event driven architectures [18] and in the fields of *overhearing* [13] and

oversensing [23]. Also, the idea of an standard tracing system available for processes in a system already existed in the field of operating systems (and at present it is contemplated by the POSIX standard[15]). This work proposes the application of these concepts to the field of multiagent systems, where event tracing is still considered a mere facility to help multiagent system developers in the verification and validation processes. In order to do so, this document introduces TRAMMAS, an abstract event *TRAce Model for MultiAgent Systems* which lets all the components in the multiagent system share trace information among them, both at run time or by means of historic information (trace log files).

The proposed tracing model is based on the vision of multiagent systems by Omicini et al [21], which models multiagent systems on the basis of two main abstractions: agents and artifacts. On the one hand agents are autonomous, proactive entities that encapsulate control and are in charge of the goals/tasks that altogether define and determine the whole multiagent system behaviour. On the other hand, artifacts are those passive, reactive entities in charge of the services and functions that make individual agents work together in a multiagent system. The event tracing model proposed in this paper also considers aggregations of agents or agents and artifacs, as well as the multiagent platform itself are also susceptible of generating trace events and, as a consequence, it is rich enough to represent changes coming from any entity in the multiagent system.

This work also presents an architecture design, compatible with the TRAMMAS model. This architecture design offers all this tracing information as tracing services which entities in the multiagent system have to request when they are interested in receiving tracing information. Taking into account efficiency and scalability of multiagent systems which may use this tracing system, this architecture is designed to be integrated within the multiagent platform.

The rest of this paper is structured as follows: First of all, Section 2 reviews previous work carried out by different authors in the field of event tracing in multiagent systems. Later, Section 3 presents the trace model and Section 4 presents an architectural design to incorporate concepts in the model to a multiagent system. Section 5 presents an example of an agent market with certain information needs where event tracing is compared with other techniques. Finally, Section 6 will comment the conclusions of this work, as well as the future lines of work.

2. RELATED WORK

Tracing facilities in multiagent systems are usually conceived as debugging tools to help in the validation and verification processes. It is also usual to use these tracing tools as a help for those users which have to understand how the multiagent system works. Thus, generated events are mostly destined to be understood by human observers, who would probably use them to debug or to validate the multiagent system, and tracing facilities are mostly human-oriented in order to let multiagent system users work in a more efficient and also convenient way.

Some multiagent platforms provide their own tracing facilities. This is the case of the Sniffer Agent or the Introspector Agent provided by JADE[3], as well as the Conversation Center, the BDI Tracer or the DF Browser provided by JADEX[24]. Other multiagent platforms which provide their own tracing facilities are JACK[26] (Agent Interaction Diagramas, Design Tracing Tool, Plan Tracing Tool, etc.), ZEUS[12] (Society Viewer and Agent Viewer) and JASON[6] (Mind Inspector Tool).

Apart from those tools provided by multiagent platforms themselves, there are also many tracing facilities provided by third party developers. This is the case of Java Sniffer[27], developed by Rockwell Automation, a stand alone java application based on JADE's Sniffer Agent which is able to connect to a running JADE system in order to track messages among agents. Another third party tool based on JADE's Sniffer Agent is ACLAnalyser[10], which intercepts and stores for later inspection messages interchanged by agents during the execution of the application in order to detect social pathologies. Later work by the same authors [9] combines results obtained with ACLAnalyser with data mining techniques to help in the multiagent system debugging process. Tracking messages has also been used in [22] to extend the Prometheus methodology and the related design tool to help in detecting protocol violations and plan selection inconsistencies by means of tracing conversations among agents in the system.

Beyond tracing messages among agents, there are also more complex tool suites. This is the case of MAMSY, the management tool presented in [25], which lets the system administrator monitorize and manage a multiagent system running over the Magentix multiagent platform [1]. In [20], the authors describe an advanced visualisation tools suite for multiagent systems developed with ZEUS, although the authors also claim these tools could be used with other platforms (more precisely, with CommonKADS).

Lam et al present in [17] an iterative method based on tracing multiagent applications and a Tracer Tool to help the user understanding the way those applications internally work. The Tracer Tool can be applied to any agent system implementation, regardless of agent or system architecture, providing it is able to interface with Java's logging API (directly or via a CORBA interface). Results obtained with this method were presented in [16]. Bosse et al present in [8] a combination of this Tracer Tool with a Temporal Trace Language (TTL) Checker presented in [7]. This TTL Checker enables the automated verification of complex dynamic properties against execution traces.

As it has been shown, event tracing in multiagent systems is mostly conceived as a way to help developers in the debugging and validation processes, instead of a way to communicate and coordinate agents. That is why trace events are more human-focused than agent-focused tools. Also, most of the work on tracing multiagent systems has been carried out by multiagent platform designer teams and, as a consequence of that, most of existing tracing facilities are designed for a specific multiagent platform. In fact, even tracing facilities and tool suites developed by third party developers are usually not only implemented, but also designed, to work with a specific platform. As far as we know, there is not a standard, general tracing mechanism which lets agents and other entities in the system trace each other as they execute like the one defined by POSIX for processes [15]. However, the trace model presented in this paper is a platform independent abstract model, which has been thought and designed to be implemented in any platform, which does not mean that any implementation of this model will be platform independent too.

3. THE TRAMMAS MODEL

This section presents TRAMMAS, a platform independent trace model for tracing events in multiagent systems, considering a set of requirements previously described in [11]. Once incorporated to a multiagent system, either at platform or user level, this trace model lets agents and other entities in the system, as well as human developers/operators, generate and receive trace events generated by other entities in the system.

From the viewpoint of this model, a multiagent system can be considered to be formed by a set of *tracing entities* or components which are susceptible of generating and/or receiving *trace events*. As they execute, these tracing entities generate certain information related to their activity as trace events. Events generated by a tracing entity are recorded by the tracing system and delivered to other tracing entities, so that they can process all that information in order to fulfil their corresponding goals.

In any computing system, tracing can be a very expensive process in terms of computational resources. Multiagent systems are by nature highly decentralized systems, where the number of running entities and hosts can be high and thus, tracing such systems can be very expensive. In this context, the tracing process must be optimized in order to minimize the overhead it produces to the system, since a very sophisticated but excessively costly tracing system can become completely useless in practice. In order to prevent this kind of situations, it is necessary to let tracing entities decide which information they want to send or receive at each moment. Thus, the tracing system must support selective event tracing.

Event tracing, specially in open multiagent systems, has obvious security issues, since many of the events registered by the tracing system may contain sensitive information that can be used by agents to take advantage from the multiagent system or even to damage it. In order to prevent these situations from happening, each entity in the system which is able to generate trace events, should also be able to decide which entities in the system can receive its events. In order to do so, this trace model also incorporates a security mechanism based on the concept of authorization.

The rest of the section will describe in more detail the way in which these main concepts are modelled in this work.

3.1 Trace event

This model defines a *trace event* as a piece of data representing an action which has taken place during the execution of an agent or any other component of the multiagent system. Trace events are generated each time the execution flow of an application reaches certain instructions (tracing points) in its source code.

This model defines the following common attributes for each event:

- **Event type:** Trace events can be classified according to the nature of the information which they represent. This event type is necessary for tracing entities in order to interpret the rest of the data attached to the trace event.
- **Timestamp:** Global time at which the event took place, necessary to be able to chronologically sort events produced anywhere in the multiagent system.
- **Origin entity:** The tracing entity which originated the event.
- **Attached data:** Additional data which could be necessary to correctly interpret the trace event. The amount and type of these data will depend on the event type. Some trace events may not need any additional information.

Attending to the origin entity which generates them, trace events can be classified:

- **Domain independent trace events:** These trace events are generated by the multiagent platform itself and thus, they can be present in any multiagent system. Examples of domain independent trace events could be *new agent in the platform* or *new service request*.
- **Domain dependent trace events:** These trace events are designed ad-hoc for a specific multiagent system. Within a virtual market, an example of domain dependent trace event could be *sold product*.

Trace events can be processed or even combined in order to generate compound trace events, which can be used to represent more complex information. Both domain dependent and domain independent trace events can also be classified into simple and compound

3.2 Tracing entities

In this model, a tracing entity is defined as any component of the multiagent system or the multiagent platform which is able to generate or receive tracing information. Thus, from the point of view of the tracing process, any multiagent system is seen as a set of tracing entities. In this trace model, tracing entities can be classified in three main groups:

- **Agents**. Agents are all those autonomous and proactive entities which define the multiagent system behaviour. This category includes not only all of the individual application agents in the multiagent system, but also those which may be part of the multiagent platform.
- **Artifacts**. In this model, artifacts are all those passive elements in the multiagent system which are susceptible of generating events at run time or receiving them as an input [21]. Artifacts model elements of the multiagent system such as databases, resources modelled as web services, physical sensors and actuators and so on. Two or more artifacts can be combined in order to perform more complex tasks and they are also susceptive of generating or receiving trace events as a tracing individual. From the point of view of the tracing system, these combinations of artifacts are also modelled as single artifacts.
- **Aggregations**. If the multiagent system supports aggregations of agents (or agents and artifacts), such as organizational units [2], then such aggregations are modeled by the tracing system as a *single* tracing entities, in the sense that trace events can be generated from or delivered to these entities as tracing individuals.

From the point of view of the model, the multiagent platform can be seen as a set of agents and artifacts. Therefore, elements of the multiagent platform are also susceptible of generating and receiving trace events as any other element in the multiagent system.

3.3 Tracing roles

Any tracing entity in the multiagent system is able to play two different roles related to the tracing process (or *tracing roles*): *event source* (*ES*) and *event receiver* (*ER*). and *Trace Manager* (*TM*). ES entities are those which generate trace events as they execute, while ER entities are those which receive these events. TM entities are in charge of coordinating the entire tracing process. The relation between ES and ER entities is many to many: it is possible for events generated by an ES entity to be received by many ER entities, as well as it is also possible for an ER entity to receive events from multiple ES entities simultaneously.

The model requires a third role, which has to be played by at least one entity in the system: the *trace manager* (*TM*). The TM is responsible for registering tracing entities and trace event types. Also, this role is responsible for controling the tracing process so that trace events generated by ES entities are received only by those ER which requested them (selective tracing), also considering the security model, both further explained below, in Sections 3.4 and 3.5.

These three tracing roles are not exclusive and any tracing entity can play one or more of them, at the same time. Regarding to the time when tracing entities can start and stop playing these roles, there are important differences between agents or agent aggregations and artifacts. On the one hand, agents and aggregations can start or stop playing any of these roles dynamically according to their current state. On the other hand, artifacts, which are passive/reactive entities, have to adopt the corresponding roles at design time. Figure 1 shows all the interactions among ES, ER and TM. In particular, it can be seen how trace events are generated in ES entities before arriving to ER entities, while the TM controls the entire process, interacting with ES and ER entities.

When a tracing entity is playing the ER tracing role, the tracing system provides it with a *stream*, which can be seen as a special mailbox where trace events are stored before the ER processes them. These streams can either be pieces of memory (in on-line tracing) or log files (in off-line tracing). In both cases, the ER entity which owns the stream has to be able to limit its size in order not to overload its resources. The model defines a set of *full policies* in order to let tracing entities decide what to do with incoming trace events once the stream is full: stop delivering events to the stream, overwriting previously delivered events in chronological order or flushing events to a log file.

3.4 Selective event tracing

In order to reduce as much as possible the overhead which tracing information can cause to the multiagent system, the model defines a protocol based in *subscription* to trace events. ER entities must subscribe to those trace events which they are interested in. In the same way, once an ER entity is not interested in receiving events to which it had previously subscribed, the ER entity may unsubscribe from them. As a consequence, only those trace events to which at least one ER has previously subscribed are generated and

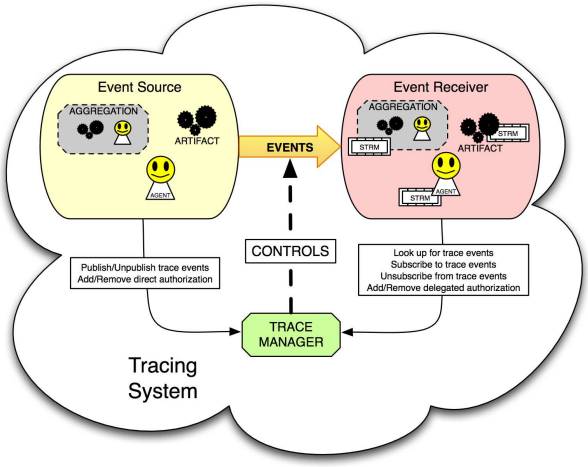


Figure 1: Interaction between the different tracing roles in the tracing system

ER entities do not receive any tracing information they are not interested in.

In order to give support to this subscription mechanism at run time, each ES entity has to publish which tracing information it can provide. In Figure 1 it can be seen how ES entities request the TM to publish and unpublish those trace events they can provide. It can also be appreciated how ER entities are able look for available trace events as well as they can also subscribe and unsubscribe at run time.

3.5 Security

In order to let ES entities decide which ER entities can receive their trace events, when an ES entity publishes its trace events, it has also to specify which roles and entities in the multiagent system are authorized to receive these events. This is defined as *direct authorization*. In this way, when an ER entity wants to receive events of a specific event type which come from a specific ES, it has to be authorized as an entity or it has to be able to assume one of the authorized roles. ER entities which are authorized to receive trace events from certain ES entity can also authorize other roles or ER entities to receive the same trace events. This is defined as *authorization by delegation*. In this way, the TM mantains an authorization graph for each event type which is being offered by each ES. This authorization graph is dynamic, since tracing entities can add and remove authorizations at run time. When an authorization, direct or by delegation is removed, all those delegated authorizations which depended on the removed one are also removed.

The direct authorization mechanism has the advantage of being conceptually simple; however, asking for an authorization each time an ER entity needs to trace an ES entity can cause an important overhead to ES entities, which may receive too many authorization requests. Authorization by delegation can help reducing the overhead this authorization mechanism can cause to some ES entities due to excesive authorization requests while still keeping the security model conceptually simple.

The tracing system does not control which entities can asume each role in order to receive trace events of a spe-

cific event type or to add and remove authorizations. It is the multiagent platform which has to provide the necessary security mechanisms to prevent agents from assuming unappropiated roles.

4. TRACING SYSTEM ARCHITECTURE

This section describes a generic architecture to incorporate a tracing system to a multiagent platform according to the model previously described in Section 3. This architecture was designed to be integrated within the multiagent platform in order to let the tracing system be as efficient and scalable as possible and that has conditioned some of the decissons which have been made when designing it. However, this is a generic, platform independent architecture and so, it has been designed to be implemented in any multiagent platform, in order to give support to any type of trace events, while supporting selective event tracing and incorporating the security issues commented in the previous section.

Tracing entities considered in this architecture are the same as in the TRAMMAS model: agents, artifacts and aggregations. As in the model, these tracing entities can be considered to be playing two different tracing roles. When they are generating trace events, tracing entities are considered *Event Source* entities (*ES*). When they are receiving trace events, tracing entities are considered *Event Receiver* entities (*ER*). According to the model presented in Section 3, agents and aggregations can start and stop playing these tracing roles at run time, while artifacts are static and are designed to play one or both of the tracing roles all the time.

This architecture is based on the concept of *tracing service*, which is used in order to model the concept of event type. Tracing services are special services which are offered by tracing entities, in a similar way to traditional services, to share their trace events. Each tracing entity may offer a set of tracing services, corresponding to the different event types which the tracing entity generates. In the same way as trace events in the model, tracing services can be classified attending to the tracing entity which offers them. Tracing services can also be compound, like trace events in the model, in order to provide more complex tracing information.

When a tracing entity wants to offer any tracing information, it must publish the corresponding tracing service so that other tracing entities can request it if they are interested in its trace events. When a tracing entity does not want to receive certain trace events anymore it only has to cancel the request to the corresponding tracing service. In this way, the architecture gives support to *selective event tracing* by means of the tracing service request mechanism.

As with traditional services, when tracing services are published, it is also published which agent roles or tracing entities are authorized to request the service. In this way, when an tracing entity wants to request a tracing service, it has to be previously authorized or it has to be able to assume an authorized role. Authorizations for a tracing service can be added and removed at run time by the tracing entity which published it by means of updating the corresponding published data on that tracing service. Tracing entities which have assumed a role which is authorized to request a tracing service, can also authorize other roles to request the service. In this way, the architecture provides support to the authorization mechanism described by the model.

The Trace Manager is the main component of the tracing system. Integrated within the multiagent platform, this component plays the TM role (see Section 3.3 for tracing roles) and thus, it is in charge of coordinating the entire tracing process. This component of the tracing system is described in more detail in Section 4.1.

4.1 The Trace Manager

As previously commented, the Trace Manager is the main component of the tracing system. Like other services provided by the multiagent platform, the Trace Manager can be distributed. This module is in charge of coordinating the entire tracing process, which means managing the trace event registration and subscription processes, as well as providing and managing the authorization mechanism. Internally, the Trace Manager incorporates different modules, each one providing different functionalities:

- **Trace Entity Module (TEM):** This is the component of the Trace Manager which registers and manages all the tracing entities.
- **Tracing Services Module (TSM):** This is the component of the Trace Manager which registers and manages all the tracing services offered by ES entities.
- **Subscription Module (SUBM):** This is the component of the Trace Manager which stores and manages subscriptions to each tracing service and ES entity.
- **Authorization Module (AM):** This is the component of the Trace Manager which stores and manages the authorization tree for each tracing service and ES.

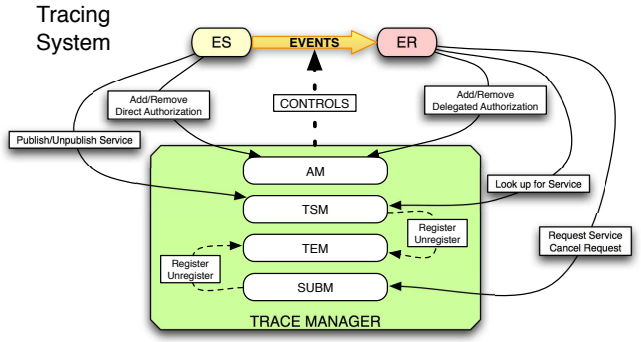


Figure 2: Architecture model of the tracing system and interactions among tracing entities depending on their tracing roles and the Trace Manager's internal modules

Figure 2 shows how tracing entities interact with the Trace Manager depending on the tracing role that they are playing. These interactions are detailed below:

- **Publish/Unpublish Service:** When an ES entity wants to share its trace events it has to publish the corresponding tracing services before any other entity can request that information. Published tracing services are stored in the TSM. When the ES does not want to offer a tracing service anymore, it has to remove the publication. If the tracing service is the first one offered by the ES entity, then this ES is internally

registered in the TEM. In the same way, when an ES entity unpublishes all of its tracing services, it is internally removed from the TEM.

- **Add/Remove Direct Authorization:** ES entities which have published a tracing service can specify which roles have to be assumed by ER entities in order to request that tracing service. ES entities add and remove direct authorizations for each of the tracing services which they provide and the corresponding authorization tree is stored in the AM.

- **Add/Remove Delegated Authorization:** ER entities which have assumed a role which authorizes them to request a tracing service can also authorize other roles to request that tracing service. In the same way, ER entities can remove those delegated authorizations which they previously added. Modifications in the corresponding authorization tree are registered in the AM.

- **Look up for Service:** ER entities can look up in the TSM to know which tracing services are available and which ES entities offer them before requesting any tracing information.

- **Request Service / Cancel Request:** ER entities which want to receive certain trace events from an ES have to request the corresponding tracing service to the Trace Manager. The Trace Manager verifies against the AM that the ER entity has authorization for that tracing service before adding the subscription to the SUBM. When an ER entity does not want to receive events corresponding to a specific tracing service, it has to cancel the request of that service and the corresponding subscription is also deleted in the SUBM. If the ER entity which requests the tracing service was not subscribed to any other tracing service, then this entity is internally registered and listed in the TEM. In the same way, when an ER entity cancels all of its requests, it is internally removed from the TEM. The trace manager only records and delivers those trace events for which there is at least one tracing service request in the SUBM.

5. EXAMPLE

Let us consider an agent-based market like the one described in Figure 3, where agents sell/buy articles to/from other agents. In this market, there are two different agent roles: *seller* and *customer*. Seller agents are those which offer products that different customer agents can buy. The market is open and thus, customer and seller agents enter and abandon the market at run time. Customer and seller agents have to negotiate and reach agreements about the price customers are going to pay for the products and the conditions in which these products will be provided (for instance, time conditions).

Customer agents are independent from each other, as well as seller agents, but they may need to know certain data concerning the negotiation process or final prices reached by other agents in the market, so that they can adapt to changes and needs in the market. For instance, a seller agent may change the price of a product according to recent sales of

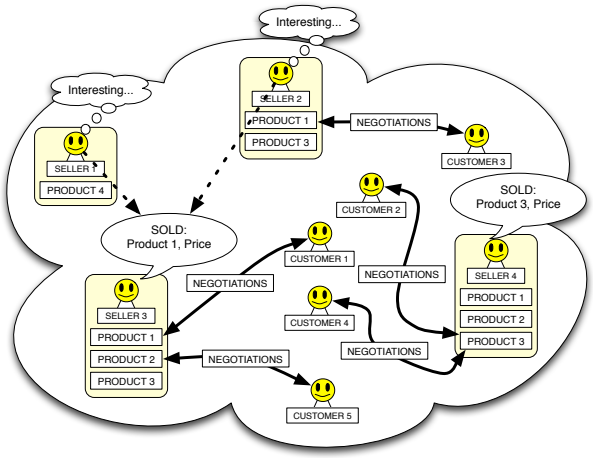


Figure 3: Multiagent system based virtual market where customer and seller agents negotiate before buying/selling products.

that product in order to be more competitive. Also, customers interested in a product may start negotiations with different sellers to obtain their products faster or for a lower price. Knowing what other agents in the market are doing can also be useful for sellers which may be considering offering a new product and want to know how many customers may be interested and which price is being payed for that product in the market.

Since discussions about norms, agent trust and so on are out of the scope of this study, it will be assumed that agents which enter the virtual market agree to share their information as well as they are also benevolent in the sense that they always inform the other agents about their activities and that they do not lie about these activities.

From this point on, the example will consider only the following concrete situation: Seller agents may want to know when certain products are sold, as well as the time and the price at which they have been sold. A seller may use this information in order to adjust the price at which he is selling that product according to the market. Agents which are considering offering a certain product may also be interested in knowing how that product is being sold. For instance, in Figure 3, *Seller 2* is interested in sales of *Product 1*, which he also offers; however, *Seller 4* is not interested in sales of *Product 1*, in spite of the fact that he also offers it. *Seller 1* does not offer *Product 1*, but he is also interested in sales of that product, maybe because he is considering selling it too. For the sake of simplicity, it will also be assumed that both roles (seller and customer) cannot be played at the same time by the same agent.

The rest of the section will explain different strategies to solve the problem of sharing all this sale-related information among the different sellers. Three different solutions have been considered: Two of them based on comon services available in most multiagent platforms (a white pages and a yellow pages services) and finally, a solution based on an event tracing system like the one presented in this paper.

In a solution based on a white pages service (such as the one provided by the AMS in FIPA), each seller would have to ask the agent in charge of providing that service for agents

in the market (this implies a message from the seller to the white pages provider to ask for the existing agents and the corresponding answer from the white pages provider to the seller). After that, the seller would have to actively send a message to all agents in the system each time he sells a product (a typical AMS would not be able to distinguish between sellers and buyers). For a system with a number of agents (sellers or customers) equal to n_{agents} and a number of sales equal to n_{sales}, the number of messages sent to inform about all of the sales would be $n_{messages} = (2 + n_{agents}) * n_{sales}$. This solution would not only cause unnecessary information traffic, since messages are sent to sellers which may not be interested in that information and to customers, but also would cause overhead in these non interested agents (sellers or customers), which would also have to process this extra information.

A yellow pages service (such as the one provided by the DF in FIPA) can also be used in order to reduce the amount of unrequested information which was to be transmited when using a white pages service. First of all, all sellers would have to register a service for each product they sell. Then, each time a sale is done, the seller asks the yellow pages provider agent for the rest of the sellers of that product (as with the white pages solution, this implies a message from the seller to the yellow pages provider to ask for the sellers of that product and the corresponding answer from the white pages provider to the seller). Finally, the seller has to send a message to each of the sellers of that product. For each sale, the total amount of messages would be $(2+n_{providers})$, being $n_{providers}$ the number of sellers which offer that product. The worst case would be when all of the sellers offer the product which has just been sold and the number of messages would be $(2 + n_{sellers})$. The best case would take place when none of the sellers provides that product and the number of messages would only be 2. As a result, for a system with n_{sales}, the number of messages sent would be $2 * n_{sales} \leq n_{messages} \leq (n_{sales} * (2 + n_{sellers}))$. Though this solution reduces the number of unnecessary messages to be transmitted and processed, it still has some withdraws. First, sellers which are interested in a product have to register as providers of that product in order to be informed (in Figure 3, *Seller 1* would have to be registered as a provider for *Product 1* in order to be informed about sales of that product). And second, sellers which provide a product are always informed about sales of that product eventhough they may not be interested (in Figure 3, *Seller 4* provides *Product 1*, but he has no interest in receiving any information about that product).

To solve this problem using an event tracing system like the one presented in this paper, seller agents have to publish data relating to their sales as tracing services. So, for each product which is provided, sellers publish a tracing service. Agents interested in sales of a product request the corresponding tracing service and, from that moment, they receive a trace event each time a product of the specifyed type is sold. In this case, no messages are sent, but trace events. For each sale, the total amount of trace events transmitted (n_{t_events}) would be the number of sellers which are interested in that product and requested the corresponding tracing service. In a system with n_{sales} sales, the number of trace events transmitted would be $0 \leq n_{t\_events} \leq (n_{sales} * (n_{sellers}))$. When there is not any seller interested in a product, no trace events are generated and so, the amount of information transmitted is reduced to that which is strictly necessary. Also, since sellers do not have to know which other sellers are interested in their sales, their internal logic remains simple, unlike in previously shown solutions.

Table 1: Summary of best and worst case costs as a function of the number of sales (n_{sales}) for the different techniques: White pages, yellow pages and event tracing.

Number of transmissions		
	Best case	*Worst case*
White P.	$n_{sales} * (2 + n_{agents})$	$n_{sales} * (2 + n_{agents})$
Yellow P.	$2 * n_{sales}$	$n_{sales} * (2 + n_{sellers})$
E. Tracing	0	$n_{sales} * n_{sellers}$

Table 1 shows the number of transmissions (either messages or trace events) as a function of the number of sales in the virtual market. The number of transmissions in the worst case is in the same order for all techniques. However, the best case is constant for event tracing while it is higher using the other techniques.

6. CONCLUSIONS AND FUTURE WORK

This paper presents TRAMMAS, an abstract model of an event tracing system for multiagent systems. Unlike traditional tracing systems, the presented model is not conceived only as a helping tool for multiagent system developers or administrators, but it is also conceived as communication mechanism which lets agents and other entities in the system generate trace events, as well as receiving events generated by other entities. In this way, this tracing information could be used by entities in a multiagent system to perceive and interact with their environment.

With the trace model, a generic architecture has also been presented. This architecture model lets concepts and mechanisms described by the model be incorporated to a multiagent system at platform level, which is more efficient and flexible than incorporating them at application level.

Finally, a virtual market example, where agents need to know certain information about other agents' activities, has been presented. In this example, different techniques and strategies have been used to transmit the necessary information. The analysis performed for each of these techniques shows that event tracing can help reducing the amount of unnecessary information which has to be transmitted and processed, while keeping agents' internal logic as simple as possible and thus, this contributes to the scalability and feasibility of multiagent systems.

7. ACKNOWLEDGMENTS

This work is partially supported by projects PROMETEO/2008/051, CSD2007-022, TIN2008-04446 and TIN2009-13839-C03-01.

8. REFERENCES

[1] J. Alberola, L. Mulet, J. Such, A. García-Fornes, A. Espinosa, and V. Botti. Operating system aware multiagent platform design. *Fifth European Workshop On Multi-Agent Systems (EUMAS 2007)*, pages 658–667, 2007.

[2] E. Argente, V. Julian, and V. Botti. Mas modeling based on organizations. *Agent-Oriented Software*

Engineering IX: 9th International Workshop, AOSE 2008 Estoril, Portugal, May 12-13, 2008 Revised Selected Papers, pages 16–30, Jan 2009.

[3] F. Bellifemine, F. Bergenti, G. Caire, and A. Poggi. Jade-a java agent development framework. *Multiagent Systems, Artificial Societies, and Simulated Organizations*, Jan 2005.

[4] R. Bordini, M. Dastani, and M. Winikoff. Current issues in multi-agent systems development (invited paper). *Post-proceedings of the Seventh Annual International Workshop on Engineering Societies in the Agents World*, pages 38–61, Jan 2007.

[5] R. Bordini and J. Hübner. *Jason: A Java-based interpreter for an extended version of AgentSpeak*, Feb 2007.

[6] R. Bordini, J. Hubner, and R. Vieira. Jason and the golden fleece of agent-oriented programming. *Multiagent Systems Artificial Societies and Simulated Organizations International Book Series*, Jan 2005.

[7] T. Bosse, C. M. Jonker, L. v. d. Meij, A. Sharpanskykh, and J. Treur. Specification and verification of dynamics in cognitive agent models. *Proceedings of the Sixth International Conference on Intelligent Agent Technology, IAT'06. IEEE Computer*, pages 247–264, Jan 2006.

[8] T. Bosse, D. Lam, and K. Barber. Tools for analyzing intelligent agent systems. *Web Intelligence and Agent Systems*, 6(4):355–371, Jan 2008.

[9] J. Botia, J. Hernansaez, and A. Gomez-Skarmeta. On the application of clustering techniques to support debugging large-scale multi-agent systems. *Programming multi-agent systems*, 4411/2007:217–227, Aug 2007.

[10] J. Botia, J. Hernansaez, and F. Skarmeta. *Towards an approach for debugging MAS through the analysis of ACL messages*, volume 3187/2004, pages 301–312. Springer Berlin / Heidelberg, Jan 2004.

[11] L. Búrdalo, A. Terrasa, and A. García-Fornes. Towards providing social knowledge by event tracing in multiagent systems. In *HAIS '09: Proceedings of the 4th International Conference on Hybrid Artificial Intelligence Systems*, volume 5572/2009 of *Lecture Notes in Computer Science*, pages 484–491, Berlin, Heidelberg, Jan 2009. Springer-Verlag.

[12] J. Collis, D. Ndumu, H. Nwana, and L. Lee. The zeus agent building tool-kit. *BT Technology Journal*, 16(3):60–68, Jul 1998.

[13] F. Dignum and G. Vreeswijk. *Advances in Agent Communication*, volume 2922 of *Lecture Notes in Computer Science*, chapter Towards a Testbed for Multi-party Dialogues, pages 212–230. Springer Berlin / Heidelberg, 2004.

[14] V. Dignum, F. Dignum, and L. Sonenberg. Towards dynamic reorganization of agent societies. In *In Proceedings of Workshop on Coordination in Emergent Agent Societies*, pages 22–27, Jan 2004.

[15] IEEE. *1003.1, 2004 EDITION IEEE Standard for Information Technology Portable Operating System Interface (POSIX)*. 2004.

[16] D. Lam and K. Barber. Debugging agent behavior in an implemented agent system. *Second International Workshop in Programming Multi-Agent Systems (ProMAS)*, Jul 2004.

[17] D. Lam and K. Barber. Comprehending agent software. In *AAMAS '05: Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*, pages 586–593, New York, NY, USA, Jan 2005. ACM.

[18] D. C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, Jan 2001.

[19] V. Mafik and M. Pechoucek. *Social knowledge in multi-agent systems*, volume 2086/2001 of *Lecture Notes in Computer Science*, pages 211–245. Springer Berlin / Heidelberg, Jan 2004.

[20] D. Ndumu, H. Nwana, L. Lee, and J. Collis. Visualising and debugging distributed multi-agent systems. In *AGENTS '99: Proceedings of the third annual conference on Autonomous Agents*, pages 326–333, New York, NY, USA, Jan 1999. ACM.

[21] A. Omicini, A. Ricci, and M. Viroli. Artifacts in the a&a meta-model for multi-agent systems. *Autonomous Agents and Multi-Agent Systems*, Jan 2008.

[22] L. Padgham, M. Winikoff, and D. Poutakidis. Adding debugging support to the prometheus methodology. *Engineering Applications of Artificial Intelligence*, 18(2):173–190, Jan 2005.

[23] E. Platon, S. Honiden, and N. Sabouret. Oversensing with a softbody in the environment: Another dimension of observation. In *Proceedings of Modeling Others from Observation at International Joint Conference on Artificial Intelligence*, 2005.

[24] A. Pokahr and L. Braubach. *Jadex Tool Guide*, Sep 2008.

[25] V. Sanchez-Anguix, A. Espinosa, L. Hernandez, and A. Garcia-Fornes. Mamsy: A management tool for multi-agent systems. *7th International Conference on Practical Applications of Agents and Multi-Agent Systems (PAAMS 2009)*, pages 130–139, 2009.

[26] A. O. Software. *JACK Intelligent Agents Tracing and Logging Manual*, May 2008.

[27] P. Tichý and P. Slechta. *Java Sniffer 2.7 User Manual*, 2006.

[28] G. Valetto, G. Kaiser, and G. Kc. A mobile agent approach to process-based dynamic adaptation of complex software systems. In *EWSPT '01: Proceedings of the 8th European Workshop on Software Process Technology*, pages 102–116, London, UK, 2001. Springer-Verlag.

[29] D. Weyns, A. Omicini, and J. Odell. Environment as a first class abstraction in multiagent systems. *Autonomous Agents and Multi-Agent Systems*, Jan 2007.

Drag-and-Drop Migration: An Example of Mapping User Actions to Agent Infrastructures

Silvan Kaiser
DAI-Labor, Technische Universität Berlin
Ernst-Reuter-Platz 7
10587 Berlin, Germany
silvan.kaiser@dai-labor.de

Michael Burkhardt
DAI-Labor, Technische Universität Berlin
Ernst-Reuter-Platz 7
10587 Berlin, Germany
michael.burkhardt@dai-labor.de

Jakob Tonn
DAI-Labor, Technische Universität Berlin
Ernst-Reuter-Platz 7
10587 Berlin, Germany
jakob.tonn@dai-labor.de

ABSTRACT

Runtime management of a distributed Multi-Agent System is a complex task. Tools that offer a generic solution for this problem and that are intuitive to use only exist in a very limited way so far. In this paper, we take the example of a Drag-And-Drop migration feature to show a concept and a prototype implementation for an intuitive to use user action. This triggers an interaction between the Multi-Agent System and the visual management application, resulting in a migration of a mobile agent between different Agent Nodes. The example shows how the software agent metaphor, used in AOSE concepts at design time, can be sustained and elaborated in runtime tools for Multi-Agent Systems.

1. INTRODUCTION

Distributed Multi-Agent Systems (MAS) are, in general, rather complex systems used to solve complex tasks. Physical distribution as well as large numbers of agents become serious issues when monitoring running MAS. Classical monitoring tools, applying user interface elements like log file outputs and tables of entities, are unintuitive and become confusing with rising numbers of elements. The software agent approach provides concepts for dealing with distributed and highly dynamic infrastructures, in which entities are added and removed in large numbers and at different locations. An user interface (UI) has to keep up with these aspects, giving the user overview and manipulation abilities to assert control over the MAS at runtime. Text based interaction, for displaying system information or issuing commands, is less than optimal. Simple user interactions can provide easy access to complex functionalities and the dynamics of MAS allow agents to react to infrastructure changes as conducted by an administrator.

We propose to provide insight into a running MAS infrastructure, by providing an approach of intuitive interaction for MAS administrators relying on graphics, common metaphors and real-world knowledge of users. Major elements of this approach are user interactions that allow an user to change his perspective or manipulate entities in the running MAS, e.g. stopping a specific agent. In this paper we focus on agent migration by Drag-and-Drop as an example of a concept, design and implementation of such an user interaction. Agent migration is well known concept that can be used to achieve goals like load balancing or administrative tasks. This example provides an excellent opportunity to show the connections between a simple metaphor known to any user, the concept of how this metaphor is integrated into the UI and the implementation details of the resulting software components. The following user interaction approach example relies on the Advanced Structured Graphical Agent Realm Display (ASGARD) [11] which provides the foundation for the MAS visualization. The example implementation uses the JIAC V framework and is implemented in Java. In JIAC V agents can be mobile to support several types of migration (weak/strong) and can be cloned in order to create similar redundant agents. In ASGARD agents are represented as boardgame play figures that can be dragged and dropped from node to node, similar to a real life boardgame.

The details of this concept and implementation are described in the following sections and are structured as follows: The next section provides an overview of related work in MAS monitoring. This allows a more accurate placement of our approach in comparison with given concepts. Related work is followed by a general concept description providing details over the different aspects of this approach. Subsequently the prototype implementation describes how the concept was realized and finally a conclusion section discusses evaluation, results and future work.

2. RELATED WORK

2.1 Shell-based Monitoring and Interaction

The classical and most frequently used approach to interact with running applications by developers is the use of command shells and their input and output functionality. These command shell applications are usually provided by the operating system. Their main advantage is that command shell output (and by a lesser margin input as well) is easily implemented in any application. This is contrasted by the fact that command shell interaction is not intuitive. There is no way for the user to judge the importance of a textual log output on first look, and keyboard commands (if available) tend to be of a rather cryptic nature. Furthermore, as waiting for keyboard input might block an application or is rather complex to implement in a non-blocking way, most developers only offer the process of stopping an application, reconfiguring it and restarting as a mean of interaction.

Those disadvantages of shell-based interaction get even more significant in a distributed MAS, as the usual approaches either require the developer to watch a dedicated

shell console for each processing entity in the MAS or try to differentiate the log outputs of each entity in a single console. Providing live interaction for the MAS is almost impossible by those means.

2.2 Tools for Post Mortem Analysis

Several monitoring tools that use a more visual approach are available for various MAS implementations. One of these is the **ADAM3D** [4] tool that visualizes interaction between agents on the base of 3D technology, using the third dimension as a temporal axis. The **Brahms Agent Framework** [9] provides it's own visualization tool as well, which visualizes property changes and communication between agents over time. Both of these tools rely on log output that is collected in a database, so they can only be used for post mortem analysis of a MAS and thus cannot provide any sufficient live management functionality.

2.3 Live Management Tools

There are few generic solutions for the problem of providing live management for a MAS so far. The **JIAC Node Monitor** [6, p. 126] for the JIAC Agent Framework [3] is a first attempt at solving this problem and can be seen as the predecessor to the ASGARD concept, but is limited to a single *Agent Node*[1] and thus does not represent the distributed nature of a MAS. It's main disadvantage is the limited scalability of it's visualizations, and it's management functions were not intuitively designed. However the Node Monitor shows that a visual monitoring and management application does improve the workflow during the implementation process of a MAS application. A developer can use the existing monitoring solution to check his entities instead of having to implement his own interface for monitoring and management.

3. CONCEPT

Herein metaphors for the user interaction, the migration concept and the management aspects for our approach are described.

3.1 Metaphoric understandability

A key requirement in the creation of an easy-to-use management solution is that it provides an intuitive interface, so that the user will understand visualizations and interactions because of prior experiences and knowledge. To achieve this effect, a common attempt is the use of metaphors [1]. Metaphors provide a mental bridge between the abstract nature of the software "world" and concrete objects in the real world. As this link is essential for understandability, a good metaphor should always have a lot in common with the represented software entity.

In the JIAC V[3] framework, there is a set of basic entities that provide the infrastructure (see section 4). To create a visualization of these structures in ASGARD, the most important JIAC V entities are replaced with metaphors according to their role. *Agents*, as the entity that provides all application-specific processes in the MAS, can easily be visualized by an abstract human figure such as a play figure from common board games. *Agent Nodes* as the providers of a runtime environment for agents are consequently represented by rectangular platforms, as a "floorboard" for agents. Agents are placed on these platforms to make the hierarchic relationship between agents and Agent Nodes instantly visible. *Communication* is a straightforward choice as well. The image of a letter envelope is one of the most common metaphors to indicate a message. To add even more understandability and the possibility to trace the communicating entities, a path between those entities is shown.

States of entities are a bit more diverse. Some states like the life cycle of an entity can best be visualized by coloring parts of the corresponding metaphor, such as an agent head being colored green or red to indicate a running or stopped state. This choice of colors is straightforward as it relates to the common experience with traffic lights. Other states and capabilities are better visualized by adding icons to an entity, such as the suitcase icon in figure 1 indicating migrateable agents and nodes that support migration.[2]

3.2 Migration

The migration of agents is a process which allows an agent to move from a source node to a target node by sending a mobile agent description as message. This process extends the agent's life cycle. Thus a new class of agents are established on JIAC V platforms – mobile agents.

Mobile Agents extend common agents on JIAC V. A mobile agent is aware of its own configuration, library dependencies and feature requirements.

A mobile agent has the capability to invoke his own migration to an other Agent Node, but is not designed to avoid his migration. A mobile agent creates an abstract description of itself, so called *Agent Image*.

This description is a message that can be transmitted. The agent image differs from a simple *Agent Description* which contains agent name, unique identifier and agent address for communication.

In JIAC V we distinguish between different modes of migration. In the first step the process always creates an agent image. The migration modes vary from each other in how thorough the agent image reflects the migrating agent. The agent image has to contain enough data to perform the specified mode of migration, e.g. for starting with a clean state, or one that maintains the complete state the agent was in right before the migration was triggered.

In the process of migration the *Agent Node*'s task is well-defined: It has to handle the migration on the agent node level. The agent nodes supporting mobility are grouped by using a reserved message bus. This bus is supposed to find other nodes with mobility support and exchange their addresses. It is furthermore an abstract facility. Intermediate agent nodes on the route of a message (e.g. nodes acting as a network gateway) are hidden.

While migrating an agent, the two involved agent nodes communicate directly to move the agent from one to the other. The individual addresses of the partner node is known from the message bus.

The migration process (see figure 5) starts with the send-request of the mobile agent to the source Agent Node. The request consists of the complete requirements of the mobile agent. Alternatively an upper authority can request the migration for a distinct agent, e.g. a node management component.

The target node is able to accept or reject this request. In

[1] see section 3.1 or [3]

[2] A more detailed discussion of JIAC entities and their metaphors in ASGARD can be found in [11].

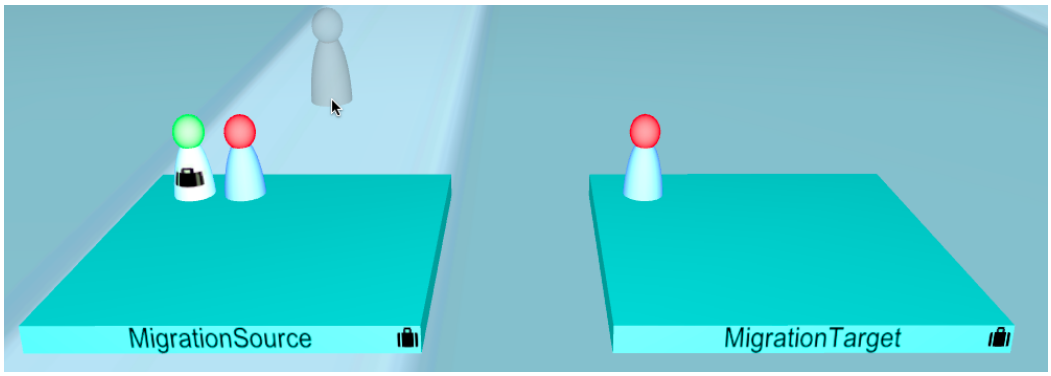


Figure 1: The user drags the migratable agent

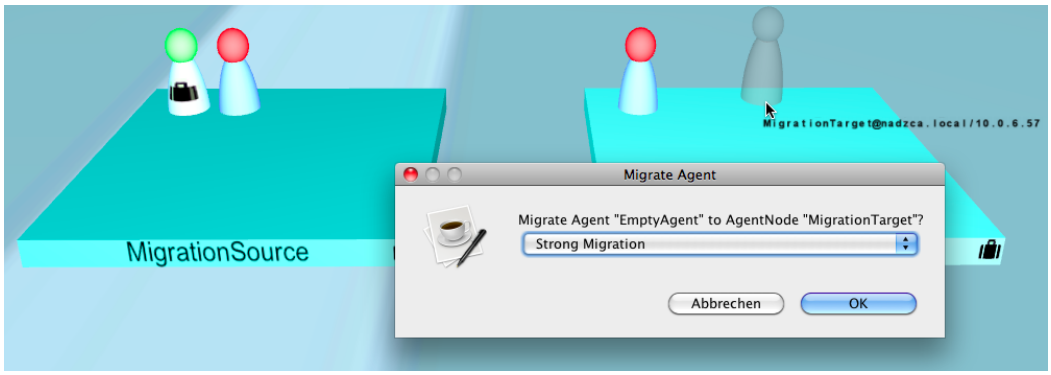


Figure 2: The user drops the migratable agent and triggers the migration

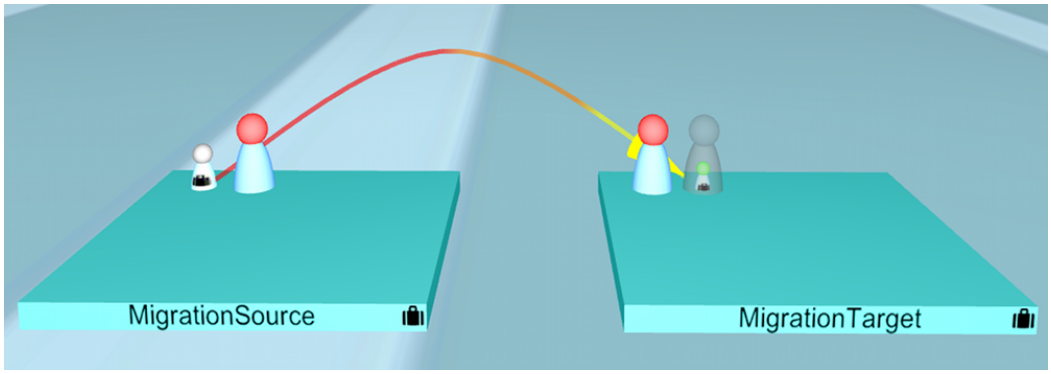


Figure 3: The migration process is animated

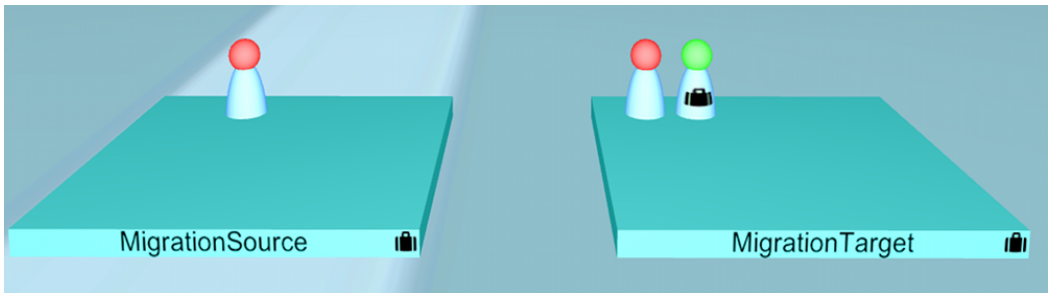


Figure 4: The migrated agent now runs on the target node.

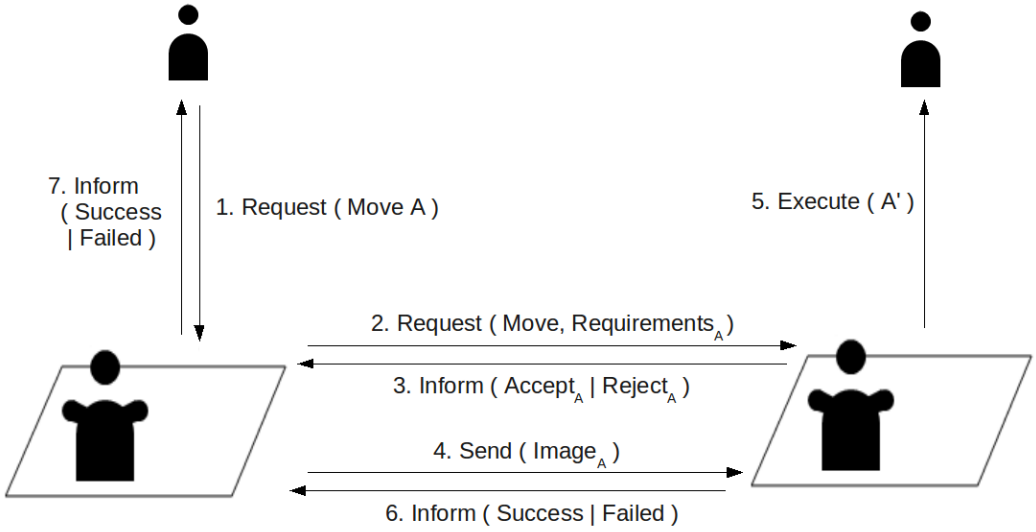


Figure 5: Interactions between Agent Nodes and agents for migration process

case of an acceptance the agent can be transported to the target node. In doing this, the complete agent image will be send from source to target node. After completing transmission the agent will be reconstructed from the transmitted image. The source node will be informed about success or failure of the transportation process. Finally the agent receives feedback about the whole migration process. In case of migration without cloning the original agent is shut down, which completes the migration process, whereas in case of cloning, the process is already completed upon receiving the success or failure message. During the whole process, notifications are send over the management interface that contain information about the current state of the migration process. This enables a monitoring tool like ASGARD to keep track of the migration process.

The visualization of this migration process makes use of the metaphors mentioned in section 3.1. The existing metaphors for agents and Agent Nodes are used as a base to visualize this process. To show the initiation of a migration process from one node to another, a translucent copy of an agent shape, a "ghost agent", is placed on the target node first. The migrating agent's representation on the source node is then animated to shrink, whereas the ghost agent is filled by a regular agent object growing inside of it, as visible in figure 3. As this visualization alone does not emphasize the communication aspect of the migration process, a path between the positions of the agents on source and target Agent Node is shown as well. This path indicates the direction of the migration and makes it easy to spot the corresponding partners in situations with a lot of migrations occurring.

3.3 Management using Drag-and-Drop

Distributed MAS are usually long lived infrastructures providing services for users or other software systems. Demonstration, maintenance or other reasons sometimes demand the ability to migrate agents from one node to another in a running platform. However, short of writing a quick program to trigger such a migration, there generally are no direct manipulation options for a MAS administrator that could accomplish such a trivial task (trivial once the rather complex process of migrating a mobile agent is already implemented). Thus a simple user action is required that allows a user to trigger migrations at will and the well known Drag-and-Drop metaphor is an obvious choice.

The Drag-and-Drop process has evolved to a standard feature in all modern user interfaces. It is based on the metaphor of moving an object from a source location to a target location, and thus used for all kinds of processes that change the physical or logical location of an entity, such as moving files between folders. The usual implementation is that the user uses the mouse to select an object and move it to the target while holding the mouse button down (Drag) and releasing the button once the target is reached (Drop). The common acceptance of Drag-and-Drop in user interfaces makes it an optimal metaphor to manage migration processes as well, as a migration process in a MAS is characterized by being a location change of an agent.

To make use of this, ASGARD offers control over migration by dragging an agent from the source node onto a target node using the mouse. Migrateable agents are easy to identify for the user by the suitcase icon on agents, showing the migration ability, Agent Nodes with support for migration are similarly marked. To indicate that an agent is being dragged, a ghost agent is attached to the mouse cursor and moved along with it (see figure 1). Once the agent has been dropped onto a target Agent Node that supports migration, a standard GUI dialog (figure 2) asks the user which kind of process (Strong/Weak Migration or Cloning) is required. ASGARD then initializes the migration process and visualizes it's progress in the same way as an internally triggered migration would be visualized (figure 3).

The separation of the interaction and visualization has the advantage of offering a base for automatically visualizing errors. If an error occurs during the initiated migration process, the migration animation will not be shown and the user will see that the migration process was not executed as anticipated. This concept can even be extended to offer

detailed information about the occurred error, by reading out the involved entities' properties and presenting them in a textual or graphical way to the ASGARD user.

3.4 Other interaction metaphors

In it's current state, ASGARD offers several other concepts of interactions using metaphors. One of them is the idea of showing a greater level of detail when using the zoom feature on a selected entity. This matches the users knowledge. He can see far more details of an object if he watches it from a close point of view than from far away. The adaptive level of detail has the advantage of solving a scaling problem as well, as a visualization of every detail in every entity would be way to complex and space-consuming to still be intuitive and understandable.

Other planned features is the use of Drag-and-Drop between ASGARD and a visual editor for agent and Agent Node configurations (AWE [7]) to deploy new agents in the MAS, and a similar way to remove agents and Agent Nodes from the running system. The latter could be done by dragging an entity onto a waste basket, as this metaphor for removing an object is common in current operating systems as well.

4. IMPLEMENTATION

In order to implement the concept of Drag-and-Drop migration the Java based Intelligent Agent Framework version V (JIAC V) was used. JIAC V combines agent technology with a service-oriented approach and provides a wide range of basic functionalities for agent deployment, communication, management and dynamically changing distributed environments. JIAC V MAS consist of a hierarchical structure as shown in figure 6.

A Platform consists of a range of different Agent Nodes that provide the environment for the execution of agents. An Agent Node is roughly equivalent to a Java VM with an agent infrastructure. Individual Agents are implemented as Java objects in an Agent Node and consist of several core components and optional Agent Beans. These beans are used to provide concrete functionalities as plug-in components for agents. The JIAC V foundation is used in a wide range of projects, ranging from service delivery platforms to simulation environments. The following sections elaborate on three specific components of the JIAC V framework relevant for Drag-and-Drop migration.

4.1 Migration Implementation

JIAC supports basic migration capabilities by adding mobility support to the Agent Nodes and instantiating a new mobile agent. JIACs Agent Migration API provides strong migration. The MAS developer only has to use the Java annotation `@Migratable` for migratable fields. Only the stateful information in the Agent Beans has to be marked. There are three ways for the developer to mark the migratable agent properties in an Agent Bean, in order to provide a solution that is usable in all combinations of field access levels. The Developer does not have to stick to a fixed naming scheme for getter and setter methods.

In order to realize strong migration an agents state has to be archived. We do not transfer the whole execution stack of an agent out of a Java VM. The JIAC V API collects all stateful information of all agent properties. The agent engineer marks all stateful information about the agent beans belonging to the mobile agent.

As described in section 3.2 an agent creates its own image. The agent is aware of its configuration and corresponding implementation, e.g. Java classes. Creating a complete image of an agent for strong migration is a complex and expensive process. Due to this we adopted Java Annotations as a simple way for realizing migration support. Each Java field, constituting a migratable state, is marked with the annotation, which is only being evaluated in the case of strong migration. For this purpose, every Java class implementing an agent feature requires a scanning for annotations. This process will be done if an agent image is needed within a strong migration.

A mobile agent is equipped with a scanner, validator and collector for agent properties. For every Java class related to an agent feature the scanner collects Java fields annotated with `@Migratable`. The output of the scanner is a list of Java fields comprising field names and field types. The scanner furthermore determines potential setter and getter methods. Subsequently, each agent property requires validation.

The validator uses the list of collected Java fields. Every field will be approved to be accessible and serializable. A field is accessible, if it is public writeable or it has public *set* and *get* methods. Java fields which do not pass the validation will be dropped, while we call the remaining ones *Agent properties*. These are accessible and serializable properties which can be read and set for the agent.

Finally a collector iterates over the validated agent properties and collects their current values. The collector creates a list of agent properties, comprising their names and values. The set of all collected agent properties constitutes the reconstructible state of the agent.

4.2 Management Interface JMX

The binding element between the migration implementation in the JIAC V Framework described above and the ASGARD visualization implementation described below is the JIAC V management interface. Since JIAC V is implemented in Java, the application of the standard management interface concept JMX [10] was an obvious step. This adds the ability to access the JIAC V infrastructure and analyze an agent Nodes internal data. It can be used by standard tools like JConsole, as well as by proprietary implementations like the ASGARD tool. Standard infrastructure elements in JIAC V, like Agent Nodes, agents and their Beans, provide JMX based access methods that allow remote access to their internal state, data and some functionalities. Agent Nodes are located in the LAN subnet by multicast messages that use a special management channel on the message bus. Last but not least remote JMX clients can register for specific notifications in order to prevent inefficient polling mechanisms.

JMX relies on managed beans (MBeans) that effectively are Java interfaces. By implementing such an interface a class gains the ability to register with a local management component (also called "agent" in JMX terminology) and provides remote access to the attributes and methods (termed "services" in JMX) and notifications through the JMX "agent" component. JMX based client classes can then access the specified interface remotely, usually through the RMI protocol. The JIAC V framework provides a range of standard JMX clients for specific infrastructure elements like Agent Nodes, agents, etc., that ASGARD utilizes as described in

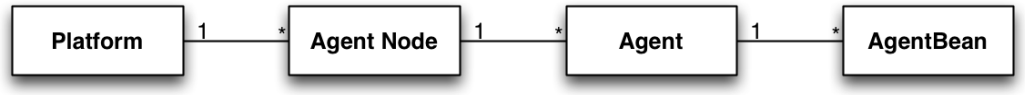


Figure 6: Basic structure of entities in the JIAC V framework.

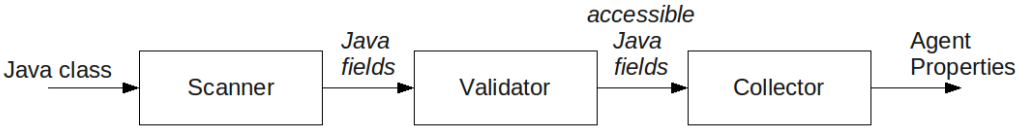


Figure 7: steps to scan Java classes for migratable agent properties

the following section.

4.3 ASGARD

The ASGARD application provides the user interface to the management functions of a JIAC MAS. Using the JMX-based management interface to retrieve information about the current state of the MAS, ASGARD generates a metaphoric visualization at runtime, which is used to provide information to the user as well as a base for user interaction with the system. To create the visualization, ASGARD has to locate the JIAC entities in the network infrastructure. This feature is currently implemented by receiving multicast discovery packets and reading out Java RMI registries. The information received this way consists of JMX service URLs, which are used to connect the management clients to the MAS entities.

As two-dimensional graphic engines and raster images have the problem of not being scalable quite well, ASGARD uses 3D graphics to visualize the structure of JIAC V applications. For this purpose, a 3D engine called *Java Monkey Engine (JME)*[5] is used. It offers bindings to the OpenGL API for most common operating systems, so that ASGARD can be used on a large variety of systems. Furthermore, JME provides a lot of functionality for creating and managing three-dimensional objects and for user interaction. These features turn JME into a solid base for a graphical and interactive application.

ASGARD uses the data from polling the management clients and receiving JMX notifications to generate a tree structure of object instances containing the metaphoric representation of a connected JIAC MAS. The reason why both polling and notifications are used is that the JIAC V management interface provides notifications only for events and property changes that are quite frequent to happen. This provides instant knowledge of their occurrence. Properties that are changed rarely or never can be retrieved by polling with a low frequency. The combination of those techniques offers a good tradeoff between network load and up-to-date information about the managed system's state.

The visual representation objects are built up in an Model-View-Controller (MVC) [8] pattern style to provide easy adaption and a separation of data, visual implementation and interactions. This separation is performed by creating three different objects for each entity. The *entity controller* manages creation and deletion handling for the object and polls the management interface for data. Furthermore, it registers as receiver for both JMX notifications and user interaction events such as mouse clicks. The data retrieved over the management interface is stored in the *data model*. It acts as a buffer for value properties of an entity and provides the base to generate a visualization as well as for a textual presentation of the data in a property table. The *visualization* is generated and managed in a third object that represents the "view" part of the MVC pattern. This object uses the data from the data model to generate and adapt the visualization to the current state of the entity.

As ASGARD uses a three-dimensional space for the visualization, finding the entity that the user is currently pointing at with the mouse cursor has to be done by an algorithm called *ray intersection*. This is done by sending a virtual ray from the camera position in the direction of the mouse cursor into the scene and checking which object's bounding box[3] is intersected by it. ASGARD then forwards mouse over and mouse click events to the foremost entity that got intersected, so that mouse events get handled by the corresponding entity controller class.

The visualization of a migration process is based on the JMX notification mechanism. The Agent Node controller instances in ASGARD register as notification receivers. The migration implementation sends notifications containing the current state of the migration process and identifiers for the involved Agent Nodes and agents, Upon receiving those migration notifications, the representations for the involved entities are identified in ASGARD's internal entity tree. The visualization handler instances then create the visuals and animations which are shown in figure 3.

By extending the user action event handler mechanism to special Drag-and-Drop events, ASGARD is able to provide the functionality to offer migration per Drag-and-Drop to the user. The drag animation is implemented pretty straightforward by moving a "ghost" copy of the entity according to the mouse cursor motion (see figure 1). If the user releases the mouse button, the entity that was below the dragged entity gets identified by ray intersection and receives a drop event that contains a reference to the dragged object. Thus, the drop handler can provide different handling for different entities. In case of the dragged entity being a migrateable agent and the agent Node supporting

[3]The ray/box intersection and the use of bounding volumes for collision checks are common algorithms in 3D computer graphics. Intersection checks are usually performed by solving linear equations to find the intersection point or plane. Further details can be found in [2].

migration[4], the Agent Node controller will trigger the migration process (see figure 3) by calling the appropriate method of the JMX management system. The migration process is then executed in the same way as if an internal event in the MAS had triggered it, so that the same visualization algorithm that shows all occurring migration processes in the MAS will provide a visualization without further adaptions.

5. EVALUATION AND CONCLUSION

5.1 Evaluation

The implementation of a visualization for JIAC V's migration feature and the subsequent adding of the Drag-and-Drop migration management has shown to be successful during an initial institution-wide testing process. The test users were able to instantly connect the visualized process with migration. The migration visualization shows clearly when a migration process is occurring in a MAS application, the current state the migration process is in, and which entities are involved. Using ASGARD during debugging processes has helped to identify migration-related problems such as failed migrations or agents migrating onto a wrong target node.

The implementation of triggering a migration using Drag-and-Drop has proved successful as well. Instead of having to add functionality to offer a user-triggered migration manually to each JIAC V application, developers can now rely on being able to test migration processes by just dragging agents between nodes in ASGARD's visualization of their implementation. As this feature is available in the JIAC V environment for all migratable agents as part of the management interface without any further work for the user, it saves a lot of implementation effort for testing and demonstration processes.

A standardized test of the whole ASGARD application with a larger and more diverse group of users is still to be conducted once the prototype implementation reaches a stable state.

5.2 Conclusion

In this paper we presented an example for mapping user actions to agent infrastructures, the Drag-and-Drop Migration. We motivated why graphical user interfaces provide added value for administrating MAS and which major technologies are involved in our example. We compared our approach with related work and gave a deeper insight into the concepts of the metaphors, migration and Drag-and-Drop concepts used herein. Afterwards we provided more insight into the implementation details of our example and described the different components involved. We conclude with a results evaluation and provide some hints for future work in this concluding section.

The resulting concept shows how user actions can be used to trigger complex operations in a MAS infrastructure, through the application of graphical user interfaces. This requires carefully selected metaphors and an extensive framework to rely on. The result is a visual interaction method that is convenient for administrators as well as intuitive to understand for other observers, e.g. for demonstration purposes.

[4]Both of these abilities are identified using JMX when the entity data and information is gathered.

5.2.1 Future Work

As the concept and initial implementation of Drag-and-Drop migration in ASGARD has proved to be successful during everyday use, more management functionality should be offered to users in a similar way. The integration of agent deployment and removal by Drag-and-Drop already mentioned in section 3.4 are two of them. Other interaction metaphors will be used for life cycle and property influence of entities and to make visual navigation and locating certain entities easier to handle.

6. REFERENCES

[1] S. Diehl. *Software Visualization - Visualizing the Structure, Behaviour and Evolution of Software.* Springer, 2007. ISBN 978-3-540-46504-1.

[2] M. Gomez. Simple Intersection Tests For Games. Gamasutra online article, www.gamasutra.com, October 1999.

[3] B. Hirsch, T. Konnerth, and A. Heßler. Merging Agents and Services — the JIAC Agent Platform. In R. H. Bordini, M. Dastani, J. Dix, and A. E. F. Seghrouchni, editors, *Multi-Agent Programming: Languages, Tools and Applications*, pages 159–185. Springer US, 2009.

[4] S. Ilarri, J. L. Serrano, E. Mena, and R. Trillo. 3D Monitoring of Distributed Multiagent Systems. In *WEBIST 2007 - International Conference on Web Information Systems and Technologies*, pages 439–442, 2007.

[5] JME Development Team, http://www.jmonkeyengine.com. *Java Monkey Engine*, 2009.

[6] J. Keiser. *MIAS: Management Infrastruktur für agentenbasierte Systeme.* PhD thesis, Technische Universität Berlin, September 2008.

[7] M. Lützenberger, T. Küster, A. Heßler, and B. Hirsch. Unifying JIAC agent development with AWE. In *Proceedings of the Seventh German Conference on Multiagent System Technologies, Hamburg, Germany.* Springer, 2009.

[8] T. Reenskaug. Models-Views-Controllers. Technical report, Xerox-Parc, 12 1979.

[9] M. Sierhuis, W. J. Clancey, and R. van Hoof. Brahms - a multiagent modeling environment for simulating social phenomena. In *First conference of the European Social Simulation Association, Groningen*, 2003.

[10] Sun Microsystems, Inc. *Java Management Extensions (JMX) Specification, version 1.4*, 11 2006.

[11] J. Tonn and S. Kaiser. ASGARD - A Graphical Monitoring Tool for Distributed Agent Infrastructures. In *8th International Conference on Practical Applications of Agents and Multi-Agent Systems, University of Salamanca, Spain*, 2010.

AGRID - Agent Based Grid System

Uygar Gümüş
Institute of Science and Technology
Istanbul Techical University
Maslak, İstanbul, Turkey
gumusuy@itu.edu.tr

Prof. Dr. Nadia Erdoğan
Computer Eng. Department
Electrical-Electronics Faculty
Istanbul Techical University
Maslak, İstanbul, Turkey
nerdogan@itu.edu.tr

ABSTRACT

This paper presents the design and implementation of an agent-based grid system (AGrid) that provides clients with a distributed execution environment for sharing of processing power resources. AGrid combines favorable aspects of two different areas of distributed computing, namely grid computing and agent technology. During the design phase, system stability and robustness has been a primary concern. The framework builds on various types of agents that are defined and implemented to handle different issues of grid computing. Each type of agent acts according to protocols that define the interaction and coordination between agents and describe actions required for the management of the grid. This paper describes in detail the procedures followed for connection and disconnection of clients and workers, task assignment, task execution and result delivery.

Categories and Subject Descriptors

D.1.3 [**Concurrent Programming**]: Distributed programming; I.2.11 [**Distributed Artificial Intelligence**]: Multiagent systems

General Terms

Algorithms, Design

Keywords

Agent systems, mobile agents, grid computing, computational grids, JADE.

1. INTRODUCTION

Distributed computing has become one of the popular research topics in computer science. Especially, very high speed Internet connections and new networking structures enable promising research to be conducted in this field. This paper, presents a new agent-based grid system, which combines two different areas of distributed computing, namely grid computing and agent technology.

Grid computing is a model for wide-area distributed and parallel computing across heterogeneous networks, aiming to reach breakthrough computing power at low cost [3]. Grids are hardware and software infrastructures that enable the sharing, distribution and collective use of heterogeneous resources [8]. These resources may be secondary storage, processing power or output data of any specific input output device. The grid system we present focuses on sharing of processing power resources. Reliability and stability are important specifications of grids systems. Data security and trustworthiness of calculation results are very important. Grid systems need mechanisms to manage the grid infrastructure as to ensure these issues.

Agents are encapsulated and autonomous software and hardware systems, which execute an assigned task by communicating and collaborating with other actors at the same or different physical environments [6]. Main attributes of agent systems are flexibility and autonomy. In traditional agent systems, generally no single agent controls the system. Each agent has limited information about the problem and limited capability to solve the problem. Agents build a virtual organization using their communication capabilities and solve the problem by combining the insufficient capability of each agent through intensive cooperation. In the context of grid computing, mobile agents are usually employed in resource discovery, job scheduling, job deployment, task execution and result collection [7].

This paper presents an agent-based grid system (AGrid) for sharing of processing power resources. In AGrid, the framework builds on various types of agents that are defined and implemented to handle different issues of grid computing. Some agents handle job scheduling and job deployment, while others execute jobs assigned to them and produce results. In the design phase, we determined the actors of the system, specifying their tasks and responsibilities. After associating each actor with an agent type, protocols were developed for each role of agents. These protocols define in detail the interaction and coordination between agents and describe actions required for the management of the grid. Methods of connection, disconnection, task assignment, task executing and result delivery are declared. In addition, an efficient and fast messaging infrastructure is developed for effective agent communication. This paper presents the agent types, their responsibilities in the grid systems and the communication protocols between agents during the life cycle of the grid. Protocols on task assignment, job scheduling and result collection as handled by agents are also described in detail. The grid was developed in a systematical manner and up to the final version, three prior versions were implemented, detecting and making design decisions to fix problems of the prior version each time. Figure 1 depicts the final grid architecture.

AGrid is compatible with the Foundation for Intelligent Physical Agents (FIPA) standards [5]. The use of autonomy and flexibility features of agents in the grid design has resulted in a stable and robust grid structure.

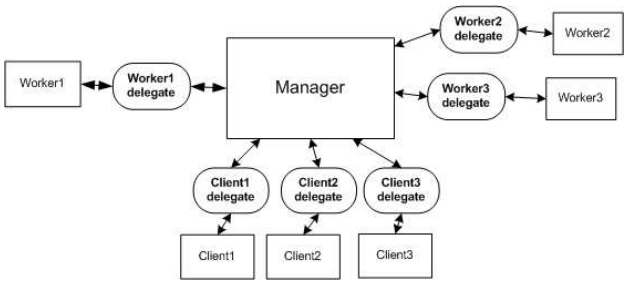


Figure 1: Final grid architecture.

The rest of the paper continues with Section 2, which briefly summarizes recent work in agent based grid computing. Next, Section 3 focuses on design and implementation issues of the system, describing the system components and runtime protocols. In Section 4, system performance is evaluated and, finally, Section 5 concludes the paper.

2. RELATED WORK

Agent based grid system has been a popular area in computer science in recent years. Athanaileas et al. argued about adding mobility support to grid systems using mobile agents [1]. This system, named GridSBAP, is build on OSGA platform and mainly focused on adding mobility feature to the grid systems. GridSBAP uses FIPA ACL standards for agent communication.

AgentScape, an agent supported Internet based grid, is developed by B. J. Overeinder et al.[10] It supports large scale agent system. This system defines its own communication protocols for agents and a resource management system for the grid. AgentScape can adapt to other communication standards using an extra layer which transforms messages in to the native format of AgentScape.

Fukuda and Smith introduced UWAgent, a grid system middleware for Java based on mobile agents [7]. Just like AGrid, UWAgent is not only an agent based grid system but also a middleware that meets management needs of distributed computing. However,UWAgent defines its own communication standards for mobile agents, which is not fully compliant with FIPA standards.

Also Poggi, Michele and Turci, worked on extending JADE framework in order to support grid computing [9].

3. AGENT BASED GRID SYSTEM

AGrid is an agent based infrastructure for distributed and parallel computing. Currently AGrid can be used to create a grid system to share processing power and task executing on remote platforms. The system is implemented on JADE (Java Agent Development Framework) which is developed by Telecom Italia SpA. JADE is a distributed runtime environment on which mobile agents can live, communicate and run parallel tasks via behaviours. JADE also supports graphical user interfaces that can be used for debugging, monitoring, logging and management of the agent system. In addition, JADE is compliant with FIPA specifications, which enables the agents to communicate and cooperate with other agent systems which are also compliant with the FIPA standards.[4].

As stated in the introduction section, multi agent systems and grid systems are two different branches of distributed systems with different perspectives on distributed computing. In this work, a hybrid of these two different approaches is implemented. In this section, we will present the components of the system and describe in detail the protocols between the agents.

3.1 System Participants

In AGrid, four different types of agents cooperate to provide a distributed computing environment:

- `Manager agent` which is in charge of general grid management,
- `Worker agents` which execute jobs assigned to them and produce results,
- `Client agents` which use the grid to run their tasks,
- `Delegate agents` which help the manager agent via coordinating the interaction and the communication between client or worker agents and the manager agent.

Figure 2 depicts the hierarchical relation between agent constituents of the grid.

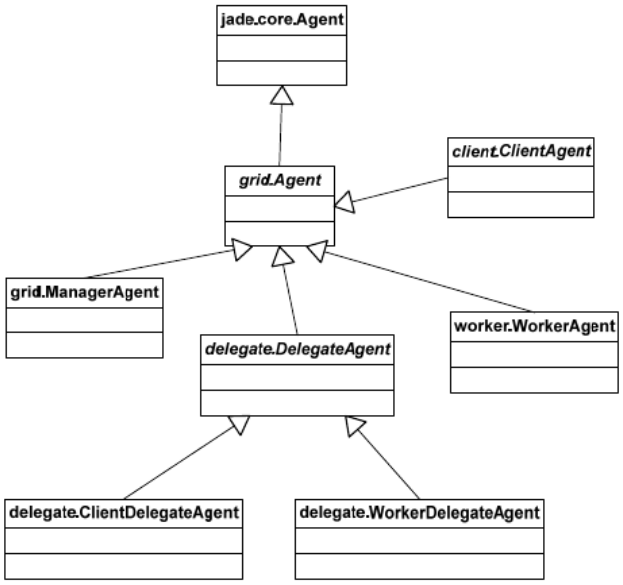


Figure 2: Hierarchical relation between agents.

The following sections give detailed information for each type of agent.

3.1.1 Manager Agent

AGrid has a central management system. There exists a manager agent which controls communication channels, task assignment and result collection issues. Classical agent systems generally do not contain a central management organization [11]. However, a computational grid system usually needs a management structure to control the entire grid [2]. Central management usually becomes a bottleneck; therefore the manager agent in AGrid conveys some of its tasks to delegate agents as to decrease its work load. The manager agent has the following responsibilities:

- Keeps the record of each connected client and worker agent.
- Creates a delegate for each worker and client agent that connects to the system.
- Ensures safely disconnection of participant agents.
- Coordinates reassignment of a task in the case of improper termination/exit of worker agents.
- Terminates task execution in the case of corresponding client agent exits the system.

3.1.2 Client Agents

Client agents are users of AGrid that connect to the grid system to receive service. They connect to the system in order to have their local tasks be executed on remote hosts which can lend their processing power. After a client connects to the grid, it sends the required task information to the delegate agent with which it is associated and waits for the result. It may disconnect while the computation is going on and may later collect the results

3.1.3 Worker Agents

These agents connect to the system to share any of their resources. Currently, AGrid only supports computing power based resources. When a worker agent connects to the system, it provides the manager agent information about its resources that are available and it is willing to share. If the participation request to the grid is accepted, the manager agent creates a delegate agent for the worker, which handles all further communication/ interaction of the worker agent with the grid system.

3.1.4 Delegate Agents

Delegate agents reside on the node where the manager agent is present and handle all communication and protocol implementation between the agents they are representative of and the manager agent. Over the life time of a computation, each agent that participates in the process needs to communicate frequently with grid management. Delegate agents are representatives of those agents and they act as addressees and coordinate the interaction, in order to minimize the heavy work load of the manager agent. As the manager and the delegates communicate locally, communication overhead of the manager is reduced significantly. Two types of delegate agents are created:

Client delegates: One is created for each connected client agent to handle the coordination between grid management and the client.

Worker delegates: One is created for each connected worker agent. Delegate agents plan and organize task assignment, task result collection and handle monitoring issues for their associated worker agent.

3.1.5 Tasks

In AGrid, a task is a set of computations designed to solve a certain problem. Tasks can be highly specialized and may require the target platform where they will be executed to carry certain attributes. During initial system registration, each worker agent provides information about its attributes, in the form of a description of its the computational features, to its delegate. The manager and delegate agent cooperate to select the proper worker agent for the task, according to the attributes requested. Actually, the manager agent consults worker delegates not only to match the requested features of a task with those of worker agents, but also to locate capable worker agents whose schedule is suitable to accept the task. The task runs its abstract "execute" method at the worker platform to which it is deployed. AGrid supports every kind of computational task which can be employed using the Java language. Tasks indicate the attributes they require by an abstract "properties" method. When the manager agent receives a task assignment request through a client delegate agent, it checks the requested attributes and matches the task with a suitable worker agent. After a worker agent is assigned a task/group of tasks, it executes each task via the execute method.

3.2 Protocols

Each agent in AGrid system must execute certain protocols during its lifetime in the grid. These protocols define the behaviour of the agent according to the role it carries in the system. This section introduces these protocols.

3.2.1 Agent Connection Protocols

JADE framework assigns an identifier number (AID – Agent Identifier) to each connected agent. Since the manager agent needs to keep the record of all the agents in the distributed environment, AID is inadequate for AGrid. The manager agent needs to know the type and the properties of each connected agent. Also, the manager agent has to decide whether to allow an agent to connect to the system or not. Therefore, a connection protocol for worker and client agents is defined. There is no need to for a connection protocol for delegate agents, since they are automatically created by the manager agent when any client or worker is connected to system.

There are some minor differences between the connection protocols of client and worker agents. Thus the protocols will be presented separately for each.

3.2.2 Connection Protocol for Client Agents

Client agents connect to the system through the following procedure:

- Client agent sends a "register message" to the manager agent.
- Manager checks if the client has already connected to the grid before. If the client is requesting to connect for the first time, the manager creates a client delegate agent for the client and pairs them. Otherwise, it runs the reconnection protocol.
- The client delegate agent sends an "accepted message" to the client agent which it represents.
- Client agent sends back an "info message" to its delegate agent. The body of this message contains client information.
- Consequently, the client agent is connected to system and is ready to issue tasks.

Client agents can temporally disconnect after transmitting a task execution request. The system allows clients to reconnect later and receive the results.

3.2.3 Connection Protocol for Worker Agents

Worker agents connect to the system with a very similar protocol to the client agents' protocol. However, the grid system does not allow the workers to reconnect. The details of the protocol are as follows:

- Worker sends a "register message" to the manager agent.
- Manager checks if the worker agent has connected to the grid before. If the worker is trying to connect for the first time, the manager creates a worker delegate agent for the worker and pairs them. Otherwise, a delegate already exists. Delegate sends "accepted message" to the worker.
- Worker sends an "info message" to the delegate agent.
- Thus, the worker agent connects to the grid system and is ready for task assignment and execution.

3.2.4 Task Assignment Protocol

Task assignment procedure and recovery from probable errors are vital issues for grid systems. During the design phase of AGrid, special care has been taken to develop an effective and flexible task assignment protocol in order to minimize runtime errors as much as possible. The details of the protocol are as the following.

- A client agent reports its identification information during the connection protocol. The "info message" which it sends to its delegate contains tasks which are to be processed in the grid.
- Each client delegate keeps track of tasks issued by its client agent. It maintains two task lists: a list that contains pending tasks and another that contains previously assigned and running tasks. Initially, all tasks requested to be executed in the info message are added to pending task list. Afterwards, tasks are moved to the running tasks list after they are assigned to worker agents.
- Client delegate agent checks the pending tasks list periodically and sends task assignment requests to the manager agent.
- The manager agent consults worker delegates in order to locate a free and suitable worker agent that meets the requirements of the task. Worker delegate accepts the request if the worker agent satisfies the task requirements and is currently available, as it is constantly informed about the status of the worker. Otherwise, the request is rejected.
- If the manager can locate an appropriate worker, it sends a "task execute request" message to the worker's delegate agent.
- The delegate agent forwards the request message to its worker agent.
- Worker agent accepts the task and begins to execute it.
- Worker delegate sends "confirm message" to the manager agent.
- Worker delegate moves the task from pending tasks list to running tasks list.

3.2.5 Monitoring Work Load of the Worker

One of the important parts of the task assignment is monitoring the work load of the machines where worker agents are located. Manager agent should assign tasks to non-busy workers. Each worker agent measures its work load periodically. If the state of the worker changes, it informs its delegate agent. In the initial versions of the system, the manager agent kept two separate lists for free and busy workers. However, as this approach caused too many synchronization problems, we decided to have each worker delegate to keep track of the workload state of its worker agent, in the final version.

3.2.6 Agent Disconnection Protocols

During the lifetime of the grid, client and worker agents may be in either connected or disconnected state. For a robust, stable grid system, the manager agent needs to be aware of the states in which client and worker agents are. It is clear that not noticing the disconnection of an agent has a negative effect on the stability of the grid system. It may result in the loss of some tasks or in the assignment of tasks to worker agents that no longer exist. Consequently, the grid system will have to run error recovery protocols. Therefore, one of the responsibilities of the manager agent is to detect disconnected agents. Hence, a flexible protocol for disconnection is defined on account of increasing the stability of the system.

As delegate agents are created and destroyed by the manager agent itself, there is no need for a disconnection protocol for them. Disconnection protocols for the worker and client agents are different. As stated in the connection protocol, while client agents can temporarily disconnect, a worker agent's disconnection is permanent. Agents periodically send an "alive message" to their delegates in order to indicate that they are still connected to the grid system. If a delegate agent does not receive this type of a message from its pair agent for a certain period of time, it decides that the participant agent has quit the system. In the normal case, a participant agent is expected to inform its delegate agent about its disconnection by sending an " disconnect message".

3.2.7 Disconnection Protocol for Client Agents

Two disconnection protocols are defined for client agents; one for temporary disconnection and one for permanent disconnection.

Temporary disconnection protocol is the following:

- A client delegate decides that the client agent has quit the system because either the client has sent an "disconnect message" or an "alive message" has not been received from the client for a certain period of time.
- Client delegate stops checking alive messages and waits for a "reconnect message from the client agent.

A client agent may decide to permanently leave the system before the computation it has requested completes. In this case the following protocol is executed:

- A client agent sends a "quit message" to its delegate agent.
- The delegate agent informs the manager agent about the disconnection of the client agent.

- The manager agent locates the delegate agents of the workers which are running tasks on behalf of the quitting client and informs them of the situation.
- Each worker delegate sends a message to its worker agent, requesting it to stop task execution.
- Each worker agent terminates its task execution thread and sends a "worker stopped" message to its delegate agent.
- After receiving "worker stopped" messages from every worker delegate, the manager agent destroys the client delegate agent and deletes information records about the client agent.

3.2.8 Reconnection Protocol for Client Agents

As stated before, a client agent may disconnect after submitting its tasks and, after a while, it may reconnect to the system in order to receive the results. Even though the reconnection protocol is very similar to the initial connection protocol, there are some major differences between them. In the reconnection protocol, the client agent does not send identification information as this information is already present in the system. The corresponding protocol is the following:

- Client agent sends "register message" to the manager agent
- The manager agent checks if this agent has connected to the system before.
- If the client has connected before, manager sends "accept reconnect message" to the delegate agent of the client.
- Client delegate agent sends a message to the client which indicates that its reconnection request has been accepted.
- Client delegate sends the results of the completed tasks.
- Client delegate starts to wait for "alive messages", that are periodically sent by the client agent in order to show that it is still connected to the system
- The client agent starts to wait for the result of its tasks.

4. TESTS AND ASSESSMENT

We have not yet carried out extensive experiments to observe and assess the performance of AGrid under various workloads and with varying number of participating workers. However, for a preliminary assessment, we have implemented a parallel matrix multiplication algorithm. Since multiplying an mXn matrix with an nXp matrix results in an mXp matrix, mXp tasks are created to compute the resultant matrix. We used 5X6 and 6X7 matrices for the test, which required a total number of 35 tasks. We ran several instances to observe the effect of the increasing number of clients and workers. The minimum, maximum and average calculation time were evaluated.

In the first test run, only one client was introduced and the number of the workers were varied between 1 to 100. As seen in the Figure 3, the working time decreases as the number of workers increases. After the number of the workers reaches nearly 40, working time stays constant, as expected.

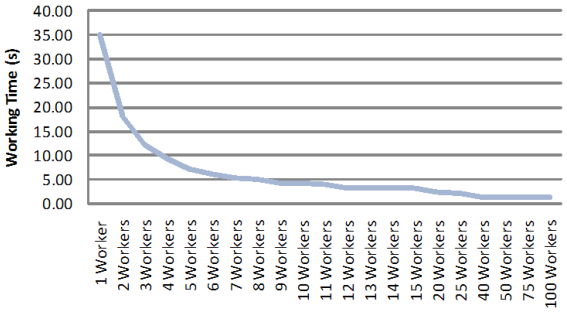


Figure 3: Results of first test.

In the second test run, the number of workers were kept constant while the number of the clients were increased. The test results for 15 workers is given in the Figure 4. As the number of clients increases, the number of the tasks also increases. As seen in the figure, total working time for the calculation increases as the number of independent tasks increases.

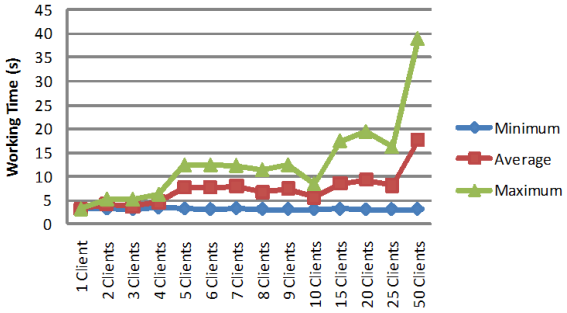


Figure 4: Results of second test.

5. CONCLUSIONS

AGrid is an agent based infrastructure for distributed and parallel computing. It combines the favorable aspects of grid computing and agent technology, producing a robust yet flexible execution environment. The system is implemented on JADE and is fully FIPA compliant. Agents with dedicated roles make up the framework. Client agents are users of AGrid that connect to the grid system to receive service. Worker agents connect to the system to share any of their resources. AGrid has a centralized control, with a manager agent in charge of general grid management. Delegate agents cooperate with the manager agent, reducing its workload significantly. Agent coordination and cooperation through well designed protocols has resulted in a robust, stable grid execution environment.

Even though experiments we have carried out to evaluate the performance of the system are not yet adequate, the results are promising. We have observed that system response time is within acceptable borders and the system is scalable, capable of serving large numbers of clients.

Currently, work is going on to enhance the system, to spot bottlenecks to optimize execution time. Our future work will

be on new protocols for dynamic load balancing to adapt to changing computation needs and changing computing resource environments. It is a fact that large amounts of data that accumulate on the manager agent may overload it, resulting in inefficient scheduling of tasks. Currently, work is continuing on a new version of task assignment policy, where market based algorithms are employed. Auctions are held to determine worker agents which can provide a requested service. These algorithms are carried out by specialized agents, thus decreasing the workload of the manager agent.

6. REFERENCES

[1] T. E. Athanaileas, N. D. Tselikas, G. V. Tsoulos, and D. I. Kaklamani. An agent-based framework for integrating mobility into grid services. In *MOBILWARE '08: Proceedings of the 1st international conference on MOBILe Wireless MiddleWARE, Operating Systems, and Applications*, pages 1–6, ICST, Brussels, Belgium, Belgium, 2007. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[2] K. F. N. T. Bart Jacob, Michael Brown. *Introduction to Grid Computing*. IBM Corp., Riverton, NJ, USA, 2005.

[3] F. R. L. Cicerre, E. R. M. Madeira, and L. E. Buzato. Structured process execution middleware for grid computing: Research articles. *Concurr. Comput. : Pract. Exper.*, 18(6):581–594, 2006.

[4] T. T. G. R. Fabio Bellifemine, Giovanni Caire. *JADE Programmer's Guide*. Telecom Italia S.p.A., 2007.

[5] FIPA. *FIPA ACL Message Structure Specification*. Foundation for Intelligent Physical Agents, 2002.

[6] I. Foster, N. R. Jennings, and C. Kesselman. Brain meets brawn: Why grid and agents need each other. In *AAMAS '04: Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 8–15, Washington, DC, USA, 2004. IEEE Computer Society.

[7] M. Fukuda and D. Smith. Uwagents: A mobile agent system optimized for grid computing. In *GCA*, pages 107–113, 2006.

[8] M. Li and M. Baker. *The grid core technologies*. John Wiley & Sons, 2005.

[9] A. Poggi, M. Tomaiuolo, and P. Turci. Extending jade for agent grid applications. In *WETICE '04: Proceedings of the 13th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 352–357, Washington, DC, USA, 2004. IEEE Computer Society.

[10] O. W. Van, B. J. Overeinder, N. J. E. Wijngaards, M. V. Steen, and F. M. T. Brazier. Multi-agent support for internet-scale grid management. In *AISB'02 Symposium on AI and Grid Computing*, pages 18–22, 2002.

[11] M. Wooldridge. Agent-based software engineering. *Software Engineering. IEE Proceedings- [see also Software, IEE Proceedings]*, 144(1):26–37, 1997.

The Development of a Middleware Tool for Extending a MAS to a Normative MAS

Farnaz Derakhshan
Electronic and Computer Eng.Dept,
University of Tabriz
29th Bahman St
Tabriz, Iran
+98 910 400 4125
derakhshan@tabrizu.ac.ir

ABSTRACT
Open multiagent systems typically require the participating agents to comply with a set of regulations, or norms, and may punish non-compliance. A remarkable challenge for designers of such systems is how to provide discipline in multiagent systems. In this paper, we present our novel and independent middleware tool developed to enable a normative architecture to be overlaid on any type of multiagent system. We achieve this by developing an independent tool in a manner which obtains its inputs from a normative knowledge base and events/actions of the multiagent system, then provides run-time assignments of norms to agents. To overlay development, we propose requirements of the system to include a normative knowledge base containing explicit representation of norms as conditional rules, where the conditions for instantiation of a norm are various runtime occurrences, such as the execution of some agent actions, or the occurrence of some events. Using this normative knowledge base and catching all actions and events in multiagent system, our middleware tool establishes norms such that it can declare the agent's permissions, prohibitions and obligations activated based on runtime actions of agents or even it can detect and report agent's violent behaviours immediately after taking place the violent actions.

Categories and Subject Descriptors
I.2.11 [**Computing Methodologies**]: Distributed Artificial Intelligence, and multiagent systems.

General Terms
Design, Experimentation, Legal Aspects.

Keywords
Multiagent Systems, Normative Multiagent Systems, Design, Implementation, Development, Norms, Conditional Norms.

1. INTRODUCTION
Using notions from human social theory in multiagent systems (MASs) is now well established and even appears in the basic foundations of agent theory. Since norms play a very important role in many social phenomena, there is an increasing interest in using norms in MASs as well.

In [2, 3], the need for defining normative MASs is discussed from two aspects. On the one hand, there are several social viewpoints on MASs from the basic agent concepts such as coordination, organization and communication to an artificial model of human societies. On the other hand, in comparison with the use of norms as a key issue in human social systems, it seems norms may be necessary too for artificial agents in MASs that collaborate with humans, or display human-like behaviors. A comprehensive definition of normative MASs is defined in [4] by Boella.

Since the enforcement of norms applicable to agents is very important task in theoretical viewpoint, we motivated to develop a middleware tool providing normative structure over multiagent system to realize the practical view as well. In essential we were trying to find a way to extend typical MAS to a normative MAS.

In this paper, the development of a middleware tool is described. This tool has been designed and implemented for those MASs intended to establish and enforce norms in their system. The aim of this implementation is to facilitate adding normative system to a typical MAS without need to reengineering the MAS or making any changes in design of the MAS. Using this tool over any multiagent system, at any specific time, one can determine which norms are applicable to each specific agent.

Following this introduction, we first explain the requirements of the tool in more detail, followed by presenting our proposed architecture in Section 2.4. Then, in Section 3, we describe the development stages of the tool including design and implementation of the tool. In Section 4, using an Auction example as a typical MAS, we demonstrate the applicability and feasibility of this tool. Then in Section 5, we explicate the main features of this tool followed by a discussion of related work in Section 6. Finally this paper ends with a summary and details of our future work in Section 7.

2. The Requirements
In this section, the requirements of the tool considered in our development are discussed. In the first stage, it was desired to develop a tool can be attached to every MAS and provide the regulative structure over it. This tool was designed absolutely independent of the MAS connected to. This tool interacts with MAS such that it obtains some required inputs from the MAS and delivers its outputs to MAS.

The type of norm implementation we consider for our desired normative MAS is very important and effective in our design criteria. Therefore, in this section, we first describe some main issues with respect to norm categories and then distinguish our choice.

In every normative MAS, it is desired to establish a set of norms the system such that agents should follow them. Primarily, norms in Normative MAS can be categorized in two types: ***protocol-based*** and ***rule-based*** [10].

Protocol-based norms are related to all the necessary conventions for agent interactions. This type of norm establishes the permitted actions at each instant of time, considering the past actions of agents. These protocols are statically designed at design time. This fact means that the system designer defines all norms or regulations of agents in the format of protocols at design time. So at runtime agents simply follow the predefined dialogues of protocols, moving from one state to another. In such a system, agents do not have any autonomy to deviate from these system norms.

Rule-based norms are defined by a certain type of first-order formulae that set up a dependency relation between actions. These norms specify that under certain conditions, new commitments will be produced for agents to do some actions.

The rule-based norms are *statically* defined in the knowledge base of the normative MAS, but the execution of the rule-based norms for agents is a *dynamic* task which is executed at runtime. In the normative knowledge base (KB), all obligations, permissions, prohibitions and rights of agents are defined. At runtime, based on the actions of the agents and the regulations in the KB, the system detects that the action was acceptable or a violation occurs; if there was a violation, then a sanction should be executed.

The type of normative system we are motivated to design is of the rule-based one; because considering the rule-based norms in normative MAS is more realistic in its norm treatment than the protocol-based ones. In this case, we make no assumptions regarding the decision making architecture of the agent in the MAS. Agents have autonomy to may follow or violate the norms. Thus, this tool requires firstly to have a knowledge base contains all norms and enforcement norms; secondly, a rule base inference engine to perform reasoning task.

As a result, next we describe normative knowledge base, and then explain the inference engine we use in this tool. After that, we describe events and actions of the MAS as a kind of inputs for our proposed application.

2.1 Normative Knowledge Base

One of the main elements of this tool is a normative knowledge base. The normative KB stores all the norms in the normative multi-agent system. Using a descriptive normative language all norms associated with roles can be formally defined. In [5] , we proposed a normative language in which we addressed different types of norms found in legal systems including various types of legal modalities [13] such as obligation, prohibition, permission and right; enforcement modalities [15, 16] such as punishment, reward and compensation; and several key elements of norms such as addressee [13], beneficiary [12], temporal notions such as currency and deadlines [13, 16], and preconditions activating norms [16] that we will describe them in Section 3.1.6..

In addition to these norms, this tool must insert additional processes to manipulate dynamic tasks in runtime, that we will describe them in Section 3.1.6.

2.2 Using Jess

For performing reasoning tasks in this architecture we use the Jess rule engine [8]. Jess is a java-based rule engine and its java APIs can be simply used in java applications as well. This rule base engine is used by a variety of users in many different application domains.

Similar to other typical rule-based systems, Jess has three main components [9]: a rule base (or Jess knowledge base), a fact base (or working memory) and an inference engine. The rule base contains all the norms the system knows. The contents of this rule base are stored in a format that the inference engine can work with. The fact base contains information which the inference engine will operate on. Whenever the inference engine is invoked, it has to decide what rules can be fired based on the rule base and the fact base; such that if the existing facts in fact base satisfy the conditions of rules in the rule base, those rules are fired. Once the inference engine decides what rules are to be fired, it has to execute the actions of those selected rules.

The role of Jess in our method can be explained as follows: our approach provides a *Jess rule base*, a *Jess fact base* and the *invocation command* for executing Jess inference engine. The *rule base* is supplied by our normative KB and created by the legislator or the normative system designer. The contents of the *fact base* are the facts of runtime occurrences which are dynamic and frequently updated in runtime. In our approach, after occurrence of each change (followed by asserting the change to the fact base), the inference engine will be invoked by executing *run ( )* command in order to perform a reasoning task.

After running Jess, Jess may return some new results; such as a list of recently fired norms. The recently fired norms are new facts which should be caught and analyzed in our approach.

2.3 Events and Actions

As we mentioned, this tool is connected to a MAS in order to validate agent's behaviors to be according to law on the basis of predefined normative knowledge base. To do so, our tool requires obtaining all important changes in the MAS. When a change in the MAS occurs we say that a *Runtime Occurrence* takes place. Here, we categorize the following types of runtime occurrences in a typical MAS as follows:

- variations in the agent population as agents enter or leave the system;
- the occurrence of actions performed by agents and other system events;
- changes in environmental parameters, such as price;
- the achievement of important times.

As the result of a runtime occurrence, the conditions for one or more norms of the normative KB may become satisfied, and so rights, responsibilities and sanctions need to be assigned to one or more agents. Therefore, we can use the runtime occurrences as

the triggers to instantiate a dynamic assignment or re-assignment of normative consequences to agents in a system. For example:

Norm1: *"The Auctioneer is obliged to reject lower bids, during the auction session."*
Norm2: *"During the auction session, if a lower bid is placed and Auctioneer did not reject it, punishment_ 2 will be applied to the Auctioneer."*

According to Norm1, the obligation is activated and assigned to the Auctioneer agent only during the auction session. Norm2 shows that if auctioneer agent violates, s/he will be punished. So if the condition of this norm is satisfied it will be activated and assigned to the Auctioneer. As a result, the activation and deactivation of the above norms is subject to the conditions of time (during the auction), event (place a lower bid) and action (rejection of bid). Thus the activation and deactivation of each specific norm happens dynamically at runtime. So assigning each activated norm to the relevant agent will be a dynamic task too.

2.4 Architecture

Using the above requirements, we present a general architecture for implementation of our tool. The full description of this architecture can be found in [5]. Using this architecture, it is possible to implement a stand alone tool over a MAS intended to facilitate transforming MAS to normative MAS.

This architecture along with the issues we have already mentioned for defining roles and normative KBs provides the complete picture for design and implementation of these techniques for such assignments. This implementation is independent of the design of MAS and can be implemented over a pre-developed MAS to provide this facility.

3. The Development of a Middleware Tool

In this section, we explain the development stages of this tool including analysis and design through to implementation. This tool is not application-specific, but is generic, and so may be incorporated into any MAS which intended to be normative MAS.

First, we explain the analysis stage along with the description of the functionality, inputs and outputs of this tool. Next, we describe the design, followed by the implementation of the tool.

3.1 Analysis and Design

This tool is implemented based on general architecture along with the creation of a normative KB. Here we explain the general functionality of the tool, specify inputs and outputs of the tool and design issues including tool's entities, normative KB, event/action simulator, timer and user interface.

3.1.1 The functionality of the tool

The main work of this tool is started at runtime. This tool is connected to a MAS and obtains all events and actions dynamically as they occur, such as entry and exit of agents, the actions agents undertake and any environmental events. For each of these runtime occurrences the following tasks are undertaken, in a continuous cycle:

- Our tool first asserts this event as a new fact in the Jess fact base, then, activates Jess to perform reasoning task. The Jess inference engine undertakes the reasoning task using the rule base (the normative KB) and fact base.
- Next, this tool collects the results of Jess reasoning and analysis of these data. The result of this analysis would be a set of new assigned norms to controller agents of the MAS or a set of new assigned norms to agents.
- Then the collected results are reported to the MAS.

3.1.2 Inputs and Outputs of the Tool

Using the above description of the functionality of the middleware tool, identifying the inputs and outputs of the tool is very straightforward. One of the main inputs of the tool is the normative KB including the main rules describing all obligations, prohibitions, permissions, and rights of agents along with the temporal functions and also enforcement of norms including punishment, reward and compensation. This input is provided by normative designer or the legislator of the normative system.

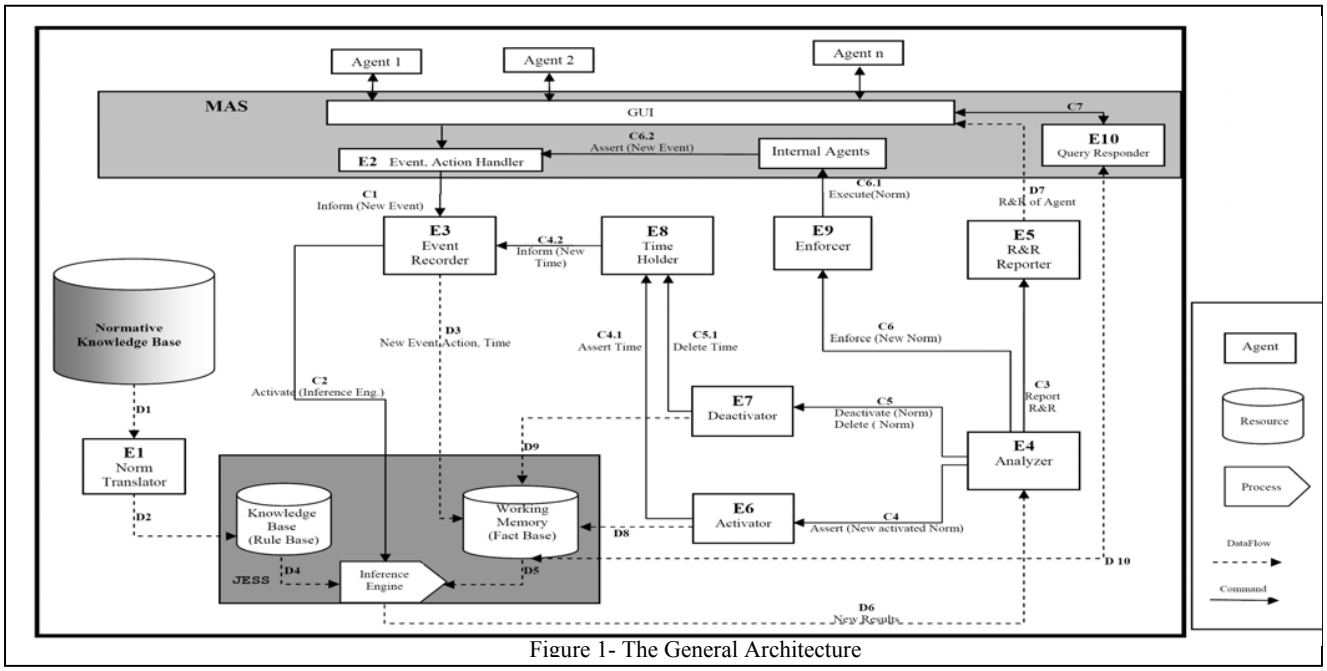


Figure 1- The General Architecture

The other inputs are the runtime occurrences which includes events, actions, and time. Runtime events and actions can be provided by the Event/Action Handler component of the MAS. For time inputs, the system needs timers for announcing the important times (detected inside the application); so the application needs to use the system's clock to provide such inputs.

This tool has two types of outputs: the first one is the result of dynamic assignment of rights, responsibilities and sanctions to agents of the MAS which at each instant of time presents what rights and/or responsibilities have recently been allocated to each agent.

The other output is the enforcement instructions for controller agents of the MAS. As mentioned, our rule base contains enforcement norms. When the tool executes norms of the system, the enforcement norms will be fired as well. These enforcement norms contain punishments, compensations, or rewards, all of which should be enforced by controller agents of the MAS over agents. In our application, every punishment, compensation or reward norm has an instruction code for controller agents. Based on this code, the controller agent can execute the related instruction for punishment, compensation or reward.

3.1.3 Design of Tool Entities

After identification of the functionality of the tool and the inputs and outputs of that, we now present the design of the application based on the analysis just presented. Diagrams of the software engineering design, including a use case diagram, an activity diagram and a class diagram, are presented with detailed descriptions for each diagram in [5].

3.1.4 Design of Normative Knowledge Base

The creation of the normative KB is a design task and the normative KB should be created by the system designer at design time. Then, this KB will be used as an input of the tool.

The normative KB contains all the main norms and the enforcement norms. Norms specify which role is obliged to/or prohibited from/or permitted to/or has the right to do which actions. Enforcement norms of the normative KB specify the responses of the normative MAS if the desired action has not been executed by the agents who play a role.

In addition to norms and enforcement norms, this tool must insert additional processes (in norm format) to manipulate dynamic tasks in runtime.

In the following, first we outline two important issues for designing of the normative knowledge base: The type of normative KB and the descriptive normative language of KB.

3.1.5 The Type of the Normative KB

In our general architecture, we do not limit the designer to use any special type of knowledge base for creating a normative knowledge base. The important concern is that this normative knowledge base is used by an inference engine, so the type of KB should be compatible with that inference engine. If the type of knowledge base is not compatible with the inference engine, a translator should be used to translate the KB to the language of the inference engine. In our architecture we also anticipate such a translator or transformer.

The inference engine we use in our general architecture is Jess rule engine [8]. To avoid translation stage, we create our KB based on Jess rule language and all examples in the following are also in Jess language.

3.1.6 Defining Norms in Normative KB

For describing norms, a normative knowledge base should be created based on a descriptive normative language. The descriptive normative language contains the primary elements of main norms and enforcement norms. We explained our descriptive normative language in detail, in [5], followed by a presentation of a formalism for such a language.

The definition of templates is one of the main features of the Jess rule base language. Therefore, here, we also define a Jess template for "*norm"* and *"enforcement norm"* comprising all the above elements of our formalism. The slots of this template are based on the key elements of main norms and enforcement norms. In the following, we defined Jess templates for "*norm"* and *"enforcement norm"* in our normative KB:

(deftemplate norm (multislot status) (slot deoMode) (slot act) (slot addressee) (slot benef) (multislot object) (slot timeMode) (multislot time))

(deftemplate enforcementNorm (slot status)(slot addressee)(slot EnfCode))

We categorize norms of the normative KB in two types: *domain-related rules* and *general rules*. Domain-related rules (including norms and enforcement norms) are the norms specifically defined for the application domain. These norms should be defined by the legislator or the system designer. For instance, the following rule shows an example of domain-related norm created by a legislator for auction domain. This rule says "*If seller advertises an item, Seller is not allowed to place a Bid during the Auction Time (between Start and End Time)*". In addition, the legislator of a normative KB is also responsible for defining reaction norms of the normative system. We defined a template for reaction norms as *enforcementNorm*. Such reaction norms determine that the system has anticipated what punishment or compensation for the case of violations, or what reward has been considered, for enactment of the various norms.

ER1 listed in Section 4.1 is an example of an enforcement norm which is defined by the legislator of the system. This rule says *"If the obligation of the act of placeHigherBid is violated, the addressee is punished by P1"*. P1 is an instruction code which should be run by internal agents of the multiagent system.

General rules are some norms which can be used in the normative KB of every model intended to use our tool. The definition of these general rules is one of the main features of our approach. General rules include a set of necessary rules for execution of commands. These rules are general and are not specific to an application. These general rules are automatically added to the normative KB by our tool at system runtime.

A number of of general rules are responsible for the modification of the status of the norms. This task is performed by, the command mode of norms in the formalization of normative commands (we defined in [5]) such that $CommandMode \in \{ToBeActivated, Activated, Deactivated, Fulfilled, Violated\}$ Then, in the template of norms, we considered a slot named "*status"* slot to represent this mode. The status of a norm is

changed at runtime when such modifications of modes are provided by our general rules. The modification of modes can be summarized as follows:

ToBeACTIVATED→ ACTIVATED

ACTIVATED → DEACTIVATED FULFILLED

ACTIVATED → DEACTIVATED VIOLATED

For instance, the following rule shows a general rule.

(defrule statusChangeForObligationNorm

?eventFact<-(event (act ?x)(actor ?y)(AtTime ?t))

?normFact<-(norm (status ACTIVATED)(deoMode Obl)(act ?x)(addressee ?y))

=> (duplicate ?normFact (status DEACTIVATED FULFILLED)

(timeMode AtTime) (time ?t)) (retract ?normFact))

This norm detects when any obligation action is not fulfilled by the deadline. So if an obligation norm satisfies the status of the activated norm it should be deactivated and labeled as fulfilled. The description of this norm states that if an event occurs (such that *actor "y" does the act "x" at time "t"*) and this event matches an activated norm (which says *actor "y" is obliged to do act "x"*), it means that the norm is fulfilled and the status of the norm should be changed to DEACTIVATED mode and labeled as FULFILLED at the time immediately after occurrence of the event.

3.1.7 Design of the Simulator of Event/Action Handler

One of the main inputs of the system is events and actions which occur in the MAS at runtime. In order to focus on the normative part of the system, we have simply simulated these inputs, using a class called the InputSimulator class which generates events/actions for the system as if they were runtime occurrences in a real MAS such as auction system.

Recording the Time of Events: As we use a simulator to simulate occurrences of events and actions, when the occurred event/action is asserted to the Jess fact base, the time of occurrence is also added to the fact to show at what exact time the event/action happened. Later, the At(t) function will be described in order to represent the time of occurrence of the time.

These events/actions are passed to the tool with the format of "event" template as follows:

(deftemplate event (slot act)(slot actor)(slot forPerson)(multislot object)(slot AtTime))

3.1.8 Design of Timer

Time is another important issue in this tool. As mentioned before, norms are mostly time-related and they contain the notion of times using the temporal functions such as before(t), after(t) and between(t1,t2) according to the grammar of the full normative language. In addition to these three functions there is another implicit time notion for the time of occurrences of event. As we use an event simulator, we add another function, namely At (t), showing the time of event/action occurrence.

When a norm is activated, the values of time functions indicate the important times for the status of the norm. Such an important time may be a start time for activation of an obligation, prohibition, permission or right. Or it may be a deadline for an activated norm and after that time norm should be deactivated.

For recording of time and date and their management, we use a timestamping method. *"A timestamp is a sequence of characters, denoting the date and/or time at which a certain event occurred."* [7]. Using a timestamp allows for easy comparison of different records and tracking progress over time.

The format of our timestamps is based on the IETF standard date syntax recognized by java.util.Date and also used by the Jess language for timestamps. The source of time for this standard is 12:00 AM on 1/Jan/1970 and contains both time and date.

In general, the process of time management in this tool has the following stages:

1. Events are recorded into the fact base at the time of occurrences with a timestamp.

2. This timestamp is kept in the timer list of the application. The timer notices whenever the current system time matches the timestamp, when the current time (notified by timer), will be asserted to Jess fact base. The format of current time has been defined as "currentTime" template as follows:

(deftemplate currentTime (slot value))

3. The Jess inference engine will be activated by asserting the current time, then the facts related to the current time will be activated.

3.1.9 Design of User Interface

This tool has very simple user interface. For every agent that has joined to the system, the tool creates a frame. Then all the assignments of rights and responsibilities, of norms and assignment of sanctions relevant to this agent, are dynamically reported in the output frame. Whenever an agent leaves the system the associated frame will be apparently destroyed.

3.2 Implementation

In the development process, the subsequent stage after analysis and design is implementation and testing. In this section, we provide some general description for implementation of this middleware tool.

Technically, we built this application using the Java language. The development environment we used for this middleware tool was NetBeans IDE version 6.1. We selected Java because our application is required to connect to any MAS and Java has the capability of compatibility for such a connection. Moreover, the inference engine we used for performing the task of reasoning was the Jess Rule Engine which is based on Java and its Java APIs can be simply imported and applied in any Java program, as we used.

One of the other features of this implementation is we used a simple function for translating assigned norms (which are the outputs) to natural language. This feature is very helpful because understanding the natural language format of outputs is much easier than the Jess format of them.

Java source of this application is available in [5].

4. Auction: A Case Study

In order to test our application, we used a realistic auction example for testing this tool, which works fine. However, in this Section, we just limit this example to the Buy section of auction.

For testing, we should provide some inputs for application and then obtain the outputs of the application. As mentioned before, the inputs of this tool are the normative KB rule base of the system designed by the legislator, and the runtime occurrences provided by MAS. We also simulate runtime occurrences (as the inputs from the MAS) and launch these inputs to the application for obtaining the outputs of our application which are the results of dynamic assignment of rights, responsibilities and sanctions to agents of the MAS.

However, before trying inputs and observing outputs, we provide the list of some norms relevant to Buy section of an auction system; we provide a scenario in which a number of different occurrences in the auction at runtime will happen

4.1 Normative KB for Auction

In order to define the normative knowledge base of this part of auction system, we follow the instruction mentioned in Section 3.1.4. We define the following norms as regulations for purchasing items in our auction. The following norms are either domain-related rules of the normative KB (in which R indicates norms and ER indicates enforcement norms) or general rules indicated by GR.

***;R1**: Buyer is permitted to place a Bid between Start & End Time.*

```
(defrule placingBidPermission
      ?roleFact <-( role (roleTitle buyer)(agentName ?x)(AtTime ?time))
      ?sFact<-(auctionStartTime (value ?startTime))
       (test (> ?time ?startTime))
      => (assert (norm (status ACTIVATED) (deoMode Prm)
          (act placeBid) (addressee ?x)(timeMode BETWEEN)
          (time ?startTime (getEndTime ?startTime))) )   )
```

***;R2**: If Buyer places a bid, s/he is obliged to place higher bid.*

```
(defrule placingHigherBidObligation
       ?roleFact<-( role (roleTitle buyer)(agentName ?x)(AtTime ?time))
       ?sFact<-(auctionStartTime (value ?startTime))
       ?normFact<-(norm (status ACTIVATED)(deoMode Prm)
                  (act placeBid)(addressee ?x))
       ?event <-(event (act placeBid)(actor ?x)
                  (object ?item ?auction )(AtTime ?time))
      =>  (assert (norm (status ACTIVATED) (deoMode Obl)
          (act placeHigherBid) (addressee ?x)
           (object ?item ?auction )(timeMode BETWEEN)
           (time ?time (getEndTime ?startTime))) )  )
```

***;R3**: If Buyer places a Bid, and the bid is a higher bid, the act of placehigherBidder is fulfilled.*

```
(defrule placingBid
     ?roleFact <-( role (roleTitle buyer)(agentName ?x))
     ?event <-(event (act placeHigherBid)(actor ?x)
               (object ?item ?auction )(AtTime ?time))
     =>(assert (norm (status FULFILLED) (deoMode Obl)
        (act placeHigherBid) (addressee ?x)
         (object ?item ?auction ?bid)(timeMode AtTime)(time ?time)) )  )
```

***;R4**: If Buyer places a Lower Bid, Buyer has a punishment :decrease the feedback number.*

```
(defrule placingHigherBidObligation
      ?roleFact <-( role (roleTitle buyer)(agentName ?x))
       ?event <-(event (act placeLowerBid)(actor ?x)
                 (object ?item ?auction ?bid )(AtTime ?time))
        => (assert (norm (status VIOLATED) (deoMode Obl)
            (act placeHigherBid)(addressee ?x)(object ?item ?auction ?bid)
            (timeMode AtTime)(time ?time)) )  )
```

***;R5**: If buyer wins the auction Seller has the right to receive the money from the Buyer.*

```
(defrule receivePaymentRightAndPaymentObligation
      ?roleFact1 <-( role (roleTitle seller)(agentName ?s))
      ?roleFact2 <-( role (roleTitle buyer)(agentName ?b))
      ?sFact<-(auctionStartTime (value ?startTime))
      ?event <-(event (act win)(actor ?b)
                 (object ?item ?auction ?price)(AtTime ?time))
            => (assert (norm (status ACTIVATED) (deoMode Right)
                (act recievePayment) (addressee ?s)(benef ?b)
                (object ?item ?auction ?price)(timeMode BEFORE)
                (time (getPaymentDueTime ?startTime))) )
                (assert (norm (status ACTIVATED) (deoMode Obl)
                (act pay)(addressee ?b)(benef ?s)
                (object ?item  ?auction  ?price) (timeMode BEFORE)
                (time (getPaymentDueTime ?startTime)) ) )  )
```

***;R6**: If buyer pays the price, buyer has the right to get the item.*

```
(defrule getItemRight
      ?roleFact1 <-( role (roleTitle seller)(agentName ?s))
      ?roleFact2 <-( role (roleTitle buyer)(agentName ?b))
      ?sFact<-(auctionStartTime (value ?startTime))
      ?event <-(event (act pay)(actor ?b)(forPerson ?s)
                 (object ?item ?auction ?price)(AtTime ?time))
             => (assert (norm  (status ACTIVATED) (deoMode Right)
                 (act getItem) (addressee ?b) (benef ?s)
                 (object ?item ?auction ?price) (timeMode BEFORE)
                 (time (getSendingDueTime ?time)) ) )  )
```

***;ER1:** the punishment for wrong bids*

```
(defrule wrongBidPunishment
       ?normFact<-(norm (status VIOLATED)
                    (deoMode Obl)(act placeHigherBid)(addressee ?x))
         =>(assert (enforcementNorm (status PUNISHMENT)
            (addressee ?x)
            (EnfCode P1:toDecreaseFeedbackValueOfBuyer)))
            (assert (enforcementNorm (status EXECUTE) (addressee ?x)
             (EnfCode P1))) )
```

***;ER2**: If buyer has feedbacks=-3, Buyer is forbidden to join to the auction.*

```
(defrule agentJointprohibition
     ?feedbackFact<-(feedback (actor ?x)(value -3)(AtTime ?time))
           =>(assert(norm (status ACTIVATED) (deoMode Frb)
                (act auctionJoint) (addressee ?x)(timeMode AFTER)
                (time ?time)) )
               (assert (enforcementNorm (status EXECUTE)
                (addressee ?x) (EnfCode barredMember))) )
```

***;GR1**: General rule for fulfillment of a permission norm.*

```
(defrule statusChangeForPermissionNorm
       ?eventFact<-(event (act ?x)(actor ?y)
                    (object ?z ?p ?q)(AtTime ?t))
       ?normFact<-(norm (status ACTIVATED)(deoMode Prm)
                    (act ?x)(addressee ?y)(object ?z ?p ?q))
             =>( assert (status  FULFILLED)(deoMode Prm)
                (act ?x)(addressee ?y)(object ?z ?p ?q)
                (timeMode AtTime)(time ?t))   )
```

4.2 Scenario

Here we present a scenario of different runtime occurrences in our auction system. The scenario of this auction is as follows: We suppose that *Sarah is Seller* and *Mari* logs into the system as *Buyer*. As an example of environmental variables, we suppose every member has a Feedback variable. We initialize feedbacks of members as : hasFeedback(Sarah,0), hasFeedback(Mari,-2).

As an identification of the auction, the name of the auction is *Auction_1*, and the item for the auction is *Gold Watch*. The starting and ending time of the auction is *"10:00:00 2/1/2008"* and *"11:00:00 2/1/2008"*respectively. The current Bid is *25GBP*.

4.3 Runtime Occurrences and Outputs

Using the above scenario, here we show the result of testing our tool assuming a number of events and actions occurred in the auction MAS. These runtime occurrences along with our defined normative knowledge base are the inputs of our tool. After taking place of each runtime occurrence, our tool provides an output.

Now we suppose that Mari joins the auction system and chooses to be a buyer. At this stage, this tool creates a frame for Mari to show the result of norm assignments to Mari. Here we assume the following actions or events happen:

An Action: Mari places a bid at "10:20:10 2/1/2008".

Mari places a bid by pressing the bid button in the auction web page and MAS reports this event to our tool as follows:

event (act placeBid) (actor Mari)(object GoldWatch Auction_1)

As a result of this occurrence, GR1 and R2 are fired.

An Action: Mari places a lower bid at "10:21:24 2/1/2008".

Suppose that this event happens and the MAS reports the following role assignment to our application:

event (act placeLowerBid)(actor Mari)(object GoldWatch Auction_1 22GBP)

As the result of this occurrence, R4 and ER1 will be fired: The rule R4 shows that a violation occurred, as it changes the status of the rule to VIOLATED. Then rule ER1 specifies the punishment for the person placed lower bid by a code. When this norm is fires, *NormAnalyzer* detects "EXECUTE" and commands to Enforcer for executing. Enforcer passes the code (here P1) of this internal command to the relevant controller agent of the MAS. This code has already been defined for controller agents. In this rule P1 is decreasing the number of feedbacks of the violator agent. As the following snapshot shows the dynamic assignment of sanction to Mari is assigning P1: decreaseFeedbackValueOfBuyer.

An Environmental Event: Enforcer increases the Neg. feedback of Mari at 10:22 2/1/2008.

According to the previous punishment, the value of Mari's feedback has been changed. MAS reports this change to our tool as: *feedback (actor Mari)(value -3)*

When this environmental event reports to our application, ER2 will be activated. This norm specifies that whenever the number of feedback of an agent is -3, it is forbidden for that agent to join to the auction anymore and the name of this agent will be added to the list of barredMembers. Subsequently, controller agents will check the legality of agents as they try to login to the system.

As this snapshot shows, Mari is forbidden to join to the auction after her feedback value reaches to -3.

5. Evaluation of the Tool

Here, we complete the discussion of development stages of this tool with an evaluation of it. The main features and potentials of this tool can be summarized as follows:

First, using this tool reduces the cost of reengineering of a MAS which intended to be normative one. In this case, the MAS can establish a set of regulations with less effort.

Second, this tool is generic, not an application-specific, and so may be incorporated into any multi-agent. It just has interaction to Action/Event Handler of the MAS to provide its required inputs.

Third, from normative viewpoint, this tool covers all aspects of norms (including obligation, permission and prohibition) and enforcement norms (including rewards, punishments and compensations), because the formalism and the normative structure we used in our normative knowledge base are very comprehensive.

Finally, from technical viewpoint, this tool works in a dynamic environment and the enforcement of norms to agents is undertaken dynamically at runtime, not statically at design time. Therefore, this feature makes our application to be well-suited for MASs with open and dynamic environments.

6. Related Work

The working group of IIIA[11] presented an integrated environment called Electronic Institutions Integrated Development Environment (EIDE) to support the engineering of MAS as electronic institutions [1, 14]. The part of this integrated environment most related to our work is the EI's management of external agents. In EIDE, external agents do not participate directly in an electronic institution. Instead, all their interactions are mediated by a tool called AMELI [6] through a special type of internal agent called a *governor*. The governors are part of the social layer of EI and manage the communication of an agent with the other agents in the e-institution. Agents communicate to their governors. Governors check the messages sent by agents within the scenes. If messages are correct and according to the institution specifications, governors transmit them to the addressed agents in the scene, otherwise agent messages are not transmitted.

From a practical point of view, EI has not developed rule-based norms at the application level, although the formalism of these norms has been proposed.

In comparison, we practically developed the formalism (including many kinds of legal modalities of norms such as obligations, prohibitions and permissions) in our proposed application, while so far EI applications have not covered all legal modalities. Moreover, our implemented norms are enforced by the application and in the case of violation will be decided based on the rules predefined by legislator, while enforcement mechanisms have not been implemented in EIs yet.

7. Summary and Future Work

This work deals with the development of a tool over a MAS intended to enforce normative system. To do so, we considered requirements of implementation, followed by presentation of a general architecture. Because of the importance of the normative knowledge base in this technique, we analyzed the structure of such a KB which contains norms and enforcement norms.

Finally, using these implementation issues, we develop a generic application to demonstrate the practical feasibility of our approach and architecture. With this perspective, using an auction example we examined the functionality of this tool.

Here, we mention some of the potential future research avenues which it would be interesting to investigate. First of all, the integration of our approach for dynamic assignment of norms and sanctions to agents with AOSE methodologies could be a valuable effort, because our proposed mechanism helps the management of the MAS, speeds up the dialogues, enforces norms, and reduces the need for a system designer to identify and exclude all behaviors at design time. Currently, no standard AOSE methodologies has this feature.

Secondly, our tools could also be integrated with reputation systems, such as those used on eBay. Currently, eBay has a static system for applying its regulations. Regulations of this auction system are based on predefined protocols or are controlled by its human resources staff to detect violations. These regulations are not assigned dynamically to the buyers or sellers, even when a violation occurs.

As an example, shill bidding -or placing a bid by a seller on their own item directly or through others- is prohibited in eBay. Currently if a seller uses shill bidding, this violation is not detected at runtime and the auction may be continued normally. However, it is possible that eBay staff detects this violation after a while and suspend the account of the seller. So this example shows that assignment of norms in this case - detection of violation - is not a dynamic task in eBay.

As a result, our application would be useful for such repetitive systems to assign rights, responsibilities and sanctions to agents dynamically, for instance, in shill bidding cases.

8. ACKNOWLEDGMENTS

I would like to thank Dr Peter McBurney and Professor Trevor Bench-Capon for their valuable guidance in this research.

9. REFERENCES

[1] J. L. Arcos, M. Esteva, P. Noriega, J. A. Rodr´ıguez-Aguilar and C. Sierra, Engineering Open Environments with Electronic Institutions, Engineering Applications of Artificial Intelligence 18, (2005), 191-204.

[2] G. Boella, R. Damiano, J. Hulstijn and L. v. d. Torre, Role-based Semantics for Agent Communication: Embedding of the 'Mental Attitudes' and 'Social Commitments' Semantics, The 5th Int. AAMAS 2006, ACM Press New York, NY, USA Hakodate, Japan, (2006), 688 - 690.

[3] G. Boella, L. van der Torre and H. Verhagen, Introduction to normative multiagent systems, NorMas Symposium at AISB'05,Hatfield, England, (2005).

[4] G. Boella, H. Verhagen, and L. van der Torre. Introduction to the special issue on normative multi-agent systems. Journal of Autonomous Agents and Multi Agent Systems, 17(1):1–10, 2008.

[5] F. Derakhshan, 'The Implementation of Dynamic Assignment of Rights, Responsibilities and Sanctions to External Agents in Normative Multiagent Systems ', (PhD Thesis, University of Liverpool), (2008), http://www.csc.liv.ac.uk/research/techreports.

[6] M. Esteva, B. Rosell, J. A. R. guez-Aguilar and J. L. Arcos, AMELI: An Agent-Based Middleware for Electronic Institutions, The 3rd Int. Conference on Autonomous Agents and Multiagent Systems (AAMAS 2004) , IEEE Computer Society,New York, NY, USA, (2004), 236-243.

[7] FIPA. Foundation for Intelligent Physical Agents (ACL). 1995 2008 [cited Sept 2008]; http://www.fipa.org.

[8] E. Friedman-Hill, Jess in Action: Rule-Based Systems in Java, Manning, (2003).

[9] E. Friedman-Hill, "What are rule-based systems?" Jess in Action: Rule-Based Systems in Java, Manning, (2003).

[10] A. García-Camino, Pablo Noriega and J. A. Rodríguez-Aguilar, Implementing norms in electronic institutions, 4rd Int. Conference on Autonomous Agents and Multiagent Systems (AAMAS 2005), ACM 2005,Utrecht, The Netherlands, (2005), 667-673.

[11] E. I. Group. Electronic Institutions. 2008 [cited Sept 2008]; Available from: http://e-institutions.iiia.csic.es.

[12] H. Herrestad and C. Krogh, "Obligations directed from bearers to counterparts ", Proceedings of Int. conference on Artificial intelligence and law table of contents, Publisher ACM New York, USA, (1995), 210 - 218.

[13] R. W. V. Kralingen, P. R. S. Visser, T. J. M. Bench---Capon, H. J. V. D. Herik and A principled approach to developing legal knowledge systems, Int. Journal of Human-Computer Studies archive, 51, (1999), no. 6, 1127-1154.

[14] C. Sierra, J. A. Rodrguez-Aguilar, P. Noriega, M. Esteva and J. L. Arcos, Engineering Multi-agent Systems as Electronic Institutions, European Journal for the Informatics Professional, 4, (2004).

[15] G. Therborn, Back to Norms! on the Scope and Dynamics of Norms and Normative Action, Current Sociology, 50, (2002), 863-880.

[16] J. Vázquez-Salceda, H. Aldewereld and F. Dignum, Norms in Multiagent Systems: from Theory to Practice, Int. Journal of Computer Systems Science & Engineering, CRL publishing, 20, (2005), 225-236.

Author Index