About a New Kind of Autonomous, Self-adapting Agents and the Library Game Environment

Herwig Unger, Daniel Berg
Fakutät für Mathematik und Informatik
Lehrgebiet Kommunikationsnetze
58084 Hagen, Germany
kn.wissenschaftler@fernuni-hagen.de

Self adapting agents with local knowledge should be able to manage global ressources in dynamic networks automatically. In this paper the library game is introduced – a simulation which uses the methaphor of migrating libraries that try to find paths through the network in such a way, that the satisfaction of all book-consumers reaches a maximum.

Keywords—self-adaption; learning; agent; Library Game

I. MOTIVATION

Computer networks of big companies today become more and more complex, i.e. not only the number of machines is growing but also the number of connected networks as well as the specialization of the machines within the network (which may equipped with special hard- or software). Furthermore, the dynamics of cooperation in more and more complex development processes result in a permanent change of user groups accessing and changing data and programs on the servers in the system. In such a manner the task of system administrators become an extremly tedious one, since a lot of different requests must be considered. Since no single administrator may oversee the huge amount of factors influencing the system behaviour and also do not have access to configure all the several networks in a respective manner, the system behaviour will be less and less an optimal one. The purpose of the present contribution is to consider a new kind of autonomous and self adapting agents, which are able to make automatic decisions on which machine in a complex system which data shall be stored. Differing to existing works, less theoretical aspects but more applicability is in the foreground of our consideration. With our agents we intend to model a multicriterial decision process, taking user behaviour as well as network parameters and economic influences into account. In order to present our approach in a well defined analyseable environment, we first abstract from real networks and setup a similar environment for our considerations: the so called library game envioronment, which shall also demonstrate similarities in the solution of problems from different areas like computer networks, traffic organization or tasks derived from a financial and economic background. Then we briefly introduce a new kind of an autonoumous, self adapting agent which is able to extract complex behavior in a process similar to human abstraction. The agents can choose from a small set of basic, atomic actions and combine them within a learning- and adaption process to achieve a more complex behaviour in order to increase its satisfaction which is a measure for its success. The last sections of the present contribution describe our simulation setup and the results. In these simulations we address mostly the task where data shall be located to and when it is worth to migrate them to which other places in a complex network. Future work will discuss more detailed the possibilities also to learn pricing strategies and the impact of combinations of different influencing factors.

II. THE LIBRARY GAME SETUP

Following our outline, the library game environment shall be introduced more formally at first. We consider a set of players P, consisting of 2 groups, server or service provider (in our case libraries) S and Customers C. Therefore $P = S \cup C$ and furthermore let us assume for the moment that $S \cap C = 0$, i.e. each player is either a service provider (library) or customer. In detail the set of service providers will be defined by the ordered set $S = \{s_i \mid i = 1(1)o\}$, the set of customers by $C = \{c_j \mid j = 1(1)n\}$ respectively. To simplify the problem, we assume that the game is played in a (geographic) neighborhood, where any 2 players $p_i, p_j \in P$ have a distance given by $d(p_{i,} p_j)$ with $d \in \Re$.

Each player may have some parameters and attributes. In detail we need functions to obtain:

1. $B(p_i)$: the current budget of p_i
2. $I(p_i)$: any fixed payments, costs (-) or income (+) of p_i per time unit
3. $N(p_i)$: a request function for services of p_i in each time unit and finally
4. $F(p_i)$: the current service fee of p_i per service unit for any customer.

It is clear that these values have some special cases, for the moment we can easily see that $I(c_i) \geq 0, I(s_i) \leq 0, N(s_i) = 0$ and $F(c_i) = 0$.

The request $N(c_i) \forall c_i \in C$ is a fixed (over all customers power-law) distributed function, determining how many goods

(books) a customer wants to obtain from any library per time unit (e.g. per month). Consequently, a respective satisfaction σ for all $c_i \in C$ can be defined by

$$\sigma(c_i) = \frac{G(c_i)}{N(c_i)}$$

where $G(c_i)$ is the number of books which c_i with its current budget $B(c_i)$ really could rent from a library in this time unit. It is clear that every customer strives to increase its satisfaction until 1.

To define the library fully, some more rules are needed as follows.

1. Each customer $c_i \in C$ must pay for the transport of goods from and to the libraries; these costs are proportional to the distance. The costs for the transport are $const_1 * d(c_i, s_j)$.
2. The goal of customers is to increase their satisfaction.
3. The goal of libraries is to survive. A library survives, if its budget $B(s_i)$ is $const_2$ months not lower than zero.

Consequently, each customer c_i will rent books at the cheapest location, whereby costs are determined from the server s_j and transport costs from the distance, only. With a monthly fixed income I, the number of goods c_i obtained from s_j can be calculated by

$$(\tilde{g})(c_i, s_j) = \frac{I(c_i)}{(F(s_j) + const_1 \cdot d(c_i, s_j))}$$

With these definitions the book rental Matrix G can be determined for each time unit (month) t by:

$$G(t) = \begin{pmatrix} g(c_1,s_1,t) & g(c_1,s_2,t) & \dots & g(c_1,s_n,t) \\ \dots & \dots & & \dots \\ g(c_o,s_1,t) & g(c_o,s_2,t) & \dots & g(c_o,s_n,t) \end{pmatrix}$$

where t denotes the number of the respective discrete time steps in the game simulation. It is clear, that this matrix may change if the libraries and customers change their location and / or payments are changed. Since every customer tries to obtain the highest possible number of goods in order to increase its satisfaction σ, it is clear that

$$g(c_i, s_j) = \begin{cases} (\tilde{g})(c_i, s_j) \leftrightarrow (\tilde{g})(c_i, s_j) < (\tilde{g})(c_i, s_k) \; \forall k = 1(1)o, k \neq j \\ 0 \quad else \end{cases}$$

In case equal $(\tilde{g})(c_i, s_j)$ exist, one being different from zero is randomly chosen.

Remarks:

1. In $G(t)$ each line (row) represent one customer each column one library
2. After each time step (month) t, the customer satisfaction $\sigma(c_i)$ can therefore be calculated by

$$\sigma(c_i, t) = \frac{\sum_{j=1}^{o} g(c_i, s_j, t)}{N(c_i)}$$

   i.e. a $G(c_i)$ can be defined by

$$G(c_i) = \sum_{j=1}^{o} g(c_i, s_j, t)$$

3. The budget of c_i after t can be calculated by

$$B(c_i, t) := B(c_i, (t-1)) - \sum_{j=1}^{o} [g(c_i, s_j, t)(F(s_j) + const_1 \cdot d(c_i, s_j))] + I(c_i)$$

4. The budget of any library s_j can be determined by

$$B(s_j, t) := B(s_j, (t-1)) - \sum_{i=n}^{o} [g(c_i, s_j, t)(F(s_j)] + I(s_j)$$

The next sections describe the algorithms used by the providers and customers in detail.

III. The Library Game Agent

A. *Definitions to understand the agents' behavior*

It is clear that service providers (libraries) $s_j \in P$ as well as customers $c_i \in P$ are autonomous agents, i.e. isolated units which can act on their own. In later sections we will mostly consider the 2 cases, where service providers may change their location and / or service's fee while most other parameters are fixed so far. However in all cases we intend to make the agents able to learn by one and the same algorithm which will be described in that section in detail. Therefore, of course, some more definitions are needed. Most of them concern values to describe the inner behaviour of an agent and its interaction with its environment.

1. The Event *e*

   An event is a tupel $e = \{z_A, a, z_E, r\}$ where z_A describes the state of the environment the agent may see before it is doing any activity, while z_E is the state the agent sees after completing this activity. Both, z_A and z_E must be from the set of states Z of the considered system. The activites of an agent may be selected from a set of elementary activities $a \in A$, which must be defined and fixed for each agent. Finally, after the execution of an activity, the agent obtains a reward $r \in \Re$.

2. The event tape *E*

All events executed by an agent (or at least a bigger number of τ of such events) are kept on the so called event tape of this agent and may be recalled for any analysis or knowledge extraction. Since we consider always one agent at the moment, we do not add the agent's name to the identifier of the event tape and just call it *E*. Furthermore we write for simplification instead of $Z_a(e_i)$ mostly $Z_{a,i}$. In such a manner the event tape is an ordered set given by $E = \{e_i \mid e_i = (z_{a,i}, a_i, z_E, i, r_i) \wedge z_{a,i+1} = z_{e,i} \forall i = 1(1)(|E|-1)\}$.

Note that $|E|$ denotes the actual length of the event tape.

3. The Intelligence *q* of an agent

Each agent is able to compose several elementary instructions to a sequence of instructions under a name. This ability is something like an abstraction process and naturally the length of sequences learnt will increase with time and experience of the player. To consider this ability of an agent the intelligence q is introduced to be the maximum length of any instruction sequence the agent may store. I.e., the actual intelligence will be denoted by $q = 1(1)q_{\max}$, where $q_{\max}$ is a constant for the maximal reachable intelligence.

4. The Instruction Memory *M*

The instruction sequences which an agent may build in the abstraction process depending on its current intelligence will be kept in a special instuction set memory M. In such a manner $M = \{m_1, m_2, ..., m_c\}$; where c is the maximum number of instruction set to be kept and each instruction $m \in M$ consists of course of a set of events, i.e. $m_i = \{e_1, e_2, ..., e_k\} \forall i = 1(1)c$ Since the instruction sequences are extracted in an abstraction process from the contents of the event tape E it is clear that $z_A(e_{j+1}) = z_E(e_j)$ and $k \le q$. Furthermore, $\forall m_i \in M : m_i \subseteq E$ and finally $|m_i| = k$ is called length of instruction sequence, which must be $\le q$.

5. Distancefunction *d*

In several stages of our library game and other game setups it will be necessary to compare two states z_i and z_j. Not in every case every component of a state is absolutely equal to the other one. That is why a distance function $d(z_i, z_j)$ is introduced and used to determine if two states are similar (almost equal) or not. If $d(z_i, z_j) < \varepsilon$, then z_i and z_j are called similar or equivalent and we write $z_i \sim z_j$.

6. Curiosity *p*

Every learning process requires that the agent is able to explore its environment, i.e. have a random component allowing him to do new, so far undetermined steps in its behavior. The curiosity p of an agent is the probability describing how often the agent does not follow any known instruction sequence but has fully random behaviour for a given amount of steps.

7. Reward of an instruction sequence $R_\sum$

The goal of each agent is of course to get a maximum profit or satisfaction from his activities. So after every activity an agent may obtain a reward $r \in \Re$ from the environment, which can be used to evaluate the success of an activity or instruction set. While the reward of each activity is a simple value, the reward of any sequence of activities is given by the sum of the rewards obtained in each step:

$$R_\sum(e_a, e_b) = \sum_{i=a}^{b} r_i(e_i), a < b, \{e_a .. e_b\} \subset E$$

Shorter we can also write for any $m_i \in M$:

$$R_\sum(m_i) = \sum_{i=1}^{|m_i|} r_i(e_i) \; \forall e_i \in m_i$$

8. LRU (Last recently used memory) *L*

Last but not least we need an intermediate memory for the extraction of information from the event tape E. Mostly it is important, to process parts of E, which are similar / equal to the last processed sequence of events and activities. This is the content of the LRU memory L: $L = \{l_{last}, l_1, l_2, ..., l_k\}$ at discrete time point $t = t_{last}$, i.e. l_{last} is the last executed sequence while higher indices of l_i denote sequences which have been executed in the farer past.

- Let seq_length be the length of the considered (last) instruction sequence on E, given as a fixed value for the moment.
- Then l_{last} is determined by $l_{last} = (e_{t-seq\_length}, ..., e_t)$ with $e_j \in E$
- The other elements of L are given by $l_i \in L \leftrightarrow z_A(e_j(l_{last})) \sim z_a(e_{j-x_i}(l_i))$ $\forall i = 1(1)k \wedge l_i \subset E$

  Of course, j runs from $j = 1(1)seq\_length$. In this process either the whole event tape or only a part of it (a finite history) maybe considered. If we require that
  - $x_i < t$ we obtain an infinite horizon while
  - $x_i < const$ results in a finite case with a finite event horizon of the agent.

B. The agents' algorithm

This section presents the agents' algorithm which is executed by the agents in an infinite loop. Beside writing down all events and the observation of the environment on a tape an agent must meet a decision in each situation. For the implementation we intended to implement a behaviour similar

to the human one. Due to its event tape the agent may remember former situations and may decide whether his behaviour in such states was successful or not. In case a state is reached again and a successful activity sequence is known, the agent repeats it in the hope of another positive reward from doing so. If no successful activity is known, the agent can only try any random, elementary activities. The same the agent shall do in any state from time to time in order to be able to optimize its behaviour or to investigate any new appearing possibilities to obtain a higher reward than so far.

In detail, the agent has to execute the following algorithm.

```
init Z = Z0   // initial state
init E = {}
init M = {}  // init empty event tape and memory
init L = {}  // empty last instruction memory
init q = 1   // reactive behaviour at start


for (;;)
{
   if (p)   // random behaviour
   {
      for i = 1(1)q
      {
         random (a)
         make e = (z, a, zE, r)
         z = zE
         write (E, e)
         //seq_length = q
      }
   }
   else if ( ∃m ∈ M : z_a(e_1(m)) ~ z)
   {
   // Point with known successful
   // instructionset at first search in M
   for i = 1(1)|m_i|  // execute that
      {
         make (ei(mi))
         z := ze(ei(mi))
         write (E, ei)
         //seq_length = |mi|
      }
   }
   else
   {
      t = compose_seq_with_biggest_reward (E, z)
        //compose a sequence by choosing a event
        //sequence from tape E which maximizes the
        //reward!
   if (!isEmpty(t))
      {
         for i = 1(1)|t|
         {
            make (ei(ti))
            z := ze(ei(ti))
            write (E, ei)
         }
      }
      else  // unknown position or no successful
      {     // sequence known here
         random (a)
         make e = (z, a, ze, r)
         z := ze
         write (E, e)
         seq_length := 0
      }
   }
    evaluate (E, M, seq_length)
    // process results from last step
 }
```

Beside reacting to each situation in an optimal manner the agent also has to evaluate and adapt his activities for any situation and to learn the best behaviour. This is done within the function *evaluate()* after each executed step as described in the next subsection.

C. *The evaluation procedure*

The evaluation procedure considers the success of the last executed activity sequence written to the event tape. In order to ensure the success of an instruction sequence, it must be found several times (minimal MIN_l times) on the event tape and during each application should have shown a suitable reward behaviour. Depending on the reward obtained and the state of the instruction memory, the following reactions may be reasonable.

- while the instruction memory still has empty positions, any successful instruction sequence with a reward > 0 can be kept
- if the memory is full, an instruction sequence can be replaced, if the current sequence guarantees a better reward for the same initial state or guarantees a globally higher reward for any other maybe so far unknown initial state.
- if the memory is full and no optimizations can be found for a long time (T_{MAX}) it is assumed that improvements can be found only with higher expenditure, i.e. longer instruction sequences which are only possible with a higher intelligence of the individuum. Therefore in this case the intelligence q must be increased by 1.

The above described behaviour is achieved by the function *evaluate():*

```
function evaluate (E,M, seq_length) {
build (L) // build LRU mem
if |L|<MIN_l break  // no activity, if sequence not
                    // MIN_l t imes on event tape
```

$$\forall l_j \in L \; count \; R_{\Sigma}(l_j) = \sum_{i=1}^{|l_j|} r_i(e_i(l_j))$$

$$set \; \overline{R}_{\Sigma} = \frac{\sum_{j=1}^{|L|} R_{\Sigma}(l_j)}{|L|}$$

```
   //average instruction set reward


if NOTFULL (M) ^ (R_z(l_last)>0) ^ l_last ∉ M
                     //no activity with same z_A
{
   add (M; l_last)
   T = 0
} // fill instruction memory
```

$$\text{if FULL (M) } \wedge \; \forall i : \overline{R}_z >> R(m_i), (m_i \in M)$$

```
{
   MIN_R_replace(M, l_last)
   T = 0
} // replace globally sequences
  // with much less known reward
```

```
if FULL (M) ^ ∃m_i ↔ z_a(m_i) ~ z_A(l_last)(R(m_i) < R̄_Σ)
{
   replace (M, mi, llast)
   T = 0
} // replace a worse sequence in known situation
if T > TMAX  // if no change for
             // a long time, increase intelligence
{
   T := 0
   q := q + 1
}
```

IV. IMPLEMENTATION

The LibraryGame is implemented in Java using the P2PNetSim framework. P2PNetSim is a highly scalable simulation tool that makes it possible to implement large p2p network simulations and many other kinds of simulations within large groups of individuals even based on social interaction.

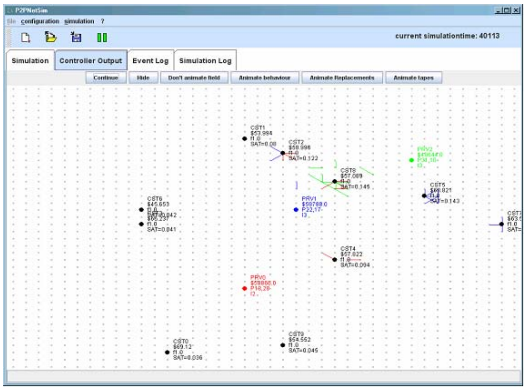

Figure 1: The LibGame in the P2PNetSim Environment

Customers and Providers within this context are modelled as interacting peers communicating via messages with each other. At each simulation time step they can perform actions like requesting services (customers), serve requests and change their positions (providers). *T*=50 time steps form a simulation cycle. Within one cycle the positions of all providers remain constant. After one cycle is over, the customers receive new payments and the providers initiate an evaluation- and migration procedure before the next cycle begins as described in the previous sections.

Some additional parameters have to be considered for the implementation of the LibraryGame simulation:

- Tapesize
  The sections above implied a potential infinite tape size. For the simulation set-up the tape is implemented as FIFO-memory with a fixed size of 1.000.000. To simulate an 'infinite' band this value is high enough, but it strongly depends on the given field-size on which the game is played.

- FieldSize
  This is the dimension of the field on which the library game is played. For the example simulation it is set to 20x20.

- Memorysize
  The maximum number of sequences which can be stored in a provider's memory. Like tapesize, the optimal value for memorysize finally depends on other simulation parameters.

- intelligenceIncreasmentThreshold
  This variable gives the number of cycles, a provider's memory needs to be unchanged before its intelligence is increased. This value is set to 100.

- initialCuriosity
  0 < initialCuriosity < 1: This value gives the propability that a providers chooses a random sequence instead of choosing an existing sequence from tape or from memory. Several simulation-runs showed that this value has to be relatively high (>0.7) to enable the providers to find acceptable sequences. However, if a certain number of sequences was found, such a high value tends to destroy those sequences. In this simulation we used a value of 0.80.

- processesPerCycle
  This variable defines the number of simulation steps a provider remains at a certain position to serve the customers' requests before it migrates to a new position. This value is fixed to 50. A cycle can be understood as somewhat like a month in real-life situation.

- defaultServicePrice ($F(c_i)$)
  The fee a provider gets every time a customer consumes a service. This is fixed to 3.0 for all services of all providers.

- transportCostsPerUnit
  When a customer consumes a services from a provider, the provider has to pay transport costs in addition to the service itself. To calculate this transport costs the Eucledian distance between the customer and provider is multiplied by the transportCostsPerUnit. For each cycle the customer then chooses the provider with the chapeast transport costs. For the simulation discussed here, the transportCostsPerUnit is constantly fixed to 2.0.

- customerPayment
  After each cycle, the customers gets a 'monthly payment' of customerPayment = 2.0

A. *Customer's Setup*

The customers get randomly chosen positions which stay constant during the simulation and a request frequency *0<f<=1*. A request frequency of 1 means, that the customer performs a request at every simulation timestep of a cycle.

The customers commonly have different, power-law distributed request frequencies given by:

$$f_i = \begin{cases} e^{\frac{i \cdot \ln \varepsilon}{n-1}} & \forall 1 < i < n \\ 1 \ \ if \ \ n = 1 \end{cases}$$

where n is the number of customers and ε is the lowest request frequency for customer *n-1*. To simplify simulation analysis, ε is set to 1, meaning that all customers have a request frequency of *1*.

Within a cycle that consists of *T=50* simulation time steps, a provider with a request-frequency of f_i can perform at most $50 f_i$ requests. Usually there will be less requests, because a provider may be too far away and especially in the end of a cycle a customer may have ran out of money. The customer then has to wait for the next cycle, in which he gets his 'monthly payment'.

This results in a decreasing satisfaction value for this customer. A customer's satisfaction for a certain cycle is defined as

$$sat_{Customer_i} = \frac{requests\_performed\_within\_one\_cycle}{simulationsteps\_per\_cycle * f_i} \quad ; 0 \le sat_{customer_i} \le 1$$

The over-all satisfaction then is the average satisfaction over all cycles. The satisfaction of a provider is directly dependant on its balance.

B. *Provider's Setup*

Following the LibraryGame definition the providers actually have to pay for migration-activities. But in first instance we are interested in the paths, a provider takes through the LibraryGame map. So providers don't get a payment at all, and they don't have to pay for position changes. Therefore, the satisfaction of a provider can directly be measured by observing its balance.

C. *Simulation Results*

Figure 2 shows some screenshots of a simulation run with the above discussed setup. One provider (red dot) offers a service to three customers (black dots). The red lines represent the provider's memory state. In a first phase (figure a to b) the sequences stored in memory are concentrated around the customers. The intelligence is not big enough yet, to model paths that are long enough to lead from one customer's position to another while preserving a big reward.

This situation starts changing at $T_{cycle} = 80.000$. The first paths occur which lead from one attractive area to another (figure 1c, from Customer1 to Customer0) Even though this paths could disappear by being replaced by other, more attractive paths, sooner or later they will be re-established and strenghed due to the growing intelligence. (see fig. 2e, 2f). The closer the positions of two providers are, the sooner a path between them will be generated. In this example the relation

$$d(cst0, cst1) < d(cst0, cst2) < d(cst1, cst2)$$

implies the order in which the paths between the customers most likely will be established.

After approximately 0.5 Million cycles and an intelligence of ~60 the memory content shows sequence combinations which finally involve all customers. It is possible to extract one path from the memory which is an optimum to serve all customers and to maximize their satisfactions.

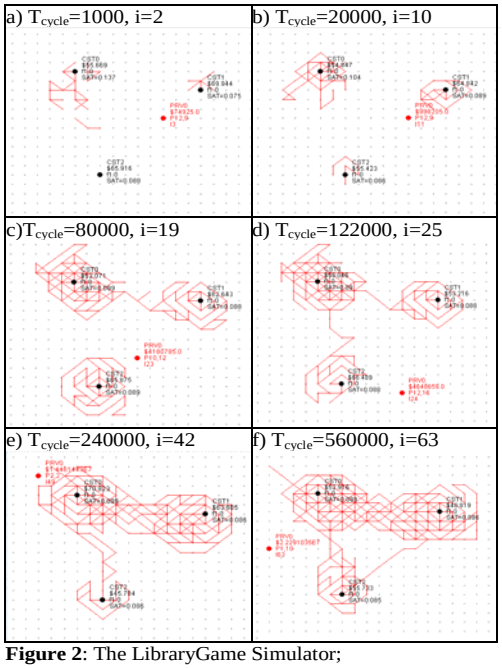

Figure 2: The LibraryGame Simulator;
i: intelligence; T_{Cycle}: simulation cycle – one cycle is 50 simulation steps

V. Discussion & Outlook

The algorithm is able to find an optimal path on which a provider could walk to maximize the statisfaction of all customers. But it still converges very slowly. To speed up the convergence, it may make sense to reduce the providers' curiosity during the game. This would reduce the effect, that sequences which actually are acceptable and could be used as subsequences for better paths just disappear. Also, the composition of random walks could be optimized. One optimisation which is already used in this simulation is to forbid duplicate path sections within the composition of a random sequence. This prevents the customer of performing too many and too tight circles around an attractive position. Another possibility would be to give a preferred direction as a tendency to the providers when composing random walks. This tendency would be changed from time to time and would enforce the provider to leave attractive positions and therefore raise the propability to enter other attractive regions.

In further researches it has to be found out, if this algorithm could be used to make more providers cooperating witch each other in such a way, that the providers dynamically 'share' the customers to provide a maximum over-all-satisfaction for all of them. For this it would make sense to give a 'preferential threshold' to the customers, that make them use a certain provider for a while, even if it is not the cheapest provider anymore. This would correspond to real-life-situations, where consumers do not immediately change the provider when he raises the price. They would still prefer

this provider ‘by habit’ at least a while before they come to the conclusion to change. Providers then would have a chance to leave their (pseudo-) optimal position to explore the environment without having a too high risk of loosing all of their customers.

Further works will accounter competing providers with more differentiated pricing strategies and with a ‘character’. This should enable a provider to decide to be defensive, aggressive or cooperative according to other providers.

VI. REFERENCES

[1] Coltzau, H.: „Specification And Implementation Of A Parallel P2P Network Simulation Environment“, 2005, Diplomarbeit

[2] n.n.: *“Logistic Network Design”*, In: SWISSLOG, Feb. 2007 http://www.swisslog.com/wds-index/wds-consult/wds-cons-network_design.htm

[3] Koller, D., Pfeffer A.: “Representations and Solutions for Game-Theoretic Problems” Artificial Intelligence

[4] Milojicic, D.; Kalogeraki, V.; Lukose, R.; Nagaraja, K.; Pruyne, J.; Richard, B.; Rollins, S. and Xu, Z.: *“Peer-to-Peer Computing”*, Technical Report HPL-2002-57, HP Laboratory Paolo Alto, Mar. 2002.

[5] Brediny, B.; Maheswaranz, R.; Imer, C.; Basar, T.;Kotz, D.; Rus, D.: *“A GameTheoretic Formulation of MultiAgent Resource Allocation”* Proceedings of the Fourth International Conference on Autonomous Agents, *2000*

[6] Bredin, J: *“Market-Based Mobile-Agent Planning: A Thesis Proposal”* 1999, http://citeseer.ist.psu.edu/bredin99marketbased.html

[7] Bredin, J.; Kotz, D.; Rus, D.: *“Economic Markets as a Means of Open Mobile-Agent Systems”*1999, http://citeseer.ist.psu.edu