Managing Information and Time Flow in an Agent-based E-commerce System

Maciej Gawinecki, Maria Ganzha, Paweł Kobzdej, Marcin Paprzycki
Systems Research Institute
Polish Academy of Science
Warsaw, Poland
maciej.gawinecki@ibspan.waw.pl

Costin Badica, Mihnea Scafes, Gabriel-George Popa
Department of Software Engineering
University of Craiova
Craiova, Romania
badica_costin@software.ucv.ro

Abstract

Recently, we have proposed a comprehensive agent-based e-commerce system. While UML formalized, it lacked details how basic functions – e.g. user request to purchase a given product – are to be implemented. Furthermore, the "airline ticket reservation model" used in the system involves time management issues that have not been addressed. The aim of the paper is to discuss the way in which the information flow and data transformations involved in it are to be implemented; assuming that information about products is to be ontologically represented. Furthermore, a simple way in which time information can be successfully managed to support the proposed product reservation approach will be discussed.

1. Introduction

In our recent work we have proposed a model agent-based e-commerce system [5, 6, 7, 14]. While there exist a very large number of papers dealing with agents in e-commerce and agent negotiations in particular, our work differs in the following ways: (1) Typically, only a single price negotiation of an item or a collection of items is contemplated. We are interested in a more realistic scenario when a number of products of a given type are placed for sale one after another – resulting in a series of price negotiations. (2) Since multiple items are sold, our price negotiations are organized differently. In the literature it is usually assumed that agents join an ongoing negotiation as soon as "they are ready" (note that this is the only price negotiation available to them). In our system, we treat price negotiations as a "discrete process." Thus, *buyer agents* are "collected" and released in a group to participate in a *given* price negotiation. While the negotiation takes place *buyer agents* communicate only with the *seller agent*. Meanwhile, a next group of *buyer agents* is collected (as they arrive) and will participate in the *next* negotiation. (3) Since multiple subsequent auctions (involving items of the same product) take place, price negotiation mechanism can change. For instance, first 243 items may be sold using Dutch Auction, while the next 37 items using fixed price with a deep discount. (4) Furthermore, we model a complete e-commerce system, and thus we conceptualize all actions that take place before and after negotiation is completed and may (or may not) result in an actual purchase. (5) While agent mobility is often considered important for e-commerce, conflict between agent mobility and intelligence is rarely recognized. In our work we address this problem by designing modular agents and clearly delineating which modules have to be send, when, by whom and where. (6) Finally, the complete system is being implemented using JADE; an actual agent environment.

It is the latter point that particularly concerns this paper. Thus far our work concentrated mainly on three aspects of the system: (a) agent modularity and mobility [5, 8, 9], (b) rule-based mechanisms in negotiations [1, 2, 7], and (c) UML-based formalization of agents and their interactions [5, 6]. Recently we have moved towards unification of existing parts of the project and towards its complete reimplementation. Additionally, we have decided to utilize OWL [19] demarcated data to semantically describe products traded in our system. As a result we had to re-think information flow that occurs in the system, e.g. when a user-request is to be serviced.

Results of this process are summarized here. In the next section we briefly describe the proposed system, agents appearing in it, as well as their functionalities and interactions. We follow with a description of the information flow in the system. We complete the paper with a brief discussion of the proposed solution to the time management problem that arises when the price negotiation ends successfully and a given product is reserved for a specific time.

2. System Architecture

The proposed system is an attempt to build a comprehensive model of an e-marketplace where *shop*

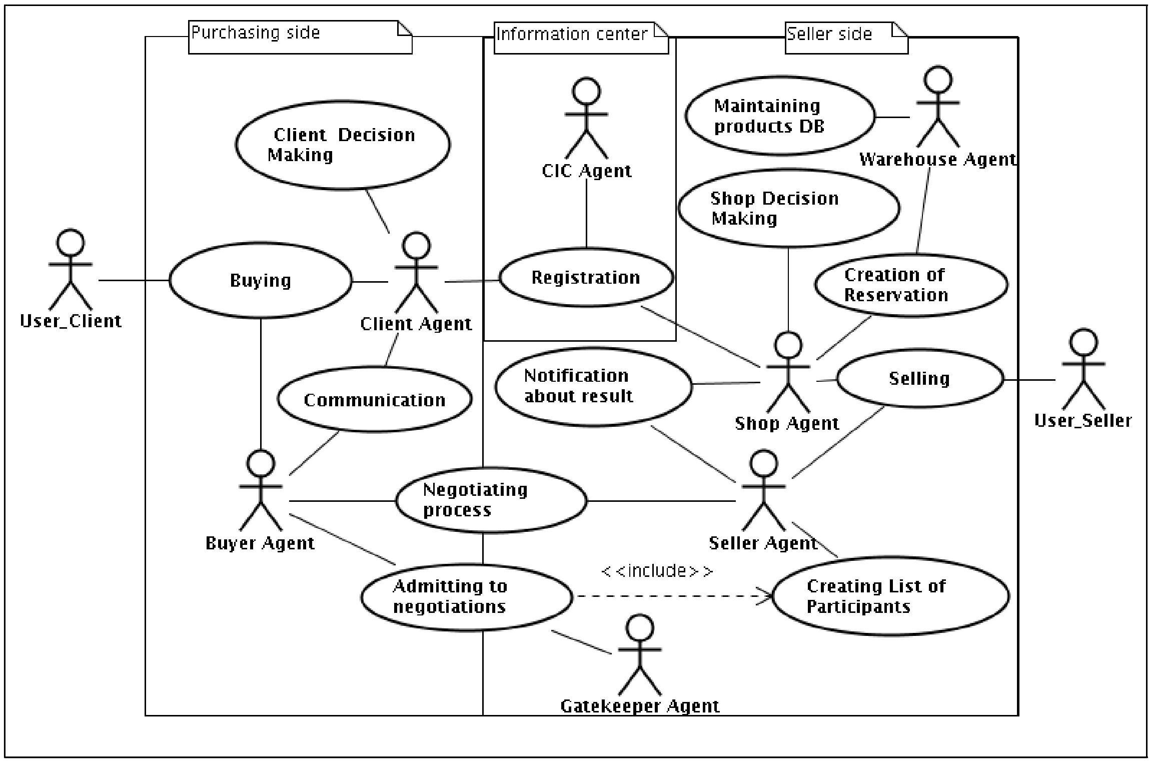

Figure 1. Agent-based e-commerce system – use case diagram

agents, representing *User_Seller*s attempt at selling products to *buyer agents* representing *User_Client*s. The complete use case diagram of the system is presented in Figure 1.

We can distinguish three major parts of the system: (1) the *Information center* where white-page and yellow-page type data is stored – this is our current solution of the matchmaking problem, (2) the *Purchasing side* where agents and activities representing *User_Client* take place, and (3) the *Seller side* where the same is depicted for the *User_Seller*. Let us now briefly describe all agents represented in Figure 1. For an extensive discussion of their functionalities, see [5, 6, 14].

The *CIC agent* is responsible for providing information which e-store in the system sells which products. Information about products and stores is semantically represented – using OWL Lite demarcation (a subset of OWL that is sufficient for our purposes [19]) and persisted in a Jena [17] environment (see next section). The *CIC agent* utilizes a pool of *CIC DB* agents (not depicted here) to handle individual queries. Here, we exploit results reported in [10], where we have experimentally established that such an approach can improve throughput of the querying system.

Within the *Purchasing side* we have the *Client agent* (*CA*) that represents its *User_Client* in autonomously making all necessary decisions related to the purchasing process and multiple *Buyer agents* (*BA*) which actually take part in price negotiations.

The *Seller side* consists of a number of agents that facilitate product sales. The crucial agent is the *Shop agent* (*SA*) which is the central manager of the e-store and autonomously makes all decisions pertinent to selling all products offered by the store. The *SA* is helped by (1) the *Gatekeeper agent* (*GA*) that is responsible for admitting (or not) *BA*s to the host (here it acts as a representative of the *SA* by utilizing *SA* prepared trust-evaluation of each incoming *BA*), management of the process of preparing negotiation which includes, among others registration of participants and supplying them with negotiation template and protocol, and releasing *BA*s to price negotiations; (2) the *Warehouse agent* (*WA*) that is responsible for inventory and product reservations management; and, (3) multiple *Seller agents* (*SeA*) that are directly involved in price negotiations with *BA*s.

A typical usage scenario (one that we are particularly interested in this paper) is as follows (for a detailed description see [5, 6, 14]). Let us assume that system is already initialized and all information about all products sold by all e-stores has been registered with the *CIC*. *User_Client* formulates a request – what product she would like to purchase. The *CA* queries the *CIC* about which stores sell thought after product and then “delivers” (see below) a *BA* to each one of them. *BA*s participate in price negotiations and report results to the *CA*. Based on obtained results, the *CA* makes decision to attempt purchase at one of the stores, to try negotiate a better price or to abandon purchase altogether. Let us now focus our attention on the flow of information that is necessary to facilitate such a scenario.

3. Information flow in the system

3.1 Ontologies in the system

As specified above, we have decided that products sold in the system are to be semantically represented. In this way we represent our belief that the Semantic Web [11] holds the key to the future of the Internet and e-commerce in particular. Since it is ***not*** our goal to develop (or utilize) intricate product ontologies, but to show how an ontologically demarcated data can be used in our system, we have decided (for the time being) to use very simple ontologies. What follows is a snipped of OWL Lite ontology of shoes which are described through following properties: with laces or not, athletic or not, price, brand, color and size.

```
:Product
      a owl:Class .
:Clothing
      rdfs:subClassOf :Product .
:Shoes
      rdfs:subClassOf :Clothing .
:ShoesWithLaces
      rdfs:subClassOf :Shoes .
:AthleticShoes
      rdfs:subClassOf :Shoes .

:hasPrice
      a owl:ObjectProperty ;
      rdfs:domain :Product ;
      rdfs:range :Price .
:hasBrand
      a owl:DatatypeProperty ;
      rdfs:domain :Product ;
      rdfs:range xsd:string .
:hasColor
      a owl:ObjectProperty ;
      rdfs:domain :Product ;
      rdfs:range :Color .
:hasSize
      a owl:ObjectProperty ;
      rdfs:domain :Product ;
      rdfs:range :Size .
:hasSalingInfo
      a owl:ObjectProperty ;
      rdfs:domain :Product ;
      rdfs:range :SalingInfo .
:SalingInfo
      a owl:Class.
:isBuyerCreatedByGateway
      a owl:DatatypeProperty ;
      rdfs:domain :SalingInfo ;
      rdfs:range xsd:boolean .
:isBuyerCreatedByClient
      a owl:DatatypeProperty ;
      rdfs:domain :SalingInfo ;
      rdfs:range xsd:boolean .
```

It is the role of the *SA* to register all products it is selling with the *CIC* agent. Therefore, the *CIC* will represent information as extensions of ontologies of products. The extension involves information about *SA* that sells a given product (specifically, the *GA* of this *SA* – which plays the role of the contact point for the shop). What follows is a snipped of OWL Lite that specifies that the *GA* named `ga509@ibspan.waw.pl` sells a certain product (`Product4094094049`) for an e-shop identified as `767`:

```
:GA-1 a :gatewayAgent;
      :name ga509@ibspan.waw.pl ;
      :addresses
        http://www.ibspan.waw.pl:9999 ;
      :shop 767 ;
      :sells  :Product4094094049.
```

When registering its products with the *CIC*, the *SA* sends an ACL *request* message containing serialized information describing information about sold products (such a message can contain information about one or more products). For instance, for a store (represented by the *GA* known as `ga509@ibspan.waw.pl`) that sells black Athletic shoes with laces manufactured by Nike, that are of size 41 and are sold at a base-cost of 33 Euro, the *SA* would send the following message to the *CIC*:

```
((
  action
    (agent-identifier
      :name cic@ibspan.waw.pl
      :addresses (sequence
        http://www.ibspan.waw.pl:9999)
      )

    Add-Product-Seller
      set
      (Entry
        :product
          (ProductDescription
            :owl "
     :Product4094094049
       a :ShoesWithLaces , :AthleticShoes ;
         :hasBrand "Nike";
         :hasColor :BlackColor ;
         :hasPrice
```

```
          [a :Price ; :ofCurrency :EUR ;
             :value "33.0" ];
          :hasSize
          [a :EuropeanShoeSize ;
             :value "41.0"];
          :hasSalingInfo
           [
             :isBuyerCreatedByGateway true;
             :isBuyerCreatedByClient true
           ] .
           "
          )
          :shop (Side-ID :id 767)
          :gateway
          (agent-identifier
            :name ga509@ibspan.waw.pl
            :addresses (sequence
             http://www.ibspan.waw.pl:9999)
             )
          )
))
```

Let us make a few comments. First, in our approach we utilize, in a very natural way, two different ontology languages: FIPA SL language [12] for demarcating actions that the CIC is requested to perform (e.g. *Add-Product-Seller*) and OWL Lite to formalize product description (marked in bold). Second, for purpose of simplicity (and sacrificing somewhat brevity and prudence in resource utilization) we have decided to represent different "variants" of a given product as separate products. This being the case, if certain shoes, are available in sizes: 39, 40, 41, and 42, they will be represented in the *CIC* database as four different products. We will evaluate performance of this approach and, in the future, may opt for a more compact product representation (which is likely to result in a more complicated code to service it). Finally, all agents processing information about products existing in the system are expected to know their ontology. Currently, question what happens when a completely new type of products appears in the system for the first time remains open. However, this problem is outside of scope of this paper and will be addressed in the future.

3.2 Processing user request

Let us now discuss what happens in the system (where data is demarcated and stored in the above described way) when user request is processed. While the problem of interactions between an agent system and an "external" user turns out to be rather difficult to be solved in general, we have found an answer (for a complete discussion of the problem and the proposed solution see [13, 18]). Here we only assume that an HTML-based user interface is developed that allows her to specify the product she would like to purchase. Using methods similar to these described in [14, 18], a querystring describing the thought-after product, packed in an ACL *inform* message reaches the *CA*. Such a querystring could have the following form (if the user wanted to buy black athletic shoes size 36 or 37, with a price in a range 25 to 50 Euros):

```
?productClass=AthleticShoes
        &hasColor=BlackColor
        &prize:ofCurrency=EUR
        &prize:value:leftBound=25
        &prize:value:rightBound=50
        &size:value1=36
        &size:value1=37
```

Using product ontology, the *CA* translates the querystring into an SPARQL query [23]. Note that in the past [13] we have used RDQL as the query language. Currently we have changed our approach slightly and decided to utilize SPARQL instead. The main reasons for this change are (1) SPARQL is more expressive than RDQL; (2) SPARQL is about to obtain standardization (it is a product of a W3C working group and the specification is very close to completion), while in the case of RDQL there exist different implementations and there is no all-agreed test suite; (3) SPARQL query engine is better tested; and (4) what is very important for developers: JENA already includes working SPARQL module.

For the querystring represented above, the resulting SPARQL query has the following form:

```
PREFIX my:
  <http://jacs.ibspan.waw.pl/ontology#>
SELECT ?product, ?gateway
  { ?gateway :sells ?product; }
  { ?product, rdfs:subClassOf
                my:AthleticShoe ;
      my:hasColor my:BlackColor ;
      my:hasPrize ?prize ;
      my:hasSize ?size }
  { ?prize, my:ofCurrency my:EUR ;
      my:value ?prizeValue }
  { ?size  my:value ?sizeValue },
FILTER (
  ((?sizeValue = "36.0"
    || ?sizeValue = "37.0")
  &&
   (?prizeValue >= "25.0"
    && ?prizeValue <= "50.0"))
```

The *CA* sends an ACL message of type *query-ref* with the field :*contents* consisting of (FIPA SL language):

```
  ((
     all (sequence ?x ?y)
       (
         (and
            (Sells ?x ?y)
            (Matches-query
               ?y
               "<SPARQL-QUERY>"
            )
          )
        )
  ))
```

where <SPARQL-QUERY> denotes the query depicted above. The *CIC* receives the message and forwards it to be executed by one of *CIC DB* agents. The *CIC DB* agent queries the central database and obtains a set of stores that sell a given product. For instance, if the e-store recognized by the *GA* ga509@ibspan.waw.pl sells shoes that were the subject of the above presented query, its id (name and address) will be packed (together with other stores that sell shoes satisfying the query – the complete response-set) into the following ACL *inform* message:

```
((=
  (all (sequence ?x ?y)
         (and
           (Sells ?x ?y)
           (Matches-query
              ?y
              "<SPARQL-QUERY>"
           )
         )
  )
  (set
    (sequence
      (agent-identifier
        :name ga509@ibspan.waw.pl
        :addresses (sequence
          http://www.ibspan.waw.pl:9999)
      )
      (ProductDescription
          :owl "
            :Product596568431
               a :ShoeWithLaces ,
            :AthleticShoe ;
            :hasBrand "Nike";
            :hasColor :BlackColor ;
            :hasPrize
              [a :Prize ; :ofCurrency
                :EUR ; :value "33.0" ].
            :hasSize
              [a :EuropeanShoeSize ;
                :value "37.0"]."
                :hasSalingInfo
          [
            :isBuyerCreatedByGateway true;
            :isBuyerCreatedByClient false
          ].

      )
    )
  )
))
```

The *CA* will now process the obtained list according to its own criteria. For instance, it will eliminate e-shops that it dealt with in the past and that were found untrustworthy. While it is very interesting by itself (for more details see [15]), the question of selection of a group of shops out of the response-set obtained from the *CA* is out of scope of this paper. As a result, a list of shops that are to be contacted (actually their *GA*s) in an attempt to make a purchase will be created. Let us note that each product on the list will be serviced by a separate *BA*. This means that if, for some reasons, the *CA* wants to negotiate a pair of 36 size shoes and a pair of 37 size shoes (of exactly the same type) in the same shop, they will be serviced by two separate *BA*s. This matches our above mentioned assumption, that two products that differ even in a single characteristic are treated as separate products.

The response-set contains information which *GAs* accept incoming *BA*s, which create *BA*s and which service both possibilities. Depending on the offered possibilities and its own preferences, the *CA* either sends *BA*s or asks *GA*s to create *BA*s. Similarly to the described above process of *CA*s eliminating certain shops for not being trustworthy, the *GA* may not admit a *BA* representing a certain *CA* (or refuse to create a *BA* on its request) if it is deemed to be a spoiler (for instance it won multiple price negotiations but never finalized a purchase), see also [15].

Let us now assume that the *BA* has been admitted to the host. It informs the *GA* which product it is interested in purchasing (by sending an ACL *request* message containing product ID, e.g. 4094094049) and the GA pre-registers it as being interested in that product. At this stage there are two possible situations. If there is already a queue of *BA*s that are to negotiate this product then the *GA* provides the new *BA* with *negotiation protocol* and *template*. When the incoming *BA* is interested in a "new" product (no agent interested in it has been pre-registered or registered) the *GA* has to communicate with the *WA* as the information about the "form of price negotiation" to be used for a given product is stored in the *Shop Database* that is serviced by the *WA*. The *Shop Database* is used to manage inventory of products. Among others, it stores information about number of products that are available for sale, number of products that are currently reserved (as a result of earlier successful price negotiations), information about expiration time of each such reservation, and the current price negotiation mechanism (described in the *negotiation template* [1, 2]). Sample information about the current situation of black Nike athletic shoes with laces in size 37 and price 33 euro is depicted below:

```
:Product596568431
    :hasSalingInfo [
       :isBuyerCreatedByGateway true;
       :isBuyerCreatedByClient true;
       :usesTemplate
         :Product596568431Template;
       :totalQuantity: 10;
       :totalReservationsQuantity: 5;
       :hasReservation :Reservation5858;
       :hasReservation :Reservation2349
     ].

:Reservation5858
  :reservationQuantity: 3 .
:Reservation2349
  :reservationQuantity: 2 .

:Product596568431Template
  a :EnglishAuctionTemplate ;
  :maxBuyers "20" ;
  :startPrize
    [ a :Price ;
      :hasCurrency :EURO ;
      :hasValue "25.0"
    ] ;
  :TimesTerminationWindow "4:30".

:EnglishAuctionTemplate
    a owl:Class ;
    rdfs:subClassOf :Template ;
    rdfs:subClassOf
    [ a owl:Restriction ;
      owl:hasValue
        :EnglishAuction ;
      owl:onProperty
         :hasProtocol
    ] .
```

In the case of new product to be sold, the *GA* forwards the product ID to the *WA*. The *WA* queries the *Shop Database* and confirms that the requested product is still available (it is possible that all products have been reserved and currently there is none available for sale) and returns to the *GA* the current *negotiation template.* The *BA* is being thus served the *generic negotiation protocol* and the current *negotiation template* and requests the *strategy* from its *CA*. Upon reception of the *strategy,* the *BA* is ready to participate in price negotiations and notifies its *GA* accordingly (thus becoming registered as: awaiting for price negotiation to start). The *GA* acts also as negotiation manager. In this capacity it manages a pool of *Seller agents.* We have changed our original design and instead of creating a single *SeA* for each product sold, we have decided to proceed with product-agnostic *SeA*s that can service any price negotiation. When the time comes, the *GA* sends a list of *BA*s that have registered to negotiate given product to a free *SeA* (here we omit questions related to: which forms of negotiations require how many *BA*s? how long to wait before starting negotiations? how to handle template change? as they are outside the scope of this paper and have been addressed in [5, 6]. In addition it includes in the message the *negotiation template* (see [1, 2]) so that the *SeA* knows what negotiation form it is to manage and configures its rule-base according to it [3, 4, 7]. Furthermore, in the case when the *SeA* is actively involved in negotiations (e.g. in the case of Dutch Auction) it will also obtain its *strategy*. *Strategy* for the *SeA* is generated by the *SA* when it makes a decision that a given product will be negotiated using a mechanism that requires such *strategy* and stored in the *Shop DB*. The *SeA,* when ready, sends invitation to negotiations to all *BA*s and from this moment on, negotiations follow the scenario described in [1, 2].

Let us assume that negotiations were successful. Upon their completion, the *SeA* informs the winner *BA* about this fact and sends information to the *SA*. The *SA* makes a determination as to how long a reservation should last (e.g. for the first time buyer, reservation time may be "medium" to check her out; for a client with a spoiler-type reputation short time may be applied – to not to freeze uselessly an available product, while for a client in good standing an extended time reservation may be issued). This information is send by the *SA* to the winner *BA* as an ACL *inform* message.

The remaining parts of the scenario have been described in detail in [5, 6, 14] and they do not involve further extensive data manipulations. Let us now focus on the time management that has to take place in the system.

4. Time management in the system

Let us now look at the same processes as described above, but from the point of view of time management. The main problem that a system like ours has to address originates from the fact that it spans multiple computers and no assumption can be made about their local time. While one computer can be running with 13 seconds before the universal (GMT-based) time, a different computer can be running with 23 seconds behind the universal time. Currently, this is even more severe than in the past, since JADE 3.4 allows agents to travel between platforms (not only between multiple containers of the same platform that spans multiple computers). Thus, any mechanisms existing within distributed Java runtime, which could have been used in the past, cannot be utilized. Interestingly, this fact does not have much effect on the system as its proposed operation is completely asynchronous. However, there are two situations that are truly time sensitive. The first is the process of price negotiations. There exist a number of price negotiation mechanisms that use time explicitly (e.g. time to issue next bid – in English Auction, or time to deliver the bid – in most forms of closed bid actions). Fortunately, price negotiations take place locally, within a host. Due to the proposed model, no "long-distance" bidding takes place; *BA*s are either created within the host, or move and are admitted into it. This means, that it can be assumed that each price negotiation takes place on a single computer (while different price negotiations may take place on different machines within a single host) and all agents participating in price negotiations have access to the same time source. More specifically, all agents can issue a *System.currentTimeMillis*() call and as a result obtain local host time. Since this approach results in all agents obtaining "the same time," the negotiation process can utilize this mechanism in all cases which require time referencing.

The second time sensitive situation involves product reservations. This situation is more complicated as it involves multiple platforms and multiple computers – while price negotiations take place on one computer the *CA* is located on another. For example, if the *CA* sent out 13 *BA*s to 13 different computers, then we are dealing with 14 different clocks providing agents with 14 different local times. Let us now assume that some of these agents have succeeded in price negotiations and have received ACL messages informing them when their reservations expire. They forward this information to the *CA* that has to know precisely how much time does it have to decide whether to make a purchase before each of these reservations expires (thus making purchase impossible and, in addition, damaging *CA*'s reputation). One of possible solutions would be to specify length of reservation (e.g. "your reservation is valid for the next 115 seconds"), but this time references local clock and only the *BA* would be able to establish when the reservation actually expires (this would be unknown to the *CA*). The only solution that we were able to find is to reference the universal time. Here we use the SNTP protocol (Simple Network Time Protocol [20, 21, 22]) to establish the universal time and offset between that time and the local clock. Let us observe that since negotiations take place using the local time of the host, the only situation when the reference to universal time is required is when the *CA* has to establish when the reservation will actually expire. The first step to solve this problem will be to establish exact time of the host and each computer running a *CA*. Therefore, when each shop is initialized and local *GA* is created, it issues a call to the time server using SNTP protocol. Agent uses SNTP protocol to calculate *TimeOffset* on the basis of 4 different times: times of sending request by the client and receiving it by the server and times of sending response by the server and receiving it by the client. *TimeOffset* expresses the time difference between the local time and the universal time in milliseconds. We assume here that local time is only slowly deviating from the universal time and thus the procedure of checking the time difference has to be repeated infrequently; note however that it is also possible that the *GA* can check time before every price negotiation. We refer to the price negotiation as the moment when time will be checked and the time difference is passed to the *BA*s as an extension to the negotiation template. Therefore the negotiation template of an English Auction presented above, when passed to *BA*s that are to participate in it, will include also (offset is in milliseconds, as returned by the SNTP protocol):

```
:Product5965684319Template
  :gatewayTimeOffset "-4983982".
```

This time difference is then sent by the *BA* that succeeded in price negotiations together with the information when will the reservation expire (e.g. that the reservation expires at 12:37:45), to its *CA*. At the same time, similarly with how *GA* finds its *TimeOffset*, each *CA* establishes its *TimeOffset* upon its creation (and updates it as often as necessary). Obviously, upon reception of a message from one of its *BA*s, the *CA* can use both *TimeOffset*s to establish exactly when will the reservation expire within the host (e.g. if the host is "10 seconds behind" the universal time and the *CA*'s system is "5 seconds ahead", then the total *CA*'s offset to the host is 15 seconds ahead). This information is then used to establish when purchasing decisions have to be made to avoid expiration of reservations.

Obviously, we recognize that network lag will play a significant role in dealing with reservations that are about to expire (in case of slow network additional time has to be allocated to assure that the ACL message carrying the decision reaches a specific *BA* in time). It is the *CA* responsibility to estimate this lag and take it into account accordingly when deciding to finalize the purchase. While the *CA* may want to buy as much time a possible to make the optimal decision (e.g. to wait for all *BA*s to report), if such a decision is reached too late, then it will not be optimal after all.

5. Concluding remarks

In this paper we have presented solutions to information flow and time management in a model agent-based e-commerce system that is currently under development within our team. We considered a typical usage scenario and described the data requirements and associated flow of information needed to support it. In particular, discussion was focused on how ontologies – OWL, semantic query languages – SPARQL, agent communication messages – FIPA ACL and agent message content languages – FIPA SL can be successfully combined to achieve desired functionality for supporting typical user requests. Moreover, we have identified two time-sensitive situations that can occur in the system: process of price negotiations and precise length of product reservations. While for the first situation the proposed solution was quite straightforward and rather simple (based on using local host time), the second case was found to be significantly more complicated because it involves remote interactions between agents. Our proposed solution is based on referencing universal time via SNTP protocol to estimate time offsets and including an additional field in the negotiation template that represents e-shop time offset from universal time.

References

[1] Bartolini, C., Preist, C., Jennings, N. R.: A Software Framework for Automated Negotiation. In: *Proceedings of SELMAS'2004*, LNCS 3390, Springer Verlag, 2005, 213-235

[2] Bartolini, C., Preist, C., Jennings, N.R.: Architecting for Reuse: A Software Framework for Automated Negotiation. In: *Proceedings of AOSE'2002: International Workshop on Agent-Oriented Software Engineering*, Bologna, Italy. LNCS 2585, Springer Verlag, 2002, 88–100

[3] Badica, C., Badita, A., Ganzha, M., Iordache, A., Paprzycki, M:: Implementing rule-based mechanisms for agent-based price negotiations. In: *Proceedings of the 21st Annual ACM Symposium on Applied Computing, SAC'2006*. Dijon, France. ACM Press, ACM Press, New York, NY, 96-100, 2006

[4] Badica, C., Ganzha, M., Paprzycki, M.: Rule-Based Automated Price Negotiation: an Overview and an Experiment. In: *Proccedings of International Conference on Artificial Intelligence and Soft Computing, ICAISC'2006*, Zakopane, Poland. Lecture Notes in Artificial Intelligence, Springer-Verlag, 2006 (in print)

[5] Badica, C., Ganzha, M., Paprzycki, M. Mobile Agents in a Multi-Agent E-Commerce System. In: D. Zaharie et. al. (ed.) *Proceedings of the SYNASC 2005 Conference*. IEEE Press, Los Alamitos, CA, 2005, 207-214

[6] Badica, C., Ganzha, M., Paprzycki, M.: UML Models of Agents in a Multi-Agent E-Commerce System. In: *Proceedings of the ICEBE 2005 Conference*, IEEE Press, Los Alamitos, CA, 2005, 56-61

[7] Badica, C., Badita, A., Ganzha, M., Iordache, A., Paprzycki, M.: Rule-Based Framework for Automated Negotiation: Initial Implementation. In: Adi, A. Stoutenburg, S., Tabet, S.(eds.): *Proceedings of the Rules and Rule Markup Languages for the Semantic Web, First International Conference, RuleML 2005*, Galway, Ireland, 2005. LNCS 3791, Springer Verlag, 2005, 193-198

[8] Badica, C., Ganzha, M., Paprzycki, M.: Two Approaches to Code Mobility in an Agent-based E-commerce System. In: C. Ardil (ed.), *Enformatika*, Volume 7, 2005, 101-107

[9] Badica, C., Ganzha, M., Paprzycki, M., Pirvanescu, A.: Combining Rule-Based and Plug-in Components in Agents for Flexible Dynamic Negotiations. In: *Proceedings of CEEMAS'2005*, Budapest, Hungary. LNAI 3690, Springer Verlag, 2005, 555-558

[10] Chmiel K., Tomiak D., Gawinecki M., Kaczmarek P., Paprzycki M., Szymczak M.: Testing the Efficiency of JADE Agent Platform. In: *Proceedings of the International Symposium on Parallel and Distributed Computing, ISPDC 2004 Conference*, Cork, Ireland. IEEE Press, Los Alamitos, CA, 2004, 49-57

[11] Fensel, D.: *Ontologies: Silver Bullet for Knowledge Management and Electronic Commerce*. Springer-Verlag, Berlin, 2001

[12] FIPA SL Content Language Specification, 2002, http://www.fipa.org/specs/fipa00008/SC00008I.html.

[13] Gawinecki, M., Gordon, M., Kaczmarek, P., Paprzycki, M.: The Problem of Agent-Client Communication on the Internet. *Scalable Computing: Practice and Experience*, 6(1), 2003, 111-123

[14] Ghanza, M., Paprzycki M., Pirvanescu, A., Badica, C., Abraham, A.: JADE-Based Multi-Agent E-Commerce Environment; Initial Implementation. *Annals of West University Seria Matematica-Informatica*, Vol.XLII, 2004, 79–100

[15] M. Ganzha, M. Gawinecki, P. Kobzdej, M. Paprzycki, C. Badica, Towards trust management in an agent-based e-commerce system – initial considerations (submitted for publication)

[16] http://jade.tilab.com/

[17] Jena 2 - A Semantic Web Framework, Hewlett Packard, http://www.hpl.hp.com/semweb/jena2.htm

[18] Kaczmarek, P.: *Multimodal Communication Between Users and Software Agents*. Masters Thesis, AMU, 2005

[19] OWL - Web Ontology Language Overview, http://www.w3.org/TR/owl-features/

[20] http://www.ietf.org/rfc/rfc1361.txt

[21] http://www.ietf.org/rfc/rfc1769.txt

[22] http://www.ietf.org/rfc/rfc2030.txt

[23] SPARQL Query Language for RDF, http://www.w3.org/TR/rdf-sparql-query/