Mobile Agents in a Multi-Agent E-Commerce System

Costin Bădică
Software Engineering Department
University of Craiova
Bvd. Decebal 107, Craiova,
200440 Romania
c_badica@hotmail.com

Maria Ganzha
Department of Administration
Elbląg University of Humanities and Economy
ul. Lotnicza 2
82-300 Elbląg, Poland
ganzha@op.pl

Marcin Paprzycki
Computer Science, OSU
Tulsa, OK, 74106, USA
and
Computer Science, SWPS
03-815 Warsaw, Poland
marcin@cs.okstate.edu

Abstract

Among features often attributed to software agents are autonomy *and* mobility. Autonomy *of e-commerce agents involves* adaptability *to engage in negotiations governed by mechanisms not known in advance, while their* mobility *entails such negotiations taking place at remote locations. This paper aims at combining* adaptability *with* mobility*, by joining rule-based mechanism representation with modular agent design, and at UML-formalizing selected aspects of the resulting system. Furthermore, we discuss the issue of agent mobility and argue why such agents have been proposed for the system under consideration.*

1 Introduction

Current agent environments (e.g. JADE [3]) offer sufficient support for implementing quasi-realistic models of e-commerce. Moreover, advances in auction theory have produced a general methodology of describing price negotiations [11]. Combination of these factors gave new impetus to research on automating e-commerce, and one of suggested approaches was utilization of *autonomous*, *mobile* software agents. Since *autonomy* is a very broad concept, we would like to focus on *adaptability* that can be viewed as ability to update the negotiation "mechanism" to engage in unknown in advance forms of negotiations [9]. Obviously, another aspect of *autonomy* is *decision autonomy* and involves "intelligence" that can be understood as capability to reason over past experiences and domain knowledge in order to maximize utility. Note that such reasoning possibly involves complex (resource consuming) numerical and/or symbolic computations. This aspect of *autonomy*, while important, appears in this paper only as a "price" that is to be paid (in terms of resource utilization) in the case when mobile agents are to be employed. Agent *mobility*, on the other hand, enables application components that require strong interactions either between themselves (e.g. in automated negotiations) or with a continuously evolving information source (e.g. the stock market), to migrate close to each other (e.g. to be collocated on the same server) or close to that information source. It should be obvious, that mobile agents have to be lightweight to be able to swiftly move across the network. At the same time, intelligent agents cannot be lightweight as they have to "carry" their intelligence with them. This makes intelligent mobile agents a clear case for the no-free-lunch theorem [10].

In this paper we describe an architecture of a multi-agent e-commerce system that aims at combining *adaptability*, *mobility* and *intelligence*. In our system, *autonomous* agents are engaged in matchmaking, negotiations and contracting (including actually "purchasing" sought after products) on behalf of their users: humans or businesses. Our proposal builds on: (i) conceptual architecture of a multi-agent e-commerce system described in [6] (see also references to our earlier work collected there), (ii) flexible framework that allows agents to participate in arbitrary negotiations introduced in [1, 2], and (iii) lightweight agents migrating to remote markets and engaging in "any" form of negotiations via dynamically loadable modules [6]. Furthermore, we proceed beyond what is typically considered in the agent-literature: the "act" of price negotiation itself. While in [8] negotiations were extended to include matchmaking, in our work we conceptualize the negotiation as a part of a more complete e-commerce scenario consisting of: requesting purchase, matchmaking, negotiating and completing purchase. Interestingly, the final stage of an e-commerce scenario: between completion of price negotiations and actual purchase, while involving a number of interesting possibilities, is practically forgotten in the literature.

We proceed as follows. In the next section outline the negotiation framework introduced in [1, 2] and the modular agents presented in [6] as well as how we combine them. We follow with a general description of agents appearing in the system and their basic interactions (Section 3) and a detailed presentation and UML formalization of selected agents appearing in our system (Section 4). Section 5 discusses the reasons why use of mobile agents is "optimal" within the proposed system.

2 Components for Automated Negotiations

Authors of [1, 2] analyzed the existing approaches to agent negotiations (primarily the FIPA-standardized auction protocols) and argued that they do not provide enough structure for the development of portable agent-based e-commerce systems. They also outlined a new agent negotiations framework, consisting of *buyers* and a *host* where the negotiations take place. Within the host, the infrastructure for negotiations was provided through a number of sub-agents: *Gatekeeper*, *Proposal Validator*, *Protocol Enforcer*, *Information Updater*, *Negotiation Terminator* and *Agreement Maker* that interact with each-other by direct messaging and via a blackboard. The central point of their framework consisted of a generic negotiation protocol and taxonomy of JESS [4] rules used for enforcing specific negotiation mechanisms.

Independently, in our earlier work, we have followed proposal outlined in [9] and implemented an agent-based e-commerce skeleton, where agents were capable of negotiation *adaptation* via dynamically loadable modules [6]. Negotiating agents consisted of three main components: (i) *communication module* – responsible for messages exchanged between agents (part of the skeleton moving across the network), ii) *protocol module* – responsible for enforcing the (FIPA) protocol that governed negotiations (public and downloadable form any server), and (iii) *strategy module* – responsible for producing protocol-compliant actions necessary to achieve agent goals (private and downloadable from the home server).

Let us make three observations that allow us to combine and extend the two approaches (due to the lack of space, [6] and references presented there should be consulted for further details):

- The framework introduced in [1, 2] assumes implicitly that *Buyer* agents are mobile and carry with them the "generic negotiation protocol" thus making them rather heavy; obviously, our approach based on pluggable negotiation modules can be employed here to achieve lightweight mobility.

- The *Gatekeeper* sub-agent does not play any role in actual price negotiations; it only allows buyers into the negotiation space and provides them with the negotiation protocol and the negotiation template; thus we have removed it from the "negotiation infrastructure" and placed in the system as a full-fledged agent (and have added to it a number of additional "managerial functions," see below).

- Analysis presented in [1, 2] involves only *Buyer* agents visiting a given host to make a purchase and becoming involved in price negotiations; actions of the system preceding and following negotiations are not considered; on the other hand these functions were a part of our original system and have been further extended for the current paper.

3 Agents in an E-Commerce Environment

Let us now briefly discuss architecture of the proposed multi-agent e-commerce environment. Fundamentally, our system represents a distributed marketplace that hosts e-stores and allows e-clients to "visit" them to purchase products. Figure 1 introduces agents occurring in our system and specifies which agents are in direct contact (solid arrows denote communication; dashed arrow denotes agent movement; rectangular boxes surround buyer and seller systems and agents populating them). The only agent not represented there is an auxiliary *CICDB* agent that interfaces the *CIC* agent with the database of agents and products. *Shop*

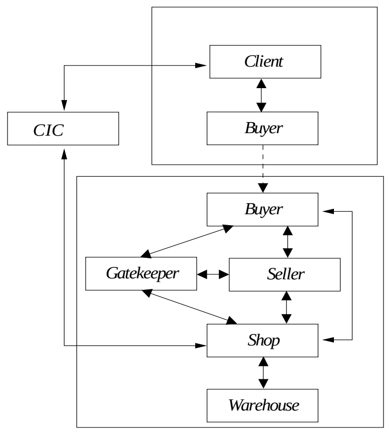

Figure 1. Agents in the e-commerce environment

and the *Client* agents represent system "users" — merchants that sell products and buyers who want to purchase them. After being created, during system initialization, *Client* agents register with the *CIC* agent and await user requests. Similarly, *Shop* agents create *Gatekeeper* agents and register them with the *CIC* agent. Furthermore, each *Shop* agent registers there all products that are available in its e-store (an initial list of available products). In this way, the *CIC* agent combines the function of white pages, by storing information (addresses and identifiers) about all *Shop*, *Gatekeeper* and *Client* agents existing in the system, and of yellow pages, by storing information about available products (see also [8]). Due to its auxiliary role, in this paper we omit the *CIC* agent from further considerations (for more information see [6] and papers referenced there).

On the supply side, *Shop* agents are responsible for creating *Seller* agents (one for each product to be sold). *Seller* agents implemented in our system embody the same functionalities as the *Negotiation host* agents described in [1, 2]. Specifically, except of the *Gatekeeper* agent that is not part of negotiations and has a more complex role, *Seller* agents consist of all sub-agents identified there; each playing exactly the same role. Finally, each *Seller* is provided (by the *Shop* agent) with its initial negotiation template.

Client agents utilize mobile *Buyer* agents to seek the best possible offer matching user-specified requirements. *Buyer* agents engage in price negotiations and report results to the *Client* agent, which gathers offers and decides where from to attempt at making an actual purchase (or decides that purchase that would satisfy its user cannot be made).

Finally, in addition to agents proposed in [6], we extend our system and include *Warehouse* agents, also created by *Shop* agents during system initialization (one such agent for every e-store in the system). *Warehouse* agents are responsible for: (i) management of products available for sale, (ii) management of reserved products, and (iii) in the future, interfacing with an agent-based supply chain management sub-system.

Let us now systematically describe each agent and its function. Due to the lack of space we present complete UML diagrams only of two most important agents in the context of agent mobility: the *Gatekeeper* and the *Buyer*.

4 Agents in the System

Shop Agent The *Shop* agent acts as the representative of the "user" (seller). We assume that after it is created it persistently exists in the system until the user decides that it is no longer needed (in practice, it can be assumed that it exists throughout the run of the system). Upon its creation, the *Shop* agent creates and initializes a *Gatekeeper* agent, a *Warehouse* agent and *Seller* agents (one for each category of product sold). The initialization of the *Warehouse* agent involves passing information about goods that are initially available for sale, while initialization of the *Gatekeeper* and *Seller* agents involves providing them with templates that are to be used initially in price negotiations. Furthermore, the *Gatekeeper* agent and the list of available products are registered with the *CIC* agent.

After initialization, the *Shop* agent enters a complex state where it supervises negotiations and the product flow. Within one execution thread it awaits finish of price negotiations. If they were successful, supervising *Seller* informs the *Shop* agent, which asks the *Warehouse* agent to reserve a given quantity of a particular product (for a specific amount of time). The events can then proceed according to three different scenarios. *Case 1:* if the winning *Buyer* confirms purchase the *Shop* asks the *Warehouse* agent to check the reservation. If the reservation did not expire then the *Shop* informs the *Buyer* agent about acceptance of transaction. This event starts the final stage — named "Sale finalization" which includes such actions as payment and delivery. *Case 2:* if the reservation has expired, then the *Shop* agent sends a rejection message to the *Buyer* agent. *Case 3:* if the *Client* agent rejects purchase (and informs the *Shop* agent about it through the *Buyer* agent) the *Shop* agent asks the *Warehouse* agent to cancel the reservation. Completing one of these three cases "closes" this branch of *Shop* agent execution.

Separately, the *Shop* agent keeps track of all negotiations and transactions and periodically performs multicriterial analysis (start of the *MCDM* — multicriterial decision making — module is a part of initialization of the *Shop* agent) that may result in changes in the negotiation template (e.g. minimal price, type of price negotiation mechanism, etc.). For instance, when only a few items are left they may be deeply-discounted, or put on sale through an auction. In this case a new template is generated and sent to the *Gatekeeper* agent that switches it in an appropriate moment (see below, Figures 2,3).

Gatekeeper agent *Shop* agents cooperate directly with *Gatekeeper* agents that (1) interact with *Buyer* agents: admit them to the negotiations and provide with the protocol and the current negotiation template, (2) in suitable moments release *Buyer* agents to appropriate *Seller*s and (3) manage updates of templates. The statechart diagram of the *Gatekeeper* agent is presented in Figure 2 (the top level depiction of *Gatekeeper* functionality) and continued in Figure 3 (detailing negotiation preparations). When an appropriate number of *Buyer* agents have registered, the *Gatekeeper* passes their identifiers to the *Seller* agent and thus allows the negotiation to start. As soon as this step is completed, the *Gatekeeper* cleans the list of registered *Buyer* agents and the admission/monitor process is restarted (assuming that the *Seller* agent is still alive). Note that the *Gatekeeper* admits to negotiations only "complete" *Buyer* agents; such agents that have all modules installed and confirmed that are ready (as indicated by the small loop above the "Waiting for the Buyer msg" box).

When a new template module is delivered by the *Shop* agent a list of currently registered *Buyer* agents is put into a buffer ("Buffer registration list" box). These agents have to be serviced first, using the current template that they have been provided with upon entering the e-store. Since

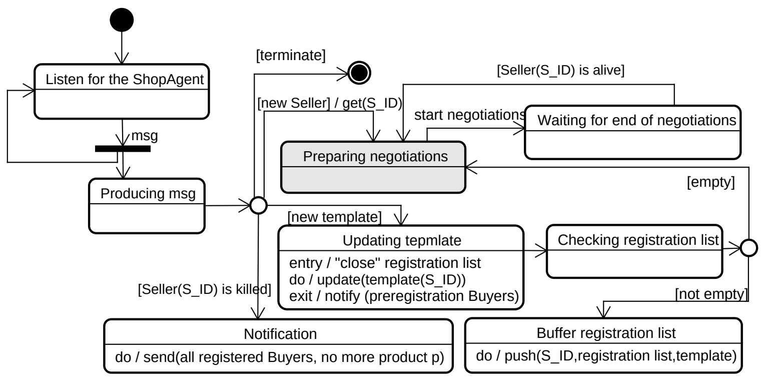
Listen for the ShopAgent
msg
Producing msg
[terminate]
[new Seller] / get(S_ID)
[Seller(S_ID) is alive]
start negotiations
Waiting for end of negotiations
Preparing negotiations
[empty]
[new template]
Updating tepmlate
entry / "close" registration list
do / update(template(S_ID))
exit / notify (preregistration Buyers)
Checking registration list
[Seller(S_ID) is killed]
[not empty]
Notification
do / send(all registered Buyers, no more product p)
Buffer registration list
do / push(S_ID,registration list,template)

Figure 2. UML statechart diagram of the Gatekeeper agent

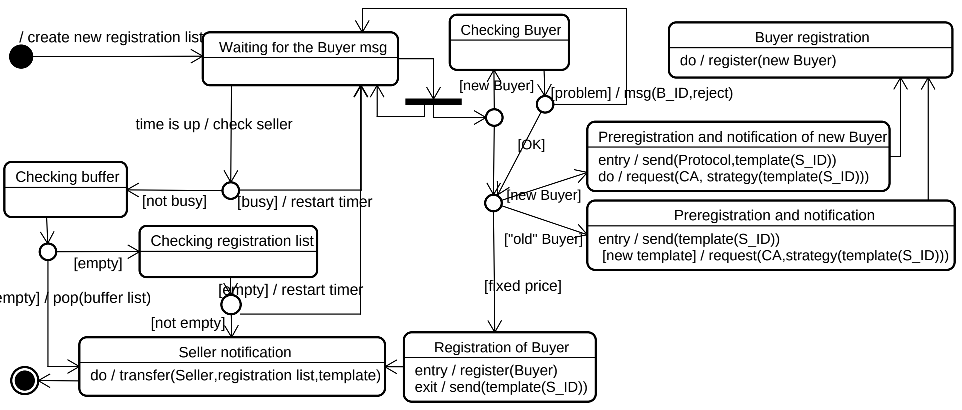
/ create new registration list
Waiting for the Buyer msg
Checking Buyer
Buyer registration
do / register(new Buyer)
[new Buyer]
[problem] / msg(B_ID,reject)
time is up / check seller
[OK]
Preregistration and notification of new Buyer
entry / send(Protocol,template(S_ID))
do / request(CA, strategy(template(S_ID)))
Checking buffer
[not busy]
[busy] / restart timer
[new Buyer]
Preregistration and notification
entry / send(template(S_ID))
[new template] / request(CA,strategy(template(S_ID)))
["old" Buyer]
Checking registration list
[empty]
[empty] / restart timer
[fixed price]
[not empty] / pop(buffer list)
[not empty]
Seller notification
do / transfer(Seller,registration list,template)
Registration of Buyer
entry / register(Buyer)
exit / send(template(S_ID))

Figure 3. Statechart diagram for Preparing Negotiations State

then, the new incoming agents will be provided with the new template. Let us note, that here we have to deal with one more case: when *Buyer* agents have already received negotiation protocol and template, but have not received (from their *Client* agent) their negotiation strategy. In this case, when the change of template takes place, the *Gatekeeper* agent (even before buffering *Buyer* agents that are ready and waiting for start of negotiations) informs such incomplete agents that there is a change in the negotiation template and provides them with the new one. As a result, *Buyer* agents have to re-request negotiation strategy (see Figure 4, loop “new template”). Note that our system allows loosing *Buyer* agents to stay at the host and re-enter negotiations after updating protocol templates (the “old Buyer” path). Finally, in a special case when a given product has been sold-off and will not be replenished, the *Shop* agent terminates the *Seller* responsible for selling it and the *Gatekeeper* informs awaiting *Buyer* agents about this fact.

Warehouse Agent The *Shop* agent interacts directly also with the *Warehouse* agent. In the early stages of its functioning the *Warehouse* agent is supplied (through messages from the *Shop* agent) with information about products and their quantities (to be saved into a database). Then the *Warehouse* agent enters a complex state where it (a) awaits notifications from the *Shop* agent and (b) manages time triggered events. The *Shop* agent notifies the *Warehouse* agent about: (i) registration of new products for sale, (ii) product reservations, (iii) purchase confirmations, and (iv) purchase terminations. The time event triggers checking existing reservations. If reserved products whose reservation time has expired are found, these reservations are canceled, reserved products are added to the pool of products available for sale and the *Shop* agent is informed about a new amount of available commodities. Note that the information about cancelled reservation is provided to the *Shop* agent only when a purchase is requested by the *Buyer* agent and the *Shop* agent is checking if a transaction can be completed. Finally, if quantity of some product becomes 0, the *Warehouse* agent informs about it the *Shop* agent, which may decide to terminate the corresponding *Seller* agent. In this case it informs about it both the *CIC* and the *Gatekeeper* agents.

Seller agent Finally, the last agent working on the supply

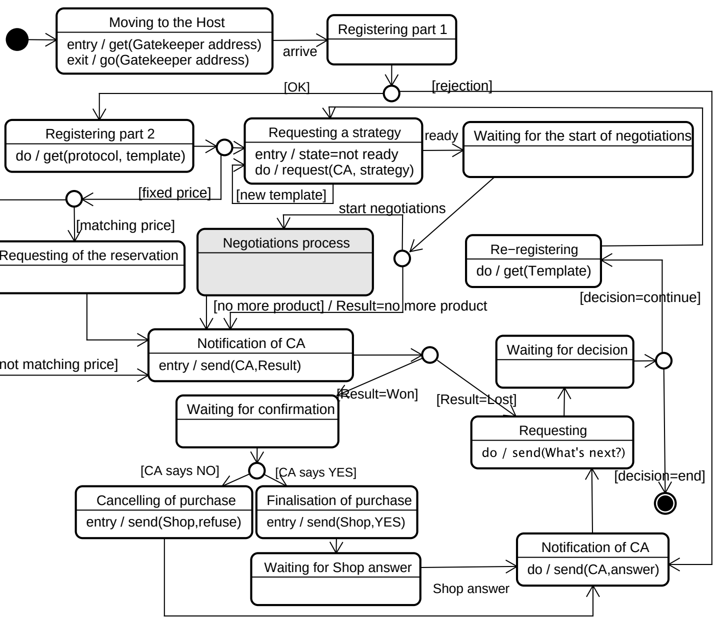

Figure 4. UML statechart diagram of the Buyer agent

side of the system is what appear to be a simple *Seller* agent. Its apparent simplicity comes form the fact that it encompasses the complete negotiation framework proposed in [1, 2]. Note that not all negotiations have to end by establishing a winner and our system is able to handle such an event. At the same time, all data about negotiations is send to the *Shop* agent that collects and analyses them. Thus, for instance, a sequence of failures should result in a change in template resulting form the multicriterial analysis that it performs.

Client agent On the purchasing side, we have two agents. The *Client* agent exists in a complex state. On the one hand it listens for orders from the customer and, to fulfill them: (1) queries the *CIC* agent which stores sell the requested product, (2) then dispatches *Buyer* agents to each such e-store (identified by its *Gatekeeper* agent). On the other hand, it directly manages the process of making purchases on behalf of the customer, on the basis of *Buyer* notifications informing about the results of price negotiations.

For a specific amount of time the *Client* collects reports (messages) sent by *Buyer* agents (each *Buyer* agent report is also stored in a database for further information extraction). When the wait-time is over (or when all *Buyer* agents have reported back) the *Client* agent enters a complex state. On the one hand it continues listening for messages from *Buyer* agents (obviously if all have reported then no messages will be coming). On the other hand it goes through a multicriterial decision procedure (*MCDM*) that has one of three possible outcomes: (i) to attempt at completing a selected purchase, (ii) to cancel the existing reservations and request that *Buyer* agents re-engage in price negotiations — thus awaiting a better opportunity, or (iii) to declare the purchase impossible and notify the customer accordingly. Note that, in a realistic system, the *MCDM* analysis should be truly multicriterial and include factors such as: price, history of dealing with a given e-shop, delivery conditions etc.

When the attempt at completing a purchase is successful, then the *Client* agent sends messages to all *Buyer* agents ordering them to self-destruct. The situation is slightly more complicated when the attempt was unsuccessful and purchase was not deemed impossible. Then the *Client* agent undertakes the following actions: (1) informs all *Buyer* agents that have already reported to cancel current reservations and return to price negotiations and (2) resets timer establishing how long it will wait before the next *MCDM* analysis. Observe that it is possible that the first *MCDM* analysis was undertaken before all *Buyer* agents have complete their “first round” of price negotiations. They could have contacted the *Client* while it was “thinking” which of the existing offers to choose. In this way, some agents make their second attempt at negotiating prices, while some agents have just finished the first. As this process continues in an asynchronous fashion various *Buyer* agents will make different number of attempts at negotiating price that is acceptable to the *Client* agent. This process will terminate when all orders submitted by the customer have been either honored or abandoned.

Buyer agent Finally, the *Buyer* agent (see Figure 4) is the only mobile agent in the system. It is dispatched by the *Client* agent to all stores that carry product desired by the customer. It arrives at the e-store (identified by the address of the *Gatekeeper* agent) and communicates with the *Gatekeeper* agent (see Figure 2,3) to obtain entry to the negotiations (in case when entry is not granted it informs its *Client* agent and is ordered to self-destruct). When entry is granted the *Buyer* obtains, from the *Gatekeeper*, the negotiation protocol and the current negotiation template. In the next step, *Buyer* agent requests and obtains an appropriate strategy module from the *Client* agent (see also the above description of the case when the negotiation template changes while the *Buyer* agent awaits the strategy module). When all three modules are installed, *Buyer* informs the *Gatekeeper* that it is ready and when prompted proceeds to negotiate with an appropriate *Seller*; note the special treatment of fixed-price negotiations by both the *Buyer* and the *Gatekeeper* agents. Upon completion of negotiations, *Buyer* informs the *Client* about their result and, if necessary (when an attempt at completing purchase is made), acts as an intermediary between the *Client* and the *Shop*. In the case when purchase was not attempted or was not successful, *Buyer* agent awaits the decision of the *Client* and if requested proceeds back to participate in price negotiations (before doing so it updates its negotiation template and the strategy module). This process continues until the *Buyer* agent is killed on the request of the *Client* agent.

Let us emphasize that in the proposed system we observe two types adaptability. First, the “negotiations mechanism” adaptability—where *Buyer* agents adapt their behavior to that expected in the e-store. This mechanism has been already implemented (see [6]). There are also other places where adaptability materializes: (1) *Shop* agent adjusting negotiation templates (e.g. mechanism or minimal price) based on the flow of products, (2) *Shop* agent adjusting the negotiation strategy for each product category, (3) *Client* agent creating specific negotiation strategies (in its *MCDM* “subsystem”) that depend, among others, on the negotiation mechanism, product and particular store (responding to that e-store strategy), (4) *Client* agent utilizes its *MCDM* module to select which store to make a purchase from. That *MCDM* process is adaptable in two senses: (i) it depends on historical data (e.g. past interactions with e-stores), and (ii) it represents the user on behalf of which the *Client* acts and thus adjusts to. These forms of adaptability are currently being considered in our research.

Finally, to summarize what has been presented thus far, in Figures 5, 6, 7, we present the complete flow of actions in the system.

5 Agent mobility

Let us now devote our attention to the question that is constantly being discussed in the agent community: “Why should one use *mobile* agents instead of messaging?” It could be (and it has been) argued that agent *mobility* is unnecessary. We have also seen papers, where agent mobility was clearly spurious in the context of the problem in question. At the same time, we believe that in the system under consideration agent mobility plays an important role.

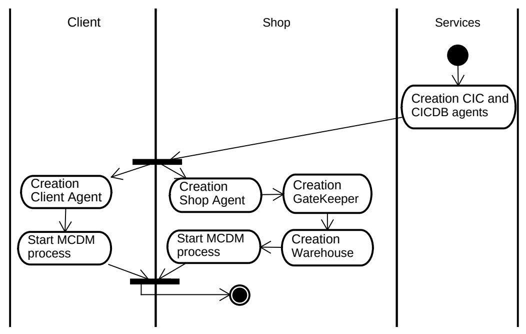

Figure 5. UML activity diagram of the initialization of the system

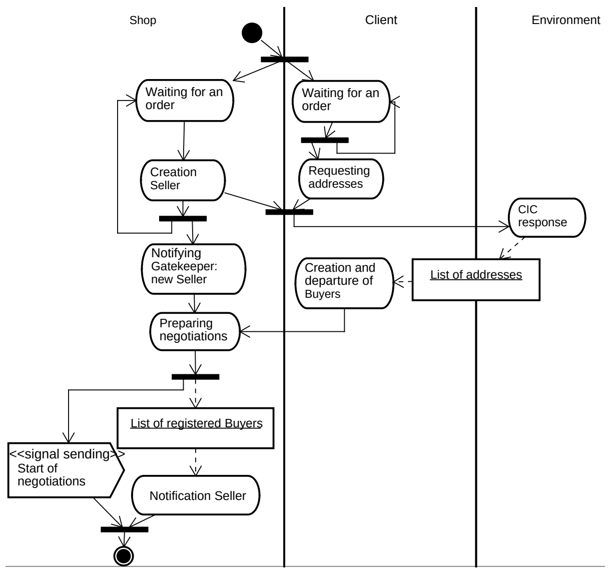

Figure 6. UML activity diagram of actions taking place during preparations to negotiations

Let us start by considering someone who, sitting behind a slow Internet connection (which is not an uncommon situation), tries to participate in an eBay auction. In this case it is almost impossible to be sure that ones bid (1) reaches eBay server in time, (2) is sufficiently large to outbid opponents that have been bidding while connected over a fast link (information about auction progress as well as our responses may not be able to reach their destinations sufficiently fast). Here, network-caused delays can be signifi-

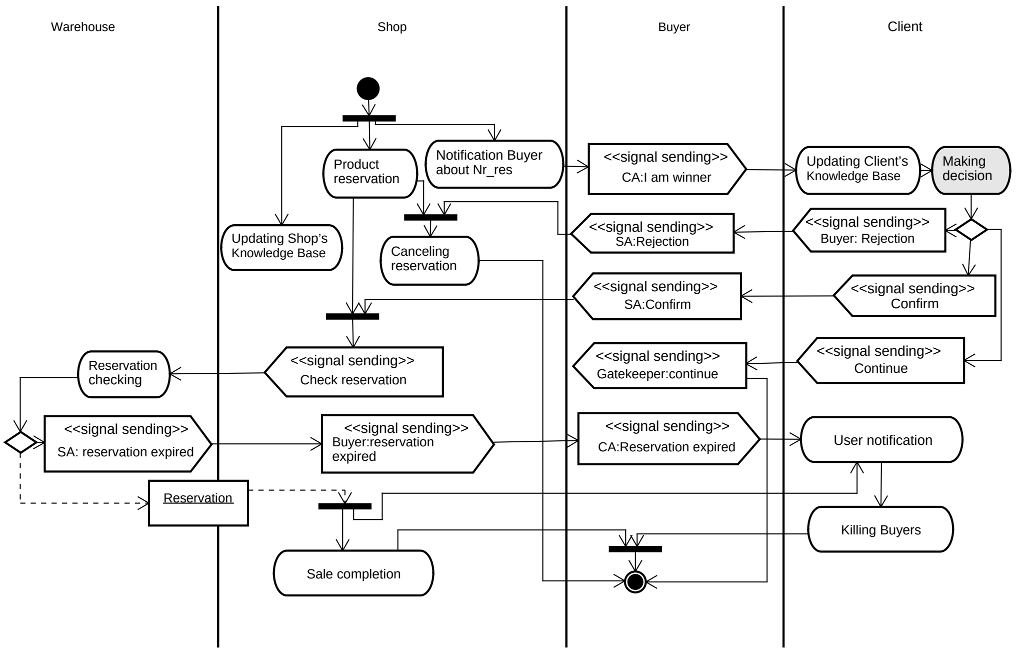
Warehouse
Shop
Buyer
Client
Product reservation
Notification Buyer about Nr_res
<<signal sending>> CA:I am winner
Updating Client's Knowledge Base
Making decision
Updating Shop's Knowledge Base
Canceling reservation
<<signal sending>> SA:Rejection
<<signal sending>> Buyer: Rejection
<<signal sending>> SA:Confirm
<<signal sending>> Confirm
Reservation checking
<<signal sending>> Check reservation
<<signal sending>> Gatekeeper:continue
<<signal sending>> Continue
<<signal sending>> SA: reservation expired
<<signal sending>> Buyer:reservation expired
<<signal sending>> CA:Reservation expired
User notification
Reservation
Killing Buyers
Sale completion

Figure 7. UML activity diagram of actions taking place after negotiations are complete

cant for the outcome of negotiations (purchase of the desired product may be prevented). Obviously, problem described here can be avoided if an agent representing user is located at the same server where the negotiations take place. Let us stress, that we take very seriously the notion of agent *autonomy*. Thus we assume that an agent that represents its user is capable of autonomously completing the requested task (purchase chosen product(s)).

Assuming that mobile agents are to be used to move negotiating agents within the auction-carrying server; resulting in offers being made fast enough to efficiently participate in price negotiations, one can ask about the price that is to be paid for moving autonomous agents across the network. Obviously, it is possible that an agent may not be able to participate in a particular auction because it does not reach the auction server in time. Our response comes in four parts. (1) If it is a particular auction that the user is interested in, then agent not reaching the auction server has exactly the same effect as not being able to win because of bid(s) being late. (2) Therefore, it is only an agent that reaches the server in time that gives its user any chance to effectively participate in price negotiations. (3) Furthermore, if an agent reaches its destination, it will be able to effectively participate in all upcoming negotiations within that server, while delays caused by network traffic may permanently prevent user from such effective negotiations. (4) Finally, it is exactly the proposal presented in this paper (dynamically loadable negotiation modules) that reduces the load that is to be transported over the network and thus attempts at addressing at least a part of this problem.

Separately, one may suggest that a very large number of agents visiting an e-shop and computing their bidding strategies in real-time may result in a substantial usage of local resources. This, in turn, may constitute an argument against agent mobility (why should an e-store supply all these resources?). In response, let us consider economical forces driving e-commerce. Here, we have buyers who want to purchase certain goods and sellers who want to sell them. Obviously, it is the sellers, who benefit financially from selling products and thus have to do “whatever it takes” to satisfy their customers. Therefore, it is likely to be enough that one e-shop provides infrastructure that will be robust enough for buyer-agents to use it successfully, and the remaining e-shops will have to follow. This is just another version of what is happening in old-commerce. If a given store is too small and/or inconvenient to handle 100 clients at a time then they are likely go to the larger store and the inconvenient store will loose clients. Furthermore, let us note that the proposal put forward in our research is also capable of addressing some aspects of this problem. Assuming that agents are to be mobile and they are to visit e-shops (as argued above) then loading only necessary modules produces agents that are as slim as possible. In this way, agents advocated here result in minimal resource (e.g. computer memory) consumption.

Let us now consider one more possibility, that *Buyer* agents are assembled completely or partially before they are send to the e-store. Obviously, since both the negotiation protocol and template can be obtained within the e-store, carrying them across the network makes no sense. Therefore, maybe it would be possible to send *Buyer* agents with the negotiation strategy module already loaded? The main problem with this proposal is a result of our assumption that e-stores can respond to the flow of commodities by actively changing their negotiation templates. This being the case, by the time that the 1/3 assembled *Buyer* reaches its destination, its strategy module may be already outdated and the first thing it would have to do, would be to request a new one (resulting, among others, in extra messages being exchanged). Therefore, we have elected to send across the network only a minimal agent-skeleton and outfit it with appropriate module within the e-store. Note that this approach considerably simplifies the overall system design.

Finally, let us consider one more problem that is related to agent mobility. Let us recall that *Buyer* agents are relatively simplistic and it is the *Client* agent that makes the final determination where to attempt at making the purchase. Therefore, *Client* agents have to communicate selected (remotely located) *Buyer* agents and their request to complete purchase may be network-delayed, resulting in an expired reservation and inability to complete the task. Unfortunately, this problem does not seem to have a simple solution, since offer comparison requires communication between agents participating in price negotiations (i.e. in our system we have selected a central point — *Client* agent — that will collect all offers, instead of all-to-all communication, but the same problem haunts all possible approaches to finding one offer among many). Furthermore, since not all sites can be expected to conduct their price negotiations at the same time, and with the same "urgency," it is simply impossible to assure that the best offer will still be available, when the "remaining" agents complete their negotiations. Therefore, our solution remains optimal in terms of reducing total network congestion by sending only minimal-size agents and minimizing the total number of messages send over the network.

6 Concluding Remarks

In this paper we have discussed a multi-agent e-commerce system that combines rule-based and mobile agent technologies for implementing flexible automated negotiations. After presenting an overview of the proposed system and UML diagrams of two of its agents as well as a complete action-diagram, we have focused our attention on questions involved in agent mobility. We have argued that *agent mobility is the most optimal solution* for the e-commerce model considered here. Then we have shown why it can be expected that in the future e-stores will provide an infrastructure robust enough for mobile agents to frequent them and negotiate prices. We have followed by arguments why the proposed solution, based on dynamically loadable modules, helps reduce auction-server resource utilization and why *Buyer* agents should not be assembled before they reach their destination. Finally we have discussed why there is no simple solution to the problem of finding the optimal offer when multiple agents negotiate prices within multiple e-stores and thus why our solution is as optimal as any other.

System described here is currently being re-implemented using JADE and JESS toolkits [3, 4] (the previous version of the system, while fully functional [6], did not involve the general framework introduced in [1, 2]). We will report on our progress in subsequent papers.

References

[1] 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.

[2] 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.

[3] JADE: Java Agent Development Framework. See `http://jade.cselt.it`.

[4] JESS: Java Expert System Shell. See `http://herzberg.ca.sandia.gov/jess/`.

[5] Fuggetta, A., Picco, G.P., Vigna, G.: Understanding Code Mobility. In: IEEE Transactions on Software Engineering, vol.24, no.5, IEEE Computer Science Press (1998) 342–361.

[6] Maria Ganzha, Marcin Paprzycki, Amalia Pîrvănescu, Costin Bădică, Ajith Abraham (2005) JADE-based Multi-agent E-commerce Environment: Initial Implementation, Analele Universităţii din Timişoara, Seria Matematică–Informatică(to appear)

[7] Tamma, V., Wooldridge, M., Dickinson, I: An Ontology Based Approach to Automated Negotiation. In: *Proceedings AMEC'02: Agent Mediated Electronic Commerce*, LNAI 2531, Springer-Verlag (2002) 219–237.

[8] Trastour, D., Bartolini, C., Preist, C.: Semantic Web Support for the Business-to-Business E-Commerce Lifecycle. In: Proceedings of the WWW'02: International World Wide Web Conference, Hawaii, USA, ACM Press, New York, USA (2002) 89–98.

[9] Tu, M.T., Griffel, F., Merz, M., Lamersdorf, W.: A Plug-in Architecture Providing Dynamic Negotiation Capabilities for Mobile Agents. In: *Proceedings MA'98: Mobile Agents*, Stuttgart, Germany, (1998) 222–236.

[10] Wooldridge, M.: *An Introduction to MultiAgent Systems*, John Wiley & Sons, (2002).

[11] Wurman, P, Wellman, M., Walsh W.: A Parameterization of the Auction Design Space. In: *Games and Economic Behavior*, 35, Vol. 1/2 (2001) 271–303.