BarterCell: An Agent-based Bartering Service for Users of Pocket Computing Devices

Sameh Abdalla, David Swords, Anara Sandygulova, Gregory M.P. O'Hare and Paolo Giorgini

University College Dublin (UCD)
Belfield, Dublin 4, Ireland.
And, University of Trento, Italy.
{anara.sandygulova, david.swords}@ucdconnect.ie
{sameh, gregory.ohare}@ucd.ie
paolo.giorgini@unitn.it

Abstract. The rising integration of pocket computing devices in our daily life duties has taken the attention of researchers from different scientific backgrounds. Today's amount of software applications bringing together advanced mobile services and literature of Artificial Intelligence (AI) is quite remarkable and worth investigating. In our research, software agents of BarterCell can operate in wireless networks on behalf of nomadic users, cooperate to resolve complex tasks and negotiate to reach mutually beneficial bartering agreements. In this paper, we introduce BarterCell that is an agent-based service application for users of pocket computing devices. We introduce new negotiation algorithms dedicated to bartering services in specific. We examine our approach in a scenario wherein it is essential for a multi-agent system to establish a chain of mutually attracted agents seeking to fulfill different bartering desires. And, we demonstrate and analyze the obtained results.

1 Introduction

Pocket Computing Devices (PCDs) such as Smartphones are increasingly showing the efficiency of relying on them and the importance of having them. Now, people are using different types of lightweight PCDs that allow them to check their emails, exchange faxes, surf the Internet, edit documents, do shopping, and play a role in a social network. Agents' deployments in industrial and profit-making applications are continually growing and, related research are relatively expanding (for an overview; [11, 10, 8]). Accordingly, the literature of Multi-Agent Systems (MAS) as well is witnessing the success of delivering advanced mobile services to users of PCDs, (e.g., Kore [3], mySAM [4], Andiamo [1]).

In an intersection between Distributed Problem Solving (DPS) [6, 5] and Multi-agent Systems (MAS) [14, 16], our main focus comes in a place related to the efficiency of the negotiation approaches provided to a set of interacting software agents. Several negotiation models were proposed by scholars to introduce proper negotiation protocols, mechanisms, strategies, or tactics that agents may

employ to reach mutually beneficial agreement. An example of that can be the *strategic negotiation in multiagent environments* presented in [9], and also those presented in [17, 12, 13]

Bartering is a disappearing type of trade where items of similar value are exchanged. "*Swapping*" is the modern approach to bartering, which is seen these days by means of websites that encourage end-users to build virtual communities and share similar interests. BarterCell is our approach to provide users of lightweight PCDs a bartering service on the go. Based on the location and characteristics of a specific community, BarterCell would use agents to build the chain-of-exchange that connects several interested frequenters of the same area.

This paper is organized as follows. Section 2 introduces the general architecture of BaretCell. Section 3 introduces the negotiation algorithms we propose for building the chain of mutually beneficial barters. In section 4 we describe the testing environment we evaluated BarterCell within. In section 5 we show the initial results we obtained.

2 BarterCell: Architecture

The architecture of BarterCell, as shown in figure 1, relies on users' capable devices or PCs to accomplish a successful bartering task. Via the pre-installed Java client-application, users create their own profiles using an interface allowing them to insert their service preferences, and add details related to the kind of items they are exchanging. Then users are asked to directly upload the saved data to a central agents platform that, in return, make it available to other agents. Different from the carpooling service application we presented in [1] wherein Jade [2] was used, for the central platform in BarterCell is running Jack [15], which is an interactive platform for creating and executing multi-agent systems using a component-based approach.

Our architecture relies on distributed Bluetooth access points located within a specific environment, (i.e., university), to receive inputs. Therefore, because of technology limitations, users are asked to be present within the coverage of a connecting spot to transmit their data to the central server. Once received from a user, the message or file content is made available to the multi-agent system, thus it can create a delegated agent that carries the particular characteristics of an end-user. This agent is identified using the Media Access Control (MAC) address of the device used to communicate its user's data. On behalf of users, agents start to interact, cooperate and negotiation with each other in order to achieve the predefined objectives in the given time frame.

Among other benefits, JACK was chosen to handle all of agents' interactions because of its ability to meet the requirements of large dynamic environments, which allow programmers of agents to enrich their implementations with the possibility to compatibly access several of the system resources. JACK has also made the communication language applied among involved agents with no restrictions, which made any high-level communication protocol such as KQML [7] or FIPA ACL easily accepted by the running architecture.

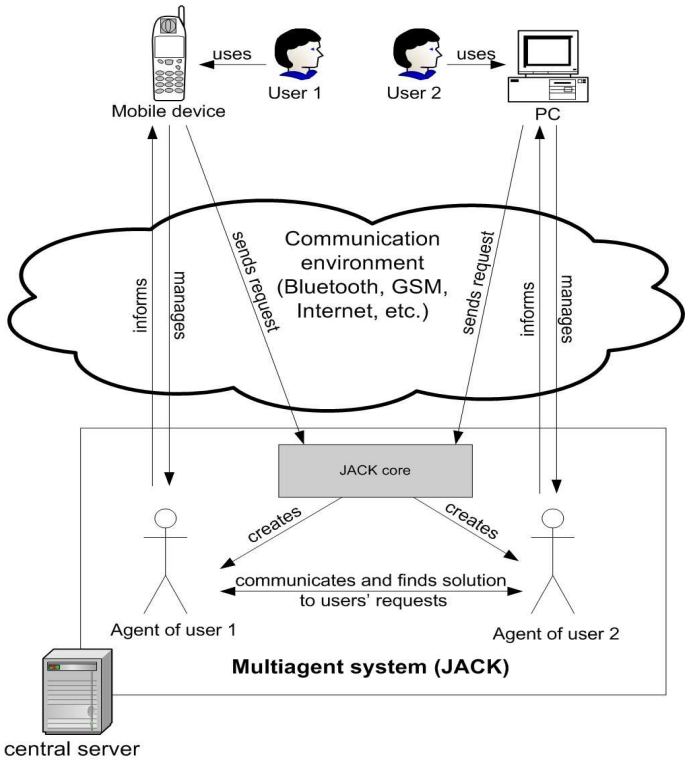

Fig. 1. The architecture of BarterCell.

3 BarterCell: Algorithms

The algorithms we present here are considering offered items names, price (estimated by owner), short textual description, time to start offer, period of time to offer and type of the item. Therefore, we assume that users who are using BarterCell own a capable mobile device with a client application installed in it. This end-user client will serve as an interface for a user to access the server-side application where our algorithm has been implemented. Then, upon a user's request, a personal agent reflecting this request is then created. This agent will then be registered within the central multi-agent system and stay active until different action is passed.

The agent created uses these variables: list of demands for all agents of a given system (`cDList`), list of offers for all agents of a given system (`cOList`), ID of an agent that will make bartering chains (`ChainMaker`), most demanded item in a system at a given time (`currentMDItem`), ID of an agent running (`currentAgent`), list of all available agents (`agentsList`), set of agents offering most demanded item (`dG`), the set of agents seeking for most offered item (`oS`),

Algorithm 1 BarterCell ChainMaker Selection Algorithm

```
1: currentAgent = agentID
2: while currentAgent ¿ 0 do
3:    cDList = cOList = currentMDItem = MOItem = ChainMaker = NIL
4:    currentChainDecision = optimalChain = NIL
5:    agentsList = getAvailableAgents()
6:    for all a_i ∈ {agentList - currentAgent} do
7:       send(a_i, currentAgent.offers, currentAgent.demands)
8:       cDList = updateCommonDemandsList(a_i.demands, cDList)
9:       cOList = updateCommonOffersList(a_i.offers, cOList)
10:   end for
11:   currentMDItem = findMostDemandedItem(cDList)
12:   dG = findMDGivers(currentMDItem, cOList)
13:   MOItem = findMostOfferedItem(cOList)
14:   oS = findMOSeekers(MOItem, cDList)
15:   ChainMaker = ChainMaker(cDList,cOList,currentMDItem,dG,oS)
16:   if ChainMaker == currentAgent then
17:      runChainMakerService(agentsList,cDList,cOList,currentMDItem,dG,oS)
18:   else
19:      T = initTimer()
20:      while agentProvidesService(ChainMaker,T) do
21:         optimalChain = getResults(ChainMaker,T)
22:         if currentAgent ∈ optimalChain then
23:            userCurrentDecision = sendResults(optimalChain,currentAgent.user)
24:            currentChainDecision=getCommonDecision(optimalChain,agentsList,currentAgent)
25:         else
26:            if currentChainDecision ==”Yes” then
27:               updateAgentODLists(optimalChain,currentAgent.offers,currentAgent.demands)
28:               sendChainContactsToUser(optimalChain,agentsList,currentAgent.user)
29:            end if
30:            sendOptChainDecision(ChainMaker,currentChainDecision)
31:         end if
32:      end while
33:   end if
34: end while
```

set of agents that are able to make an optimal bartering chain in a given system at particular time (`optimalChain`).

As seen in algorithm 1, an agent first tries to get to know other agents available in the system (Line 5). If all other agents will be already involved in a current process of chain creation, the newly arrived agent will seek the ID of the system's `ChainMaker` and notify it of its availability for bartering. The `ChainMaker` will finalize its ongoing computational cycle and inform all agents of service finish (introduced in algorithm 2), then all agents will start a new search for optimal bartering chain.

The new cycle of bartering chain creation will start again from attempting to discover all available agents in the system (Line 5) and creation of list of these

agents. For each available agent, a step is taken to communicate its demanded and offered items; (Lines 6-10). Thus all agents of the system will have common demands list (`cDList`) and common offers list (`cOList`). Based on the list of common demands, each agent finds the most demanded item in a given group of agents at given time (`currentMDItem`) and the corresponding set of agents that proposes that item; (Line 12). The agent with the most offered item other agents are seeking is then selected to further define `ChainMaker`; (Line 15).

Here, if an agent is selected to be the `ChainMaker`; (Line 16), then a simple algorithm run showing this agent's acceptance of the role of `ChainMaker` and thus providing other agents with corresponding service. Alternatively, if other agent was chosen for this role then other agents are informed to be tracking responses from `ChainMaker`; (Lines 19-31). However, tracking ChainMaker's responses, following results, and also checking for service availability are made in (Line 20). This function is designed for both parsing of messages from `ChainMaker` and checking whether it can carry out its role. Every time a `ChainMaker` finishes creating an optimal chain, it notifies all agents involved and those agents that are out of it.

Timer initialization, (Line 19), is made to check ChainMaker's availability to agents that are not interacting for a predefined time. If an agent is notified by the ChainMaker that it belongs to an optimal chain, (Line 22), it will then reports to the user it delegates whether the current status is acceptable or no by giving the chain; (Line 23). Eventually, newly selected ChainMaker starts building its list of rejected bartering chains. Having a positive decision as for proposed optimal chain, agents will send to their users contacts of other users whom they should contact in order to make barter (Line 27).

In algorithm 2, the `ChainMaker` start giving its service if it has non-empty list of own demands; (Line 3). Provided that it has the list, ChainMaker starts new computational cycle; (Lines 3-56). The cycle starts with a search for new agents; (Line 5), that might wait to join existing group of agents, which are in agentsList. If there will be at least one agent waiting to join, the ChainMaker will inform all known agents of service finish (Lines 7-9). All agents, including new, will start negotiation process from the beginning (`BarteringService Builder`).

If there are no new agents, ChainMaker will inform all agents of a new computational cycle, and then checks for optimal chains in `queuedChains[]`; (Line 16), that it has proposed during previous computational cycles (if there were any). If at least one user has previously refused to barter in a queued chain, this chain will then be considered as refused and it will never be proposed again by the current ChainMaker. Refused chains will be stored in a `refusedChains[]` set that will be updated along with `queuedChains[]` every time the ChainMaker gets information of refused chain; (Lines 18-22). Accepted bartering chains will simply be removed from `queuedChains[]`; (Line 23).

Assuming that the ChainMaker will now have a list of available agents and a TreeRootAgent, it will then start building bartering trees. Each tree will begin from treeRootAgent with every child, representing agent that demands at least one item from list of its parent's offers. While analyzing every path on such

tree the ChainMaker will find repetitions of agents, it will create a complete set of agents that can barter between them. The shortest possible chain will be recorded to `chains[]`, which will consist of shortest bartering chains of three types (combinations of demand types): 1) Strict; 2) Strict + Flexible; 3) Strict + Flexible + Potential. Eventually, the shortest chain selected is built for each corresponding combination; (Lines 25-27).

If a ChainMaker will succeed to find more than one short chain, it will select the optimal one from `chains[]`; (Line 28). Considering chains of equal length, the highest selection priority is given to a chain that will be based on Strict demands while the least priority is given to a chain that will be based on Strict + Flexible + Potential demands. Assuming that an optimal bartering chain is found, then the ChainMaker will inform all concerned agents of being fulfilled; (Line 30). Each agent in the chain will have information such as *which other agents are involved into proposed optimal chain* and *which items should be exchanged and corresponding contact information of users*. The ChainMaker will then remove, from the *common demands list* and *common offers list*, those items that are already chosen to be in proposed in the optimal chain (and will be potentially exchanged later); (Lines 31-32).

If one of optimal chains will be refused to be executed, ChainMaker will restore items that were involved into it ;(Lines 20-21). Every proposed optimal chain will be placed into `queuedChains[]`; (Line 36) to further track whether it will be accepted by users or not. Every agent that will wait for results from ChainMaker and will not be involved into optimal chain, will get a message "cycle finished"; (Lines 37-39). This will be indicator that ChainMaker has finished computing optimal chain, during previous computational cycle that agent was not into it and new computational cycle will be started by the same ChainMaker. This message will cause every agent's timer restart to check chain making service availability.

If ChainMaker fails to achieve a goal, it will notify all involved agents; (Lines 41-43). This message will cause the restart of negotiation process "`BarteringService Builder`". If optimal chain will consist not only of Strict demands items then the ChainMaker tries to make it so by changing treeRootAgent to the next most appropriate agent; (Lines 47-48). In the rest of the algorithm, if there will not be any agent for *current most demanded item*, the next most demanded item and corresponding treeRootAgent will be chosen. If finished with the list of demands or a suspension message received from its user, the ChainMaker will inform all agents of service termination. Agents still interested in bartering service will restart a new negotiation process.

4 BarterCell: Testing Environment

To test our architecture we used a D-Link DBT-900AP Bluetooth Access Point that is connected to the university LAN through a standard 10/100 Mbit Ethernet interface. This device offers a maximum of 20 meters connectivity range

Algorithm 2 BarterCell's ChainMaker Operationing Algorithm

refusedChains[] = queuedChains[] = newAgentsQueue = optimalChain = NIL
treeRootAgent = currentAgent
while currentAgent.demands ¡¿ NIL **do**
  chains[] = NIL
  newAgentsQueue = searchNewAgents(agentsList)
  if newAgentsQueue ¡¿ NIL **then**
    for all a_i ∈{agentsList - currentAgent} **do**
      inform (a_i, "service finished")
    end for
    return NIL
  else
    for all a_i ∈{agentsList - currentAgent} **do**
      inform (a_i, "new cycle start")
    end for
  end if
  for all $chain_i$ ∈ queuedChains[] **do**
    if hasDecision($chain_i$) **then**
      if $chain_i$.decision == "No" **then**
        refusedChains[] = refusedChains[] + $chain_i$
        cDList = restoreCommonDemandsList($chain_i$)
        cOList = restoreCommonOffersList($chain_i$)
      end if
      queuedChains[] = queuedChains[] - $chain_i$
    end if
  end for
  chains[] = findShortestChain(agentsList, treeRootAgent, refusedChains[], "S")
  if chains[] ¡¿ NIL **then**
    optimalChain = chooseOptimalChain(chains[])
    for all a_i ∈ optimalChain **do**
      inform (a_i, *optimalChain*)
      cDList = removeFromCommonDemandsList(a_i.demands, cDList)
      cOList = removeFromCommonOffersList(a_i.offers, cOList)
    end for
    queuedChains[] = queuedChains[] + optimalChain
    for all a_i ∈{agentsList - optimalChain - currentAgent} **do**
      inform (a_i, "cycle finished")
      cDList = removeFromCommonDemandsList(a_i.demands, cDList)
      cOList = removeFromCommonOffersList(a_i.offers, cOList)
    end for
  else
    for all a_i ∈{agentsList - currentAgent} **do**
      inform (a_i, "no bartering chain")
    end for
    return NIL
  end if
  if includesFOrPDemands(optimalChain) **then**
    if existNextAgent(treeRootAgent, currentMDItem, cDList) **then**
      treeRootAgent = nextAgent(treeRootAgent, currentMDItem, cDList)
    else
      if existNextItem(currentMDItem, cDList) **then**
        currentMDItem = nextItem(currentMDItem, cDList)
      end if
    end if
  end if
  treeRootAgent = ChainMaker(cDList, cOList, currentMDItem, dG, oS)
end while
for all a_i ∈ {agentsList - currentAgent} **do**
  inform (a_i, "service finished")
end for

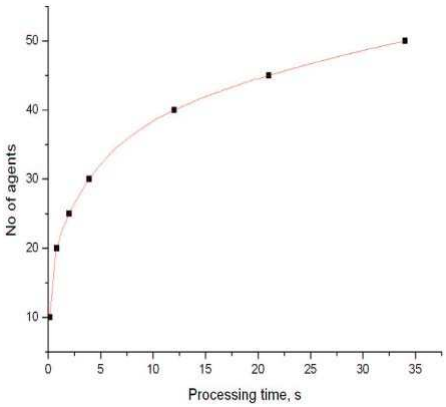

Fig. 2. Simulating the number of Agents

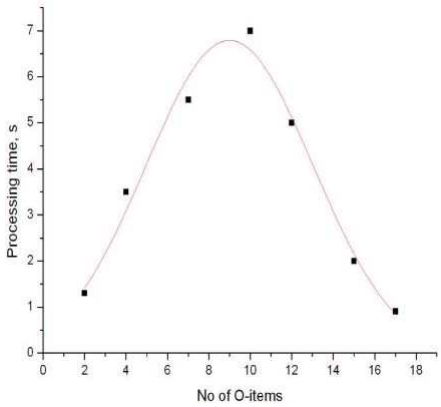

Fig. 3. System Load Distribution

with the maximal bit rate support of 723Kbps, and the possibility to concurrently connect up to seven Bluetooth-enabled devices. The same access point is authenticating pocket devices that have BarterCell previously installed in it and, it works as a deliverer of the service requests and responses from and to the central servers. On the end-user side, four competent cell phones were used to communicate semi-adjusted bartering interests with central servers. These devices are Nokia 6600, 6260, 6630 and XDA Mini. On the server side, a capable PC was used with JACK 5.0 and BlueCove installed in it.

5 BarterCell: Results

While simulation BarterCell we used JDots for tree building, which is object oriented software component. Each node of a tree was built with JDots representing an object with its own fields and methods. Our algorithm works slower with a large number of agents (e.g., ¿300) because the main agent needs to build three trees. Nevertheless, while testing with less than 200 agents, our algorithms are giving efficient results in time.

Figure 2, shows how fast the main agent finishes searching for possible optimal chains depending on the total number of known agents. Here, we assumed that the number of O-items is 5 and the number of D-items is 15.

Figure 3 shows how fast the main agent finish searching for all possible optimal chains depending on number of items that each agent proposes. Total number of offered + desired items is constant (20). Peak of the graph represents the most time consuming state when number of offered items is equal to number of desired items. In this state the main agent has the biggest number of possible exchange combinations. Assumptions used, Max number of items: 20, Max number of D-items: 15, and Number of agents: 30.

Figure 4 shows how fast main agent will finish searching for all possible optimal chains depending on number of items at each known agent. In this run,

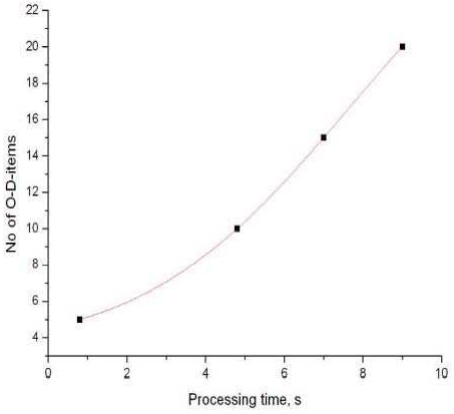

Fig. 4. Simulating the number of items at each agent level

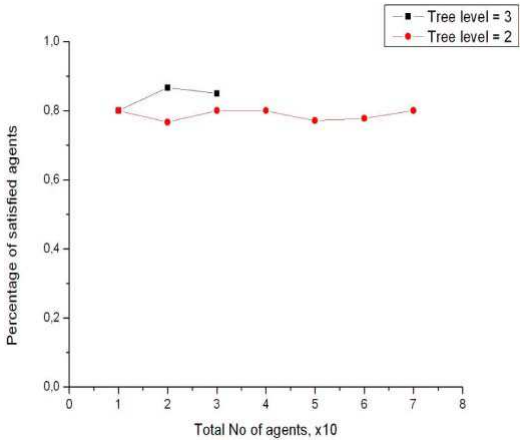

Fig. 5. Agent Satisfaction Level

results here are particular since we assumed that the number of demanded items `D-items` = number of offered items `O-items`, and the number of D-items are only of "Strict" type and the number of agents is equal to 30.

Figure 5 represents how many agents would be satisfied (i.e. involved in one of optimal chains produced by main agent) until the main agent finishes all possible chain-building processes. Depending on the trees level (depth) the percentage of satisfaction will vary. Here, we assumed that the number of D-items is 20 and the number of O-items is 5.

6 Conclusions

In this paper we introduced BarterCell that is a software architecture for providing location-based bartering service for users of pocket computing devices. This application was developed to: 1) revive the idea of bartering within members of a specific community and promoting the benefits of location-based services, 2) to test new voting-*like* negotiation algorithms for agents representing nomadic users and interacting very actively on-the-go, 3) motivate existing users of pocket devices, and attracting new ones to benefit from recent advanced technologies by widening the range of services that can be offered to them on the go.

Acknowledgment

Authors would like to acknowledge the support of the Science Foundation Ireland (SFI) under the grant 07/CE/I1147 as well as the European regional Development Fund (ERDF) for supporting partially the SC project trough the Ireland Wales Program (INTERREG 4A). Authors would like to thank Oleksiy Chayka for the effort made to implement BarterCell.

References

1. Abdel-Naby, S., Fante, S., Giorgini, P.: Auctions negotiation for mobile rideshare service. In: Proceedings of the Second International Conference on Pervasive Computing and Applications (ICPCA07). IEEE, Birmingham, UK (July 2007)
2. Bellifemine, F., Rimassa, G.: Developing multi-agent systems with a fipa-compliant agent framework. Software - Practice & Experience 31(2), 103–128 (2001)
3. Bombara, M., Calì, D., Santoro, C.: KORE: A multi-agent system to assist museum visitors. In: WOA. pp. 175–178. Villasimius, CA, Italy (September 2003)
4. Bucur, O., Boissier, O., Beaune, P.: A context-based architecture for learning how to make contextualized decisions. In: Proceedings of the First International Workshop on Managing Context Information in Mobile and Pervasive Environments. Ayia Napa, Cyprus (May 2005)
5. Durfee, E.H.: Distributed problem solving and planning. In: EASSS'01: Selected Tutorial Papers from the 9th ECCAI Advanced Course ACAI 2001 and Agent Link's 3rd European Agent Systems Summer School on Multi-Agent Systems and Applications. pp. 118–149. Springer-Verlag, London, UK (2001)
6. Durfee, E.H., Lesser, V.R., Corkill, D.D.: Trends in cooperative distributed problem solving. IEEE Transactions on Knowledge and Data Engineering 1(1), 63–83 (1989)
7. Finin, T., Fritzson, R., McKay, D.: A language and protocol to support intelligent agent interoperability. In: The Proceedings of the CE&CALS Conference. Morgan Kaufmann, Washington, USA (June 1992)
8. Jennings, N.R., Crabtree, B.: The practical application of intelligent agents and multi-agent technology. Applied Artificial Intelligence 11(5), 3–4 (1997)
9. Kraus, S.: Strategic negotiation in multiagent environments. MIT Press, Cambridge, MA, USA (September 2001)
10. Mckean, J., Shorter, H., Luck, M., Mcburney, P., Willmott, S.: Technology diffusion: analysing the diffusion of agent technologies. Autonomous Agents and Multi-Agent Systems 17(3), 372–396 (2008)
11. Munroe, S., Miller, T., Belecheanu, R.A., Pěchouček, M., McBurney, P., Luck, M.: Crossing the agent technology chasm: Lessons, experiences and challenges in commercial applications of agents. The Knowledge Engineering Review 21(4), 345–392 (2006)
12. Sierra, C., Faratin, P., Jennings, N.R.: A service-oriented negotiation model between autonomous agents. In: Collaboration between Human and Artificial Societies, Coordination and Agent-Based Distributed Computing. pp. 201–219. Springer-Verlag, London, UK (1999)
13. Smith, R.G.: The contract net protocol: High-level communication and control in a distributed problem solver. IEEE Transactions on Computers C-29(12), 1104–1113 (1981)
14. Weiss, G.: Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence. The MIT Press, Cambridge, MA, USA (March 1999)
15. Winikoff, M.: JACK intelligent agents: An industrial strength platform. In: Bordini, R.H., Dastani, M., Dix, J., Seghrouchni, A.E.F. (eds.) Multi-Agent Programming, chap. 7, pp. 175–193. Springer US, Soeul, Korea (2005)
16. Wooldridge, M.: An Introduction to MultiAgent Systems. John Wiley & Sons, New York, NY, USA (June 2002)
17. Zlotkin, G., Rosenschein, J.S.: Negotiation and task sharing among autonomous agents in cooperative domains. In: Proceedings of the Eleventh International Joint Conference on Artificial Intelligence. pp. 912–917. ACM, San Mateo, CA (1989)