Department of Computer Science
Aachen University of Technology
prof. Gerhard Lakemayer
prof. Dr.-Ing. Klaus Wehrle

Department of Information and Communication Technology
University of Trento
prof. Paolo Giorgini

Master thesis

Optimal distribution of barter resources in a mobile multiagent system

Oleksiy Chayka

Matr. No: 268652

2008

Abstract

Autonomous software agents best suits in development of applications that tie together computing pocket devices and sophisticated mobile services. Efficiency of software agents' usage depends on negotiation protocols used in multiagent systems. Negotiation protocol that will efficiently consider all given constraints on available resources, given tasks and expected results will bring the most benefits to users. We introduce new negotiation protocol that considers recent requirements of a distributed dynamic mobile environment, computing capabilities of user pocket devices and limitations of particular system resources.

In a multiagent system each software agent represents its user and tries to make barter with other agents considering its user resources and preferences. Communication protocol and algorithm of optimal barter chain calculation help to find an optimal solution in a given multiagent system. User will get interactive response from personal agent in a localized system on availability of the most optimal barter with other users of the system.

Each agent in the system has quasi peer-to-peer architecture, making the system decentralized and highly adaptive to any environment.

Efficiency of the proposed algorithm implementation we have compared to that of an adopted version of the most popular negotiation protocol – Strategic negotiation protocol.

Table of contents

1. Introduction

Today people acquire more and more products to satisfy their living needs in the world which demands intensification of their activities in social and private lives. Besides obvious advantages, acquisition of a new product raises a question: what to do with a product when it will become obsolete for its owner?

Barter can help to get rid of unused stuff and to get something useful in exchange. Unfortunately nowadays barter has almost disappeared because of certain complications to do it. People can spend a lot of time to find a way to exchange (or even to sell) their items for something else that will be useful for them. To save the time they usually can store such products in hope to use them later or give them to friends, but most often people throw them out and buy new products.

Exchange of products can occur in a group of people with similar interests. Nowadays there are special places where people gather to barter (or sell) their old stuff – flea markets. Such markets usually have substantial disadvantages: they require dedicated area; people need to come there at particular time and spend a lot of time, trying to find appropriate swap option. Even having gathered many people with similar interests in one place is not enough to satisfy needs for barter of most of them. Discovery of optimal exchange options is a complex non-trivial task. Currently there is no easy mechanism to discover such options. To find a mutual barter even between two people can be not easy task for a human, but barter can be more complex and involve not only two persons. There can be a chain of several people that can only altogether mutually satisfy their barter needs. Discovery of such chains in a given community can optimize exchange process inside it and make the most people satisfied. Finding such chains will require enormous effort from people if they will not use any help of machine, because all possible optimal barter options among all available people with their resources must be considered. The current work aims to introduce an IT solution that will help people to discover their exchange options with other people, thus greatly facilitating application of the idea of barter.

To simulate not only user behavior, but also models of interactions of people in a given community, we have considered intellectual software agents as the most appropriate technology for this. After software agents will create virtual communities – multiagent systems – they will simulate interactions that could be found in a real life in groups of people that share similar interests. Carrying corresponding user data, software agents can wisely negotiate between them and find a solution that will be optimal for their community.

Efficiency of a goal achievement depends on protocol and negotiation strategy applied. Ongoing research on the negotiation protocols used in multiagent systems is limited to distributed systems that consider physical presence of open computing networks [7, 15]. However, there is no much research in application of software agents in dynamic mobile environment.

Obviously, standard negotiation mechanisms of cooperative, intelligent and egoistic agents must be improved to satisfy demands of such environment.

In our work we propose an innovative algorithm of optimal barter finding in a given community and corresponding negotiation protocol that will consider features of dynamic mobile environment. Considering existing software technologies, the algorithm and negotiation strategy can be wisely implemented by software agents that will compose hybrid decentralized multiagent system. Architecture of our system (that we called BarterCell) includes user mobile devices, (optional) personal computers and a central server that will host all software agents. Personal software agents will be created and managed by users through their mobile devices or PCs. Agents in the BarterCell will communicate between themselves, reporting results to their users according to our protocol.

Since calculation of optimal barter chain for a community can be done by one of its agent, for each computational cycle there will be selected one agent of that community that will be responsible for a solution finding. Selection process will consider probability of each agent to be involved into optimal solution. The probability will be calculated based on trade resources possessed by each agent in context of its community. According to our model, goal of each agent is to find an optimal solution for its community, sidelining own egoistic goals. Thus each agent of the BarterCell will trust other agents of its community and will not try to cheat them.

As a result we will demonstrate efficiency of proposed negotiation model – the negotiation protocol and the algorithm for optimal barter chain finding. We will compare performance of agents' negotiations when they will implement our negotiation model and when they will implement the most popular protocol – Strategic negotiation [3]. Simulation of agents' interaction will be shown in real-life situations when computational resources are limited and system response is time-bounded. Advantages and disadvantages of the BarterCell and used middleware in our experiments will be discussed.

The rest of the thesis is organized as follows. In chapter 2 we give overview of existing agents' technology and show its role in the BarterCell. We also give overview and evaluation of existing negotiation mechanisms and a gap of their application in dynamic mobile environment. In chapter 3 we present multiagent platform that our solution is based on. Chapter 4 describes the BarterCell – its role in finding solution to the barter problem, its architecture, applied algorithms and communication protocols. Chapter 5 presents experimental results produced by the system using proposed algorithms. It also shows comparison of those results with application of adapted version of Strategic negotiation protocol. In chapter 6 we summarize objectives of our work, achieved results and outline future work.

2. Agent-oriented approach

2.1 Software agents in the BarterCell

Concept of agent can vary from biological entities (animals) to robotic and computational agents. The last type of agents usually designed to fulfill a given task, perceiving their environment and acting according to their embedded plan. Such plan can foresee options of task fulfillment both by agent along and in cooperation with many other agents.

This work is based on the computational agents, in particular on software agents. Those agents are programs, designed for potential users to fulfill their particular tasks.

Putting aside computer nature of a software agent, its performance depends on level of intelligence it implements in algorithms to achieve a given task. Intelligent agent can handle unforeseen situations and adapt to a new environment by applying its additional knowledge of environment.

In addition to agents intelligence another important feature is their autonomy. Autonomous agent can decide how to achieve a given goal, cooperating or not with other agents. Interaction of user with such agent can be restricted to giving task and getting results only.

Agents that embody intelligence with autonomy, ability to learn and to cooperate with other agents, are smart (hybrid) agents [23].

Agents of the BarterCell are designed to have ability to learn. Such agents can store information from their environment in some form and then use it during acting and decision making process. Learning allows agents to improve their performance at carrying out a particular task. Agents of the BarterCell can adapt to decisions of other users (indicated by their agents).

Depending on the role agent plays in a given community, its behavior can vary from that of goal-based agent to behavior of utility-based agent. Goal-based agent tries to achieve its goals by perceiving information from environment and correspondingly acting. Utility-based agent not only tries to achieve a set of goals, but also tries to maximize utility values of its utility function. Such values indicates how successful agent is in its goal achieving. It may also indicate how long it can take from agent to finish a given task.

Each agent in the BarterCell embodies utility function. Utility values are the same for all agents of our system: they need to find the best possible common solution of a given problem inside their community. In a given community of agents at every moment there can be at most one agent that will be directed by its utility function. That agent will play temporal role of a manager (chain maker) of known to it agents.

According to our algorithm after period of time (usually defined by computational cycle) role of a chain maker can be passed to another agent of a community.

Our agents cooperate with other agents of their environment in order to find a solution to a given problem. Cooperation with their users bounds to initial information receiving, results of a solution publishing and their confirmation by users. None of agents can resolve the problem alone, but it can do this in cooperation with others only. Sometimes agents can lie during communication with other agents, but in the BarterCell all agents are designed to be truthful. Operation with unfaithful information by any of agents in the system will not bring any benefits to its user, that's why we suppose that all users of the BarterCell will provide their agents with trustworthy information.

2.2 Multiagent systems

Simple reactive agents can deal with wide range of various tasks. But such tasks are limited with set of rules that those agents are based on. Unknown situation for reactive agent can lead to a failure of results delivery. In fact, most agents exist in agent environments only. Such dynamic operating environments allow agents, even without their ability to learn, fulfil much more complex tasks that can be done only in cooperation of many agents.

Usually each agent of the system has insufficient information to achieve the goal or is unable to do it by other reasons. Forming multiagent systems, agents exploit their common potential to achieve individual or common complex goals. There is no centralized control of such multiagent system, since all agents form it based on equal rights.

Communication and collaboration are of most desirable properties that every multiagent system must have. Communication means that agent can inform other agent of the system of any relevant changes in their environment or other discoveries that might be useful for other agents. Collaboration is ability to work together to achieve a common goal.

Another useful property of multiagent systems is that their agents are able to do simple operations and communicate with each other in a simple way. Individual agent not necessarily should know the global problem for the multiagent system, since overall result can benefit from investments of each agent of the system.

Multiagent systems can be of competitive or collaborative nature. Competitive systems consist of agents that have opposite goals to achieve (for example, agents on stock exchange) while agents of collaborative systems benefit from achieving common goal, thus expecting maximization of their benefits and sharing resources with other agents (possibly, considering trust level to them).

2.2.1 Collaborative agents

Multiagent systems where agents collaborate with each other to achieve a certain common goal are collaborative agent systems.

Collaboration of agents in such systems usually means that agents will not have ability to learn. In special cases there could be some basic learning abilities of such agents. Nevertheless agents keep their intelligence to fulfil a given task as effective as possible. The problem that can be solved by collaborative system is impossible task for a single agent in the system.

Since problem solving does not rely on a single agent, the system is protected from having failures of some agents or even from having incorrect information from them, because in that case other agents will correct wrong agent. Additional agents can be added to the system to make it more reliable than traditional multiagent systems.

2.2.2. Negotiation in multiagent systems

According to [7], negotiation protocol in multi-agent systems is ”public rules by which agents will come to agreements, including the kinds of deals agents can make and the sequence of offers/counter-offers that are allowed”. These rules can be changed depending on the environment where multi-agent system is located. They have distinguished between high-level negotiation protocol like those in multi-agent systems and low-level negotiation protocols like those in networks (e.g., Ethernet). The key factor of such stratification is the fact that high-level protocols are concerned with the content of the communication and not the mechanism that content use to transfer.

In [12] Smith considered high-level protocols as methods that lead system designers to decide ”what nodes should say to each other” while low-level protocols lead system designers to decide ”how nodes should talk to each other”. He defined a proper negotiation protocol that must implement two parts:

– connection issue: related to the distribution of tasks among nodes in a task-sharing environments, so that nodes looking for certain task execution can easily find their proper neighbours.

– contract negotiation: assumes simultaneous operation of both nodes asking to execute tasks and nodes ready to handle it. The asking nodes broadcast a call for proposals, and the helping nodes submit their offers.

In context of multiagent systems, proper negotiation protocol assumes that each agent there is seeking for other agents that can help to fulfil their common complex task. Those agents are supposed to create a coalition to solve their task. Such coalition can yield results that can not be achieved in case of individual problem solving by any of its agent.

3 JACK multiagent platform

JACK is an environment to build, test, integrate and document both agent-oriented software components and multiagent systems. It's built on top on Java language, supporting cross-platform usage. From Java it took low level programming basis, bringing ability to implement high level agent behaviour model. JACK was developed to expand object-oriented Java language with agent-oriented constructions.

In contrast with usage of usual object-oriented programming style, agent-oriented style forces to change logical and/or physical design of a resulting software system, because it introduces key concepts that changes system design. The system is consisted of agents which are autonomous entities that can reason, make pro-active decisions and respond to events of their environment.

3.1 The JACK agent language

JACK agent language is a programming language used to describe software system using agent concepts. The JACK programming language is based on Java language, inheriting its syntax and extending it to represent agent-oriented features. The JACK agent language is not only extending Java, but also introducing new key agent-oriented concepts that can't be efficiently presented with native Java mechanism.

Set of base classes make a core that keeps together the whole agent-oriented program. Those classes are defined in kernel of the JACK. The JACK compiler pre-processes source code and converts it to a Java code that can be further compiled to machine code to run on a target system.

Object-oriented and agent-oriented programming paradigms have some syntactic and semantic differences between them. The JACK agent language introduces to the standard Java syntax its own extensions at 3 levels:

1) the class definition level, providing a set of classes that agent-oriented constructs can inherit from;

2) the declaration level, providing a set of statements that identify relationships between those classes;

3) the statement level, providing a set of statements that can operate on data structures specific for the JACK agent language.

Runtime behaviour of programs written in object-oriented style and those written in agent-oriented differs a lot, since modelling concepts in these two cases are different.

The JACK agent language extends the Java execution engine in the following way:

1) Multi-threading control has been removed from programmer and built into the kernel of the JACK language. Programmers can implicitly specify

multi-threading using special instructions of the language. Such instructions provide transparent control over multi-threading program execution.

2) JACK program execution follows software agent life model: it processes its set of plans, accessing to its set of beliefs. To handle incoming events, agent executes its corresponding plans, consulting with its beliefs, if necessary. If some event requires complex plan execution, agent may run subtasks, trying to fulfil a given task.

3) A new data structure called logical member represents knowledge of fact from environment. This logical member has unknown value until its initialization. Once initialized, it cannot be changed.

3.2 The JACK agent kernel

The JACK agent kernel is an engine to execute programs written in JACK language. It consists of set of classes that allows implementation of agent-oriented model. Some of those classes implement transparent call to Java classes, allowing agents to execute their code, while other classes explicitly provide agents with their unique functionality.

Considering all abovementioned properties of the JACK multiagent platform, we have chosen exactly this platform in our work. Strengthen development of software agents by introducing semantic and syntactic extensions to an object-oriented language, keeping cross-platform nature of a base language and having high research interoperability quality, JACK environment best fits in our work.

3.3 JACK agents

JACK is based on software agents of intelligent type. All they have theoretical BDI (Beliefs, Desires, Intentions) model of reasoning and acting.

According to the BDI model, JACK agents are intelligent autonomous software components that have definite goals, set of beliefs and desires to achieve given goals or handle events from their environment. Plans include description of actions agents should do to achieve a certain goal. Set of plans allow agent act autonomously, reacting correspondingly to incoming events.

When run, BDI agent will try to pursue its goals (desires) and according to its beliefs about surrounding environment and its state, it will follow some of its plans (intentions). Agent may want to achieve some of its goals because of its user or another agent’s request or such desire can be originated from some event or when some beliefs of the agent will change.

JACK software agent can reveal behavior of both reactive agents (that acts according to incoming events) and proactive agents (that are directed by their goals).

Each JACK agent has:

1) set of beliefs about the world;

2) set of events (that it will respond to);

3) set of goals that it may desire to achieve;

4) set of plans that describe how it can handle its goals or events that may appear.

When an agent will run in a multiagent system, it will wait until some of its goals become active (or until agent will get its goals) or arise some event that agent must respond to. When agent will have such goal or event, it will determine which actions it will take next. In situation when agent will believe that given goal has already been achieved or certain event has already been handled, that agent will do nothing. Otherwise agent will search in its plans the most suitable set of plans to respond to an event or achieve a goal in certain situation. In case of failure of some plan execution, agent will try to find and execute alternative plans. It will try to search alternative plans until there are left any for particular situation.

Set of plans describe in procedural language how to achieve a certain goal or how to respond to a particular event. At first glance it may seem to be like an expert system behavior – consulting knowledge base with set of rules and actions for particular situations. But the difference of software agent is that it can behave as a rational instance, processing available plans and exhibiting such features of rational behavior as:

1) Goal-directed focus (when agent focus on its goal achievement rather than on method to do it);

2) Real-time context sensitivity (analysing given situation at particular moment of time, agent will decide which actions it will be better for it to take at that moment);

3) Real-time validation of approach (agent ensures that its chosen way to achieve a certain goal is valid only as far as conditions that lead for this way to be chosen, remain the same);

4) Concurrency (exploiting its multithreading nature, agent will decide how to handle newly arrived tasks or events, whether to put them into concurrent execution and with which priority).

JACK agents fit very well for developing systems where time and mission is a critical issue, since BDI model allows validation and verification of such systems. One of tasks of this system can be online tracking of agents' intentions and progress in current goal achievement.

4. BarterCell

4.1 General scenario

For one computational unit it is sometimes impossible to achieve a given task due to lack of resources (e.g., information, hardware power, etc.). Such complex tasks can be completed in cooperation with other system entities so that each entity is able to solve a part of the overall goal. Solving a joint problem means that there must be correct negotiation mechanism between all of the units involved into the same process.

In a decentralized environment it is hard for all peer units to organize themselves in a way to effectively solve a given problem. It is even harder when those units are asked to find definite structure (sequence of their cooperation, scheme or their resources exchange, etc.). In order to choose the optimal cooperation scheme in a given set of peer units at particular time, we made agents organize themselves to choose the group manager that will "see" the overall picture of all given units and can decide the best possible cooperation scheme. We assume that each of those peers trusts to every other peer and has the same criterion of solution optimality and utility function of group manager selection as others. The only difference could be resources that are available to each unit.

Initial phase of cooperation of a given group of agents includes discovery of all available peers. Then exchange of information on available resources will take place between all of them. Based on this information each unit will compute which of them will be group manager to further define the optimal cooperation scheme. Group manager agent will be the one involved into the next optimal solution processes with the highest probability among all other agents. This could be defined by resources it will have in that exact moment.

Group manager will first notify all other units of its role acceptance. Other units will wait for results from group manager, which will implement algorithm on finding optimal solution in the group. This algorithm will rely on resources of the group and will not require any communication between units. When the group manager will find solution, it will notify all units whether they are involved into an optimal solution or not. If an agent must take part in a foreseen solution, it will "know" the overall cooperation mechanism between agents in that scenario (set of units of the system involved, sequence of their cooperation and resources that they should give to others or get from them). Group manager will have a short-term list of solutions that were proposed and refused by other units. Newly selected manager unit will start tracking such list from scratch.

To update the list of available resources, group manager will search for optimal solution until it can find any. As far as manager unit has its interests in group cooperation and it will be able to carry out its role, it will keep its main task. As soon as a group manager refuses its role or due to a sudden occurrence

of malfunction, all other agents choose another manager by restarting the negotiation process with discovery and information exchange phase.

Other units can join existing group at run-time by notifying the manager agent. Eventually, after the manager agent finishes its current cycle for optimal solution discovery, it will report the solution to involved agents and then it will inform all agents of negotiation process to restart. From the discovery phase recently arrived units will be part of a new cycle.

4.2 Barter with a wise negotiation

When somebody will decide to barter his item to another item, that will be more useful, and use the BarterCell system for this, first he should set personal agent on mobile device and specify items that he wants to exchange using the system (for more details please refer to chapter 4.3 “Software design”). That agent will represent its user in virtual flea market, searching for other users that want to get items of that user and users that can propose needed items.

In order to more effectively satisfy user needs we introduced 3 types of user demands:

1. Strict demands - those caused user to use the system. This includes all items that user strictly want to acquire, using the BarterCell.

2. Flexible demands - those specified by user explicitly as possible alternatives to his strict demand. These are items that user can accept if in case if there will not be any proposition of items of Strict demand, but the user will want to get rid of something.

3. Potential demands - those specified by user implicitly as areas of interest. Among items of this type there could be even items, that user could not think about to acquire before using the system, but since they are in area of his interest, he could be interested to get them if he could know that they are available for barter.

To keep consistency of item types and ensure that all users of the system can’t mention the same item using different names (still, due to flexibility of the system, for persistent users there is opportunity to make a name mess), each item type and areas of user interest should be specified with help of ontology of item types. Ontology will be predefined during localization of the system but it will be flexible enough to foresee unhandled types of items.

System localization will consider features of working environment, such as physical parameters of a place, social characteristics of system users, etc.

Having entered all necessary barter-related information (this also includes user contact data), user can activate personal agent by sending request to do this into available multiagent system. Multiagent system is analogous of a flea market where software agents, representing interests of their users, will wisely negotiate how they can most efficiently barter between them.

Barter could involve 2 or more users, if there will be some group of users that will mutually satisfy their needs (for example for group of 3 people there can be situation like in fig. 4.1). Such groups of users and order of items to be exchanged must be calculated based on information of all items available in the system for exchange and desires of all users in it. Being activated in the system, each agent will try to discover which other agents will be available to it. Having discovered surrounding agents, each agent will try to exchange item information with all of them. After exchange phase every agent has list of all demanded items on the system and all offered items. Moreover some personal-specific information (that user will allow to expose) will be also at disposition of every agent.

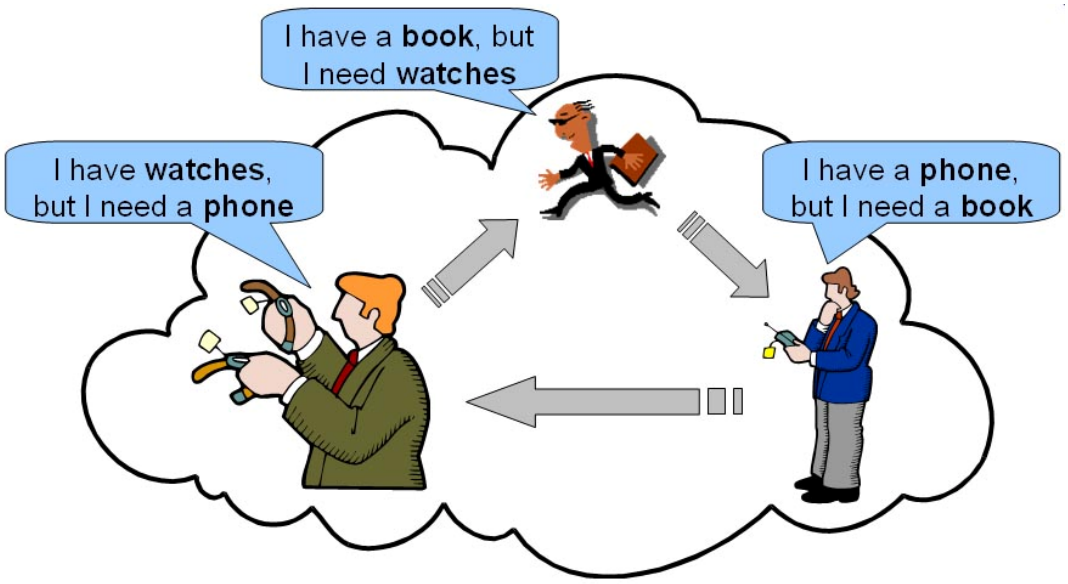

Figure 4.1 – Barter in a group of users

From this moment every agent can “see” the overall field of items: who wants what in exchange for what. To figure out which users can create the most efficient barter in given situation, calculations of possible barter options should be done. Such calculations can be done by any of peer of the system, but it will be enough if only one agent in a group will do this, avoiding computational resources overuse. Such agent can be the one, which most probably will be included in optimal barter chain. Optimal chain will be the shortest possible barter chain considering priority of demand types. Every agent of a group will pre-calculate initial information to know which agent can be included in optimal chain with the highest probability among others. Let’s call such agent ”chain maker”. Later on we will describe why we need it, what it does and how other agents will know which agent is the chain maker.

4.3 Software design

Each user of the system that wants to barter something will have to set his offered items (O-items) and demanded items (D-items) using client part of the BarterCell. For each O-item such attributes should be defined:

- Item type – will be chosen by user with the help of items ontology. The ontology can be used as a directive to use common item names among all users of the system, but it can be customized to cover unforeseen items. For example “Tracking->Boots”.
- Item name – name that possible includes very short additional information like trademark. For example “Technica”.
- Price (estimated price of the item – used to find an appropriate barter partners)
- Description (short description of item: condition, special features, manufacturer, etc.)
- Initialization time (used to track user proposal for current item expiration)
- Period of time to offer (desired time to offer particular item)

Each D-item will be defined by:

- Item type (by analogy with O-item type)
- Item name
- Desired price (price that user is ready to pay for the item)
- Demand type
- Initialization time
- Period of time to demand

User of the system will be presented by such attributes:

- User name (any name set by the user during the system usage to refer to him by other users)
- User ID (MAC address of user device that was used to communicate with multiagent system)
- Contacts (phone, e-mail, barter point name and address that will be used to calculate distances between potential barter users and will not be exposed to any other user of the system)
- Preferences (D-items)
- Set of O-items
- Preferred chain length (maximal number of people that can be involved in a barter chain with current user)

Based on demand types, user preferences will consist of 3 components (see details in chapter 4.2):

1) specified items (Strict demand);
2) items with a fuzzy description (Flexible demand);
3) items with potential interest for user – taken by the system from fields of interest for particular user (Potential demand).

Personal agent will have the following attributes:

- Agent ID – it will be the same as MAC address of user device with which personal agent will communicate
- User Name, Contacts, lists of O-items and D-items, other user preferences
- Chain length
- Agent Initialization time
- Chain maker's ID – if available, ID of an agent that will search for optimal solution in a given agent's community

Chain maker concept we introduced to have a system's peer that will carry out calculations to find optimal barter chains. Other agents of the system will know which agent is the chain maker at each particular period of time (if the chain maker will be defined by then). Chain maker's selection criteria are (in the order of importance):

1) most demanded product (which agent can offer it);
2) most offered product (which agent demand it);
3) by earliest time on coming to the system.

For more details on chain maker's selection process see chapter 4.5.2.

Once defined, chain maker can refuse its role if:

1. it has accomplished its own tasks OR
2. one of agents pending results from the chain maker will notice malfunction of the agent OR
3. existing agents will change their lists of items OR
4. new agents will wait to joint existing agents' coalition

Chain maker will calculate all possible exchange combinations in a given environment (with particular user data). Then optimal chain will be selected by the chain maker. It will propose the optimal chain to involved agents and restart chain building process after this. Next cycle of the chain building will consider previous (local to the chain maker) experience as to proposed optimal chains and optimal chains refused to be applied by at least one agent involved into it. Exchange chains could include more than 2 participants. In that case chains length restriction (user preference) will be considered during optimal chain selection.

Root agent concept is similar to the chain maker. It will be selected using the same criteria, but it will be used by the chain maker to build tree of possible exchanges of the users. The root agent will serve as a starting point for each such tree (since among all agents of the system the root agent has the most

chances to be involved into optimal exchange chain). Once built the tree, chain maker will analyze it and choose an optimal chain (the shortest valid chain, considering demand type preferences and amount of items to be exchanged between users). Optimal chain could be even without root agent, since analysis will be done for each subtree of the tree.

4.4 Architecture of the BarterCell

Our agents have hybrid architecture, which implements all abovementioned agents' properties. Trying to maximize goals of their own users, agents create community of peers (quasi peer-to-peer system). None of agents is autonomous (since agent itself can't resolve the whole problem by its own) but none of them is directed by any other agent.

When somebody will decide to exchange something with other people using a BarterCell application, he should have a capable mobile device or PC with a client part of BarterCell application installed on it (fig. 4.2). Client part will serve as interface for user to access server-side part of BarterCell application where negotiation algorithm of software agents has been implemented.

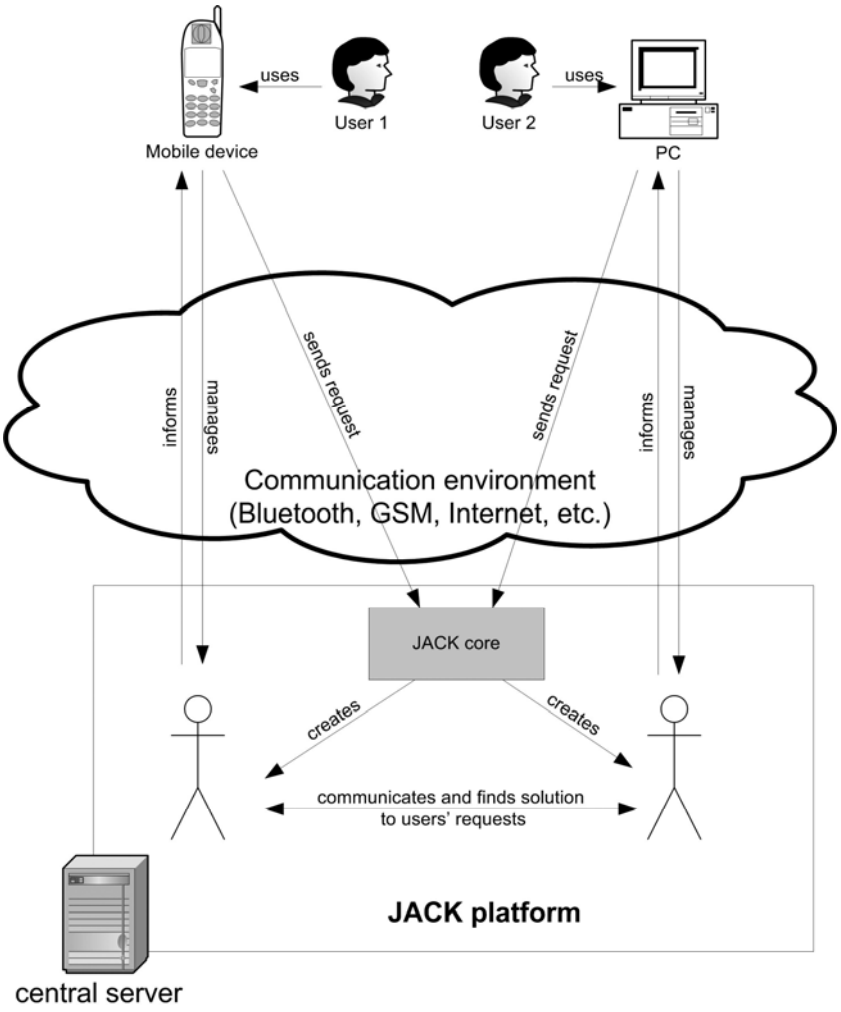

Figure 4.2 – Architecture of the BarterCell

First user will specify which items he wants to exchange and which items he wants to get from others. This information will be stored in a personal user profile that will reside in agent of user device.

User device agent is a part of a client side of BarterCell that will help to define user requests and present results of work of other agents using friendly user interface. It can reside on a capable mobile device or a PC.

Device agent will be connected to a personal agent that can be created upon user request by multiagent system's core. Indirectly managed by its user, it will reside on a server side of the BarterCell and will be responsible for cooperative (with other personal agents) solution finding and communication with corresponding device agent. Personal agent will be identified by Media Access Control (MAC) address of the device used by its user to communicate with the server.

Having defined the profile and request, user can ask device agent to send request for a personal agent creation to a JACK platform. In case of absence of the corresponding personal agent JACK platform will create the agent and send a corresponding user data to it (for more details on agent creation protocol see chapter 4.6.1).

Depending on parameters of system localization, communication environment, that will bind user device and multiagent system, can be built using BlueTooth/GSM/Internet or other appropriate technologies. It can be chosen based on local system requirements, such as speed of response to a request, reliability of a communication, etc. Due to firmness of our protocol it will work correctly with any of proposed technologies (see chapter 4.6).

Upon user request user profile will be transferred directly to an existing personal agent residing in a JACK platform through communication environment. From this point JACK platform will not be included in user-agent communication process. Nevertheless it will control interagent communication activities inside the platform (check for compliance with communication rules only).

Having got corresponding user request, agent will start to interact and negotiate with other system actors in order to achieve predefined objectives in a given time frame.

Flexibility of an agent communication protocol used in JACK platform allows using any high-level communication protocol such as KQML [28] or FIPA ACL [27] easily acceptable by the running architecture. Eventually this will ensure smoothness of any future integration between the BarterCell and other mobile service architectures, such as [24] or [25].

4.5 Introagent computations in the BarterCell

To present computations that will go on in the BarterCell's agents, we have divided them into 3 algorithmic parts: "Barter Service", "ChainMaker Selection" and "ChainMaker Operation".

The first one presents agent behavior when it comes to the system, exchanges information with other agents and waits for results. "ChainMaker Selection" represents selection criteria to choose an agent that will compute optimal barter chains. The last algorithm shows how ChainMaker will search for optimal barter chains and communicate with other agents waiting for results.

In the subsequent section you will find narrative description of those algorithms, which are presented in details in attachments.

4.5.1 Algorithm 1: Barter Service

Upon user request personal agent will be activated starting with execution of "Barter Service" algorithm. Such request can be done using capable mobile device or PC that will be connected to a central server with multiagent system and BarterCell server part on it. Both mobile device and user PC must have BarterCell client part preinstalled on them. Client part of the application will connect to a local (or global – depending on localization options) central server of BarterCell. Agent will be registered in multiagent system on central server and will stay active there until suspension by its user or until it fulfils all requests of its user.

Following the algorithm, agent will use 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 barter chains (ChainMaker), most demanded item in a system at a given time (currentMDItem), ID of an agent running the instance of a code (currentAgent), list of all active agents of a given system and their state (agentsList), set of agents which offers currently most demanded item (dG) and set of agents which seeks for currently most offered item (oS), set of agents that are able to make an optimal barter chain in a given system at particular time (optimalChain).

Being activated, agent will try to get list of other registered agents in current MAS (Line 3). List of registered agents will be stored in the core of MAS and can be given to any requesting agent of the MAS. Registered agents list will consist of active or temporarily suspended agents to the moment of the list request. Using this list of registered agents, new agents will discover all other active agents in its multiagent system and their state (Line 8).

For a new agent other active agents involved into process of optimal solution finding will have 3 roles: ordinary, chain makers and free agents. Ordinary agents must answer to discovery request reporting their state

(“waiting”) with ID of their chain maker. Chain makers will report their state (“processing data”) and own ID which can be further used by new agents as reference to existing communities. Free agents will report their state only (“free”).

All the time each software agent will be active in the MAS (Lines 2-57) it will provide discovery service for other agents. For details on discovery service of active agents and interaction of a new agent with other agents see Figure 4.9 “New agent arrival”).

Having list of discovered active agents (agentsList), each agent will decide whether to join one of already existing agents’ communities of try to establish new community that will search for optimal solution between them (Line 9). Criterion for such decision will be critical amount of agents in existing community (*N*) and amount of free agents that potentially can create new community. *N* represents maximum amount of agents in a community joining to which new agent can loose in its potential performance in comparison with establishing of a new community. It will be set according to the system’s localization parameters (hardware used, criticality of speed response for user request, etc.).

In case if there will be less than 3 free agents new agent will try to join existing community with minimal amount of agents in it disregarding parameter *N* (Lines 10-13). If there will be available more than 3 agents and all other communities will have more than *N* agents in each of them then new agent will propose to free agents establishing of a new community (Lines 15-24).

When an agent will decide to join one of existing communities, it will first select the one with minimal amount of agents in it and check if chain maker of that community isn’t in list “notRespondingAgents” of a new agent (Line 10). Such list will have all those chain makers that didn’t reply (during timeout) to the current agent’s request to join community. This list will be reset as soon as agent will restart agents’ discovery phase from the beginning (Line 3). Depending on localization criteria selection can also consider other features, like exchange item topics scope or chain makers’ features, etc. Result of such selection is chain maker (requestingChainMaker) which then should be asked to finish its service and make other agents of its community restart negotiation from discovery phase (Line 3).

Having selected the chain maker, new agent will send a request to it (Line 11), notifying of presence of agent that will wait to joint the community. After such notification new agent will wait for a definite period of time (timeout), listening to messages from selected chain maker (Line 12). When chain maker will send message “finished serving” new agent will proceed to agents discovery phase (Line 8). If during timeout new agent will not receive any message from selected chain maker, then new agent will put that agent into “notRespondingAgents” list.

Proposition to create a new community will start from sending to all discovered free agents proposal (Line 16). After definite period of time current

agent will check if all free agent agreed to create a new community (by sending symmetric proposition) or not. If not, current agent will restart agents' discovery phase, otherwise it will request from all free agents their lists of free agents (discovered from their side). This is done to verify that every agent of a given community can communicate with every other agent of that community and discovery scope of each agent is the same. Every agent will check for equality of its own list of discovered free agents with such lists from the rest of free agents (Line 20). If this supposition will be confirmed and each such list at each agent will have the same set of agents, then "agentsList" will become commonly accepted and all agents of the community will be considered as its verified members (Line 22).

If there will be inequality of at least one agentsList from one of agent (meaning that the agent has discovered different agents rather than rest agents of the community) then discovery process will begin again (Line 8).

In the case with multiagent system where management of agents' interaction is relied on single core, there usually will not be any disagreements on commonly discovered agents of the system. This is because our current architecture of multiagent system has a communication bus on a hardware platform. Once loaded to the platform, software agent will be able to communicate to other peer agents within the platform with the signal speed and communication reliability much higher than outside the platform.

Another issue concerned with agents discovery is period of time when this procedure takes place. When some agent will be initiated earlier than others, then they can finish discovery procedure earlier (and start sending list of agent discovered by them to other agents). If right at that moment new agent will arrive to the system, other agents will be able to discover it, but agents that will finish discovery procedure surely will not be able to do it. In multiagent system built according to our architecture, disagreement of common agentsList can appear only in this case. In such situation agent that will finish discovery will send their agentsLists and will wait for responses from other agents (that most probably will be in discovery phase). Agents that will notice disagreement being in discovery phase, will send responses to waiting agents, giving them notice of agents' list disagreement. After this point all agents will restart discovery phase. Such scenario could rarely appear since new agent's appearing in the system in period of time that is absolutely negligible compared to operational period of time (agents discovery vs. communication between discovered agents) has very low probability (<0.001%). Though, restarting discovery phase, all agents will start again with newly arrived agents as peers. So, appearance of a new agent even during this critical period of time will be successfully handled by our protocol.

In case of decentralized multiagent system communication environment between agents (Bluetooth, Ethernet, etc.) will serve as a system communication bus. This architecture can potentially cause significant variety of discovery scope of every agent. Moreover, discovery phase can take

significant period of time. Thus, more complex agreement mechanism should be implemented to reach agreement on common list of agents in such system. This architecture is a part of peer-to-peer system that can be implemented later based on ours one (see chapter 6 Conclusion).

Detailed scenario of new agent's arrival into the system is described in chapter 4.6.5. "New agent arrival".

After agreement of common list of agents in a community agents will start process of personal items information exchange (Lines 27-31). For each discovered agent every other agent will send lists of its demanded and offered items. Every agent will combine all those lists into 2 common for all lists: items that are needed by agents of the system and items that can be proposed by them. Thus all agents of the system will have common demands list (cDList) and common offers list (cOList) that will be basis for further computations.

If some new agent will try to join existing community of agents, but during agents' discovery phase all other agents will pass this phase and will have already agreed agentsList (that will surely not include the trying agent), then the agent will get ID of the system's ChainMaker and notify it of its desire to join to established group of agents. After ChainMaker will finalize its ongoing computational cycle it will inform all agents of its service finish (see algorithm "ChainMaker Operation"). After this point all agents (including agents pending before) will start new cycle for search of optimal barter chain.

Based on common demands list each agent will find the most demanded item in a given (established) group of agents at a given time (currentMDItem) and corresponding set of agents that proposes the item (Line 33). Most offered item (Line 34) and agents seeking for it (Line 35) will be selected to further define ChainMaker (Line 36). As a result of "ChainMaker selection" algorithm each agent will have ID of agent that will calculate optimal barter chains. That agent will manage current group of agents as long as it will carry on role of ChainMaker.

If current agent will find out that it must be ChainMaker at that time (Line 37), then it will run "ChainMaker Operation", accepting the role of ChainMaker and thus providing other agents with corresponding service. If the role must belong to another agent of the system, current agent will have to have an acceptance message from agent that must be ChainMaker (Line 39). In case of absence of such message after definite period of time, current agent will annul its agentsList and start agents' discovery phase. Having received acceptance message from ChainMaker, every other discovered agent of the system will track responses from it (Lines 40-54).

Tracking of ChainMaker's responses will also include checking for service availability (Line 41). This function is designed for both parsing of messages from ChainMaker and checking if it can carry out its role or not (for example due to unforeseen hardware of software failures that will lead to violation of a communication protocol). Every time ChainMaker will finish creation of an

optimal chain, it will notify both agents involved into it and those agents that will be out of it (in order to let all agents know the state of ChainMaker).

Timer initialization (Line 40) is done to check ChainMaker's availability by any agent that will not be notified by ChainMaker for a definite period of time. In that case agent that will notice unavailability of a ChainMaker, will notify other agents of the system of that observation. To avoid fraudulent notifications, all agents that will receive "ChainMaker failure" notification, will check it by themselves by verifying availability of the ChainMaker. When all agents of the system will discover ChainMaker's unavailability, they will start new computational cycle.

If agent will find out from response from ChainMaker that it belongs to a current optimal chain (Line 43), it will queue items of proposed solution (removing them from user lists of offers and propositions) and ask its user to accept or reject proposed chain (Line 45). After each agent of the given optimal chain will have responses from their users, common decision whether that chain is accepted or rejected will be sent to users of solution and ChainMaker (Line 51). If proposed chain will be rejected, it will not be selected any more until current ChainMaker carries out its role. Newly selected ChainMaker will start building its own list of rejected barter chains. Details on possible reactions of agent on user decisions to proposed barter see chapter 4.6.4.

Having a positive decision for proposed optimal chain, device agents will expose to their users contacts of other users whom they should contact in order to make barter.

4.5.2 Algorithm 2: ChainMaker Selection

In order to define which agent in a given system will be responsible for creation of a barter chain, each agent of the system will preprocess initial information. This information should be received during phase of agents' discovery (see "Barter Service").

Running ChainMaker function, agent will give such parameters as common demands list for all agents of the system (cDList), common offers list (cOList), most demanded item in the system at a given time (currentMDItem), set of agents which offers currently most demanded item (dG) and set of agents which seeks for currently most offered item (oS).

If dG set will be represented with only 1 agent (dG = 1), then algorithm will return this agent as a ChainMaker. In case of absence of agents in dG set, the next most demanded item will be selected from a list cDList (Line 16). For this new item corresponding dG set will be chosen from a list cOList (Line 17).

In a case if there will be more than 1 agent that offers currently most demanded item (dG > 1), attempt to find ChainMaker in a common set {dG ∩ oS} will be done. If such intersection will give the only agent, that agent will be a ChainMaker. If it will give more than one, ChainMaker will be agent from the set that stayed the most in the system. This will be figured out by comparison of initialization time. If common set {dG ∩ oS} will consist of 0 agents, then the oldest agent from dG will be returned as a ChainMaker.

This algorithm is used for 2 purposes: for every agent it helps to define which agent is ChainMaker in a given system (and thus to wait for information from it of an optimal barter chain in a system) and for ChainMaker it helps to define the root node of barter trees (see “ChainMaker Operation”).

As a result, this algorithm will give ID of an agent that must be ChainMaker based on given set of propositions and needs.

4.5.3 Algorithm 3: ChainMaker Operation

ChainMaker can give 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 3-58). The cycle starts with a search for new agents (Line 5) that might wait to join existing group of agents (that are in agentsList). If there will be at least one agent waiting to join the group, ChainMaker will send a “service finished” message to all known agents of its group and new agents waiting to join it (Lines 7-9).

Each agent got a message “service finished” will start negotiation process from the beginning (“Barter Service” algorithm).

In case if there will not be any new agent, ChainMaker will inform all agents of new computational cycle start (Lines 12-14). This is done to let all agents know that ChainMaker is available and continue providing its service, starting new computational cycle. Then it checks for optimal chains in queuedChains[] (Line 16) that it has proposed during previous computational cycles (if there were any). If at least one of user in some queued optimal chain has refused to barter in it, the whole chain will be considered as refused and it will never be proposed again by current ChainMaker as long as it will carry out its role. Refused chains will be stored in a refusedChains[] set that will be updated along with queuedChains[] every time ChainMaker gets information of refused chain (Lines 18-24). Accepted barter chains will be simply removed from queuedChains[] (Line 25).

To find out all possible barter combinations within current group of agents, ChainMaker will build barter trees that will represent barter options. Those trees will consider 3 sets of demand types: 1) Strict; 2) Strict + Flexible; 3) Strict + Flexible + Potential. For each of these combination corresponding tree will be built and corresponding type-shortest chain will be selected (Lines 28-30). The

overall shortest valid barter chain among all 3 tree types will be finally selected and recorded to chains[].

Root agent of a barter tree will be selected by ChainMaker using ChainMaker() procedure (Line 57). First execution of ChainMaker will use ChainMaker itself as a root agent, since selection criteria for the tree root are the same as for ChainMaker agent. During next computational cycles there could be more suitable agent for the tree roots (Lines 50, 57), but current ChainMaker will not refuse from computation of those tree, since BarterCell is a collaborative multiagent system.

Each tree will begin from treeRootAgent with every child, representing agents that demand at least one item from list of their parent's offers. If during analysis of every path on such tree ChainMaker will find repetitions of agents (for example: A1->A5->…->A1->…), it will select a complete set of agents that can barter between them. Clearly, such set of agents will include all those agents that will be between repeating agent in the chain (for example, repeating agent is A1).

Selecting an optimal barter chain, ChainMaker will consider its length and combination of demand types that chain will be based on. Considering chain of equal length, the highest priority will be given to a chain that will be from a tree that based on Strict demands while the least priority is given to a chain that will be from a tree based on Strict + Flexible + Potential demands (Line 32).

After selecting current optimal barter chain, ChainMaker will inform all agents from that chain of being involved into it (Line 34). Each agent in the chain will have information such as: which other agents are involved into proposed optimal chain, which items should be exchanged, corresponding contact information of users and that barter chain's ID. That ID will be a sequence of shortcuts for agents involved into the chain. ChainMaker will remove from common demands list and common offers list those items that will be in proposed optimal chain (and will be potentially exchanged later) (Lines 35-36). If one of optimal chains will be refused to be executed, ChainMaker will restore items that were involved into it (Lines 19-20). Every proposed optimal chain will be placed into queuedChains[] (Line 38) to further track whether it will be accepted by users or not.

Every agent will get a message "cycle finished" if it will wait for results from ChainMaker and will not be involved into optimal chain (Lines 39-41). This will be indicator that ChainMaker has finished computing optimal chain. This message will also show that during previous computational cycle agent, received the message, wasn't involved into optimal chain and new computational cycle is going to be started by the same ChainMaker. This message will cause restart of a timer of every agent to track chain making service availability.

If ChainMaker will fail to find any chain, it will inform all agents of service finishing (Lines 43-45). This message will cause restart of negotiation process between all agents from the beginning ("Barter Service").

Clearly, that the most perfect barter chain will be that of shortest length and consisting of items to be exchanges of Strict demand type only. If optimal chain will consist not only of Strict demand items (Lines 48-56) then ChainMaker will try to make it so by changing treeRootAgent to the next most appropriate agent (Lines 49-50). If there will not be any agent for current most demanded item, the next most demanded item and corresponding treeRootAgent will be chosen (Lines 52-54).

Having finished list of own demands or having suspension message from its user, ChainMaker will inform all agents of service termination (Lines 59-61). All agents that will be still interested in a barter service will restart negotiation process.

4.6 Interagent communication protocol

4.6.1 Sending request

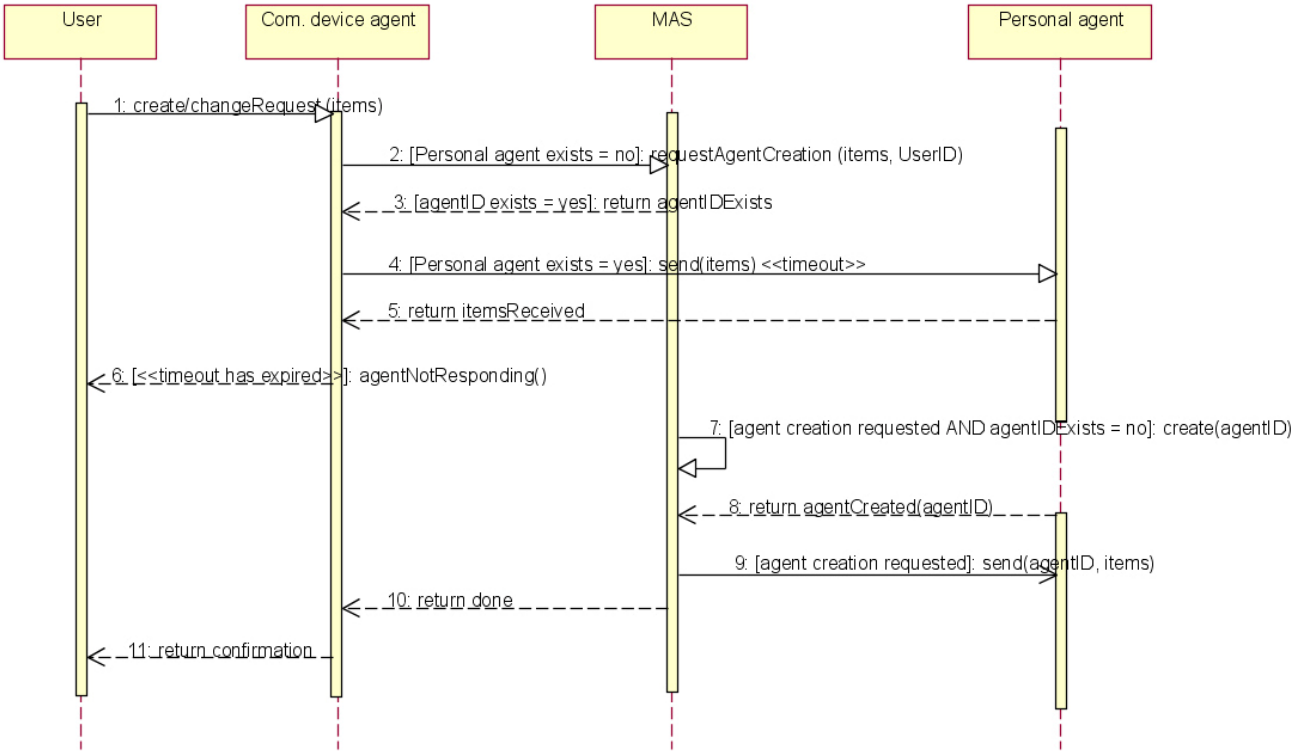

Figure 4.3 – Sending request

General scenario of the BarterCell usage begins with running by user a client part of the system. This client part will run agent that will live only within communication device of its user. To start using the system, user should set up own available resources (items that can be proposed by user to exchange) and desires (items that user wants to get in exchange for his ones). Such setup will be done on user communication device with help of device agent. After initial setup later on user will be able to change items list and personal preferences. Finish of initial setup will make device agent to send a request to corresponding MAS to create personal agent with given set of attributes.

Request for agent creation will be sent by device agent according to its believes on personal agent existence in current MAS. If device agent will believe that there must be already active personal agent in the MAS, it will try to contact that agent; otherwise device agent will request the MAS to create a personal agent. The system will check for existence of agent with given ID (that will refer to its user) in it. If such agent will be already in the system, the MAS will send to that agent given user data and reply to device agent saying that agent with given ID is already active in the current system and has received updated data.

If the MAS will verify that agent with requested ID does not exist in the system, it will create such agent. After this it will send to a newly created agent attributes given by user device agent. This phase finishes with reply message from server to device agent, which will then report results to its user.

There is a chance that reply from the MAS will not reach device agent. Thus, requesting new agent creation, device agent will not get feedback on existence of the personal agent. This case is successfully handled by the communication protocol (see interaction diagram “Delete agent request”).

Disregarding of existence of a personal agent with given ID, request to create personal agent will lead MAS to tend to a state with existing agent in it that will have given ID and corresponding user data.

If device agent will believe in existence of personal agent in MAS, it will send to the agent initial or updated user information. This message has a restriction: timeout to have a reply from personal agent. If after definite period of time (that will be defined depending on custom system specifics, communication environment, etc.) personal agent will not reply with information acceptance message, user will get a message from device agent regarding personal agent reachability problems. After getting such message user can decide the next steps on communication with personal agent: send items again or request agent deletion.

Summing up, results received by user (through communication device) after sending request “agent creation/change data” could be:

1) Initiated by MAS: personal agent has been successfully created, requested items have been passed to it (on user request to create agent);
2) Initiated by MAS: personal agent is already existing in the system, requested items have been passed to it (on user request to create agent);
3) Initiated by personal agent in the MAS: agent has received given items and has updated them in its database.

Summary

Object name	Description
User	Active user of the system
Com. device agent	Agent of a user that resides either in mobile device or PC of its user and can communicate with a personal agent residing in multiagent system
MAS	Multiagent system that is running on a dedicated server and consists of personal agents
Personal agent	Software agent that is managed by its user through corresponding device agent. Resides in a dedicated server and represents part of a multiagent system

Message No	Message type	Description
1	synchronous	User request that defines set of items that he wants to exchange. Defined set must be then transferred by device agent to a personal agent in multiagent system (in case of existing of such agent) or to a multiagent core with request to create personal agent.
2	synchronous	In case of absence (based on device agent believes) of a personal agent in a MAS, device agent will request its creation, providing user data.
3	return message	Response to personal agent creation request (in case if existence of agent with given ID in a MAS).
4	synchronous	In case of presence (based on device agent believes) of a personal agent in a MAS, device agent will send user items set. Return message (No 5) must be received by device agent from personal agent within definite period of time after items has been sent.
5	return message	Response to message No 4.
6	return message	Response to user request to create/update personal agent. It will be sent by device agent if personal agent will not respond within given time interval after message No 4 will be sent.
7	synchronous self message	In case of getting personal agent creation request and absence of the agent (with requested ID) in a MAS, the system will create such agent.
8	return message	Return message on agent creation operation.
9	asynchronous	Upon agent creation request and such agent (with given ID) existence in a MAS, the system will send requested user item to the personal agent.
10	return message	Response to personal agent creation request (message No 2).
11	return message	Response to create/update personal agent.

4.6.2 Deletion of a personal agent

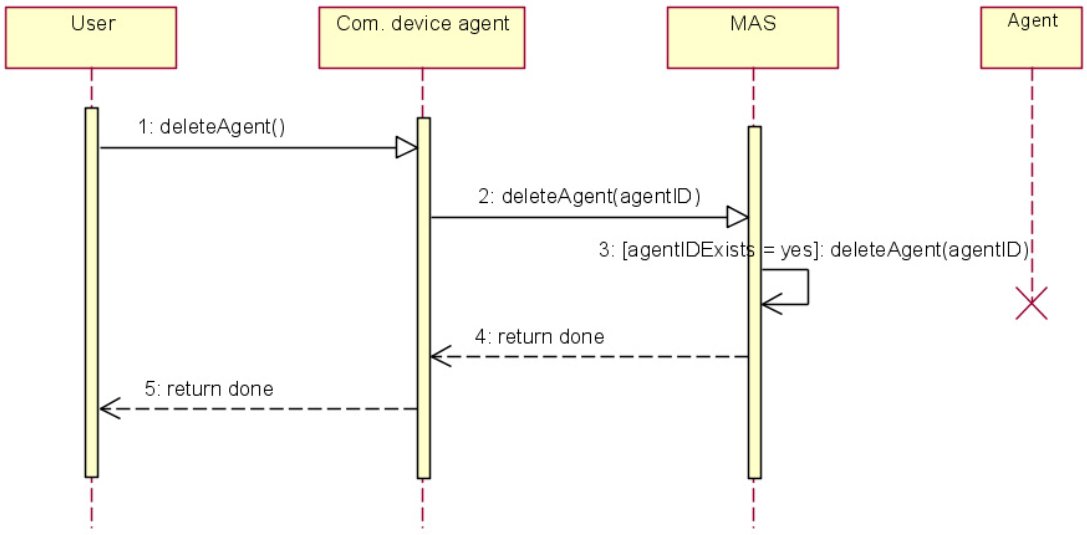

Figure 4.4 – “Personal agent delete” request

When user will want to stop using the system and thus delete personal agent in it, request to agent on communication device should be made for this. Agent on user’s communication device will send “personal agent delete” request to corresponding MAS system, specifying corresponding agent’s ID. This request will be done disregarding information (that will be stored in device agent’s database) of personal agent existence in the MAS. Thus, previous possible failure to get a response from MAS for request to create a personal agent will not affect the way how devise agent will proceed with “personal agent delete” request. Such user request will be send directly to MAS and will lead to personal agent deletion (that is shown by a cross that ends lifetime of agent on fig. 4.4).

Let’s look in more details at scenario with failure of getting response from MAS to agent creation request. Suppose user has requested creation of a personal agent with definite set of items. Agent on user’s communication device will send such request to MAS which will check for existence of agent with corresponding ID. In case of absence of such agent MAS will create it and send to it corresponding items of user. As a result, device agent must receive confirmation of agent creation. If such confirmation will not reach device agent, then both device agent and user will suppose that personal agent in MAS still doesn’t exist. In this case user can do two types of request: either create/changeRequest again or send deleteAgent() (despite of absence of confirmation of agent creation, user still will have option to send such request). Create/changeRequest will execute communication scenario shown in fig. 4.3, while deleteAgent() in this situation will cause direct request made by device agent to MAS.

If trying to reduce communications between central server and communication devices, there could be verification on user device side of personal agent existence in MAS (according to database stored in device agent) then device agent could stop user request to delete personal agent, since device agent will believe that there is no personal agent in MAS. User would be notified and misinformed by information from device agent of personal agent's absence. Meanwhile personal agent would be able to have its normal lifecycle in MAS, communicating with other agents of the system. Our protocol successfully overcomes this possible situation: device agent always sends user request for personal agent deletion to MAS.

Summary

Object name	Description
User	Active user of the system
Com. device agent	Agent of a user that resides either in mobile device or PC of its user and can communicate with a personal agent residing in multiagent system
MAS	Multiagent system that is running on a dedicated server and consists of personal agents
Agent	Software agent that is managed by its user through corresponding device agent. Resides in a dedicated server and represents part of a multiagent system

Message No	Message type	Description
1	synchronous	User request to delete personal agent.
2	synchronous	Device agent to delete personal agent residing in MAS with given ID.
3	asynchronous self message	In case of personal agent existence in a MAS, the system will delete it upon corresponding device agent request.
4	return message	Response for device agent to personal agent deletion request.
5	return message	Response for user to personal agent deletion request.

4.6.3 Computations of a chain maker

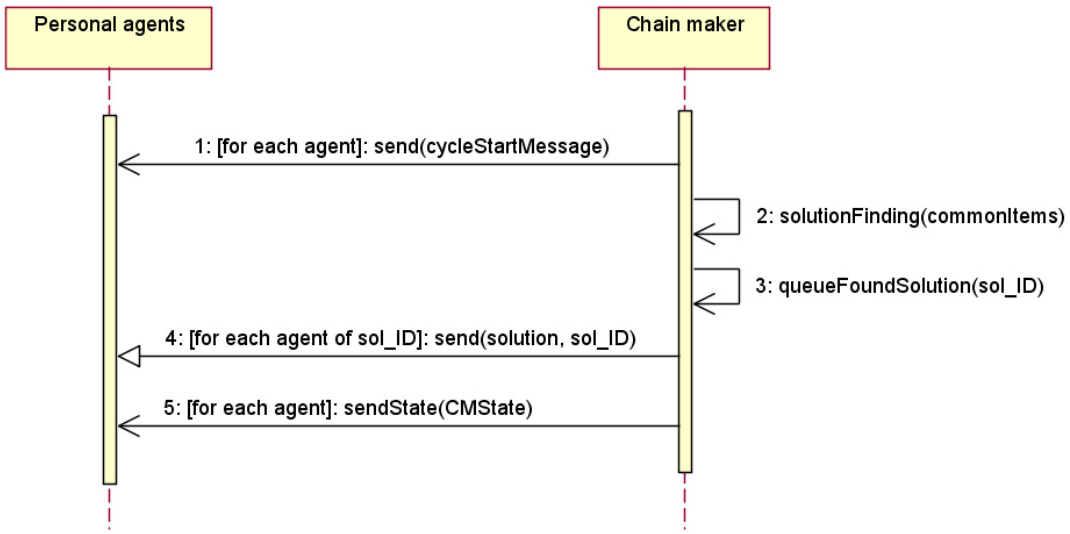

Figure 4.5 – Optimal solution computation of a chain maker

Having been defined after calculation by every agent of the system, chain maker will start computation of optimal barter chain from informing all discovered agents of its role acceptance. After this each computation cycle will be started and finished with corresponding message. Such message will be indicator for waiting agents which will be able to track state of chain maker.

After the report message chain maker will proceed to finding a solution based on current common lists of offered and desired items. Having found a solution, chain maker will put it into own queue, taking off items involved in potential solution from common lists of available and required items.

Each agent of a queued solution will be notified by chain maker of found barter chain with details on potential involvement of that agent in it. Details of participation will include sequence of agents involved into the chain, particular items that notified agent can get from preceding agents (of the sequence), items that agent have to pass to subsequent agents, user contact data and given solution's ID. That agent will then pass proposed solution to its user (see details in fig. 4.6) and get an answer back. As a result chain maker should have a common response on each queued solution – whether all users involved into corresponding solution want to make proposed barter or not (in case if at least one of them will not want to take part in proposed barter).

Queue of solutions of chain maker will consist of set of such proposed solutions that will remain without common answer from all users of corresponding solution. As soon as chain maker will get common response for a queued solution, it will update its queued solutions list.

Keeping group agents informed of its state, chain maker will notify all of them after it will propose found solution to corresponding agents. Thus, all agents will know that the chain maker will have finished computational cycle

and depending on its intentions (that will be reflected in the message) it will either continue computations or refuse from its role. Refusing from chain maker's role will cause all agents of the system to restart negotiation process from agents' discovery phase. Newly selected chain maker will start solution finding phase with empty queued solutions list, updating the list with own proposed solutions.

Summary

Object name	Description
ChainMaker	Agent that is responsible for optimal solution finding in a current system
Personal agents	Software agent that is managed by its user through corresponding device agent. Resides in a dedicated server and represents part of a multiagent system

Message No	Message type	Description
1	asynchronous	Upon ChainMaker's role acceptance and after each computational cycle, ChainMaker must inform all other agent of its community of a new computational cycle start (unless ChainMaker will not want to refuse from its role – in this case "service finished" must be sent).
2	asynchronous	The main computations on optimal barter chain finding.
3	asynchronous	Found optimal solution must be queued and stored in ChainMaker's memory until having response on the solution firing or ChainMaker's role refusal.
4	synchronous	Informing all agents of the optimal solution. Common response regarding proposed solution firing should be received later (see ch. 4.6.4).
5	asynchronous	Informing all agents of the community of ChainMaker's state after computational cycle ("cycle finished" or "service finished").

4.6.4 Solution confirmation

Having found an optimal solution for a given group, chain maker will notify each agent of the solution. Notification will contain ID and details of a solution (such as which other agents are involved into proposed optimal chain, their sequence, which items should be exchanged and corresponding contact information of users). Each personal agent (residing in MAS) will queue proposed solution by taking out correspondent available user resources and required items of the solution from user items lists (stored in agent's database). Thereafter agent will pass proposed solution to corresponding agent in communication device. This message will be limited in time to be answered. If there will be problems for personal agent to reach communication device and solution proposition could not be sent to user, then agent will consider by default unknown answer ("N/A") from its user to the current solution. This decision will be kept until personal agent will be able to get user response to proposed solution or by the moment when common decision for the solution should be given by all agents of proposed barter chain.

When agent in user communication device will get a solution, it will request its user to confirm participation in it or refuse to do it. In this phase the agent will provide user with minimal necessary information: which other users are involved into proposed optimal chain and which items should be exchanged with them. After user will give the answer, device agent will send to personal agent user decision to proposed barter chain. Personal agent will inform all other personal agents of that solution about its own user decision.

Timeout for having decision will be controlled by each personal agent after it will get a proposition from chain maker. Upon its expiration it must send to all agents of a corresponding solution its user decision.

There are 3 possible decisions that personal agent can have for a given solution:

1) Positive: if agent got a message from user with agreement on executing of proposed barter. In this case agent will do the following:

a) delete corresponding queued items

2) Negative: if agent got a refusal from its user on proposed barter. The agent will do the following:

a) restore corresponding queued items to user lists of offers and propositions

b) delete corresponding queued items

3) Unknown (N/A): if agent would not receive message from its user (in case of problems with communication or due to not giving answer by user) after timeout for a given solution. The following actions will be done:

a) restore corresponding queued items to user lists of offers and propositions

b) delete corresponding queued items

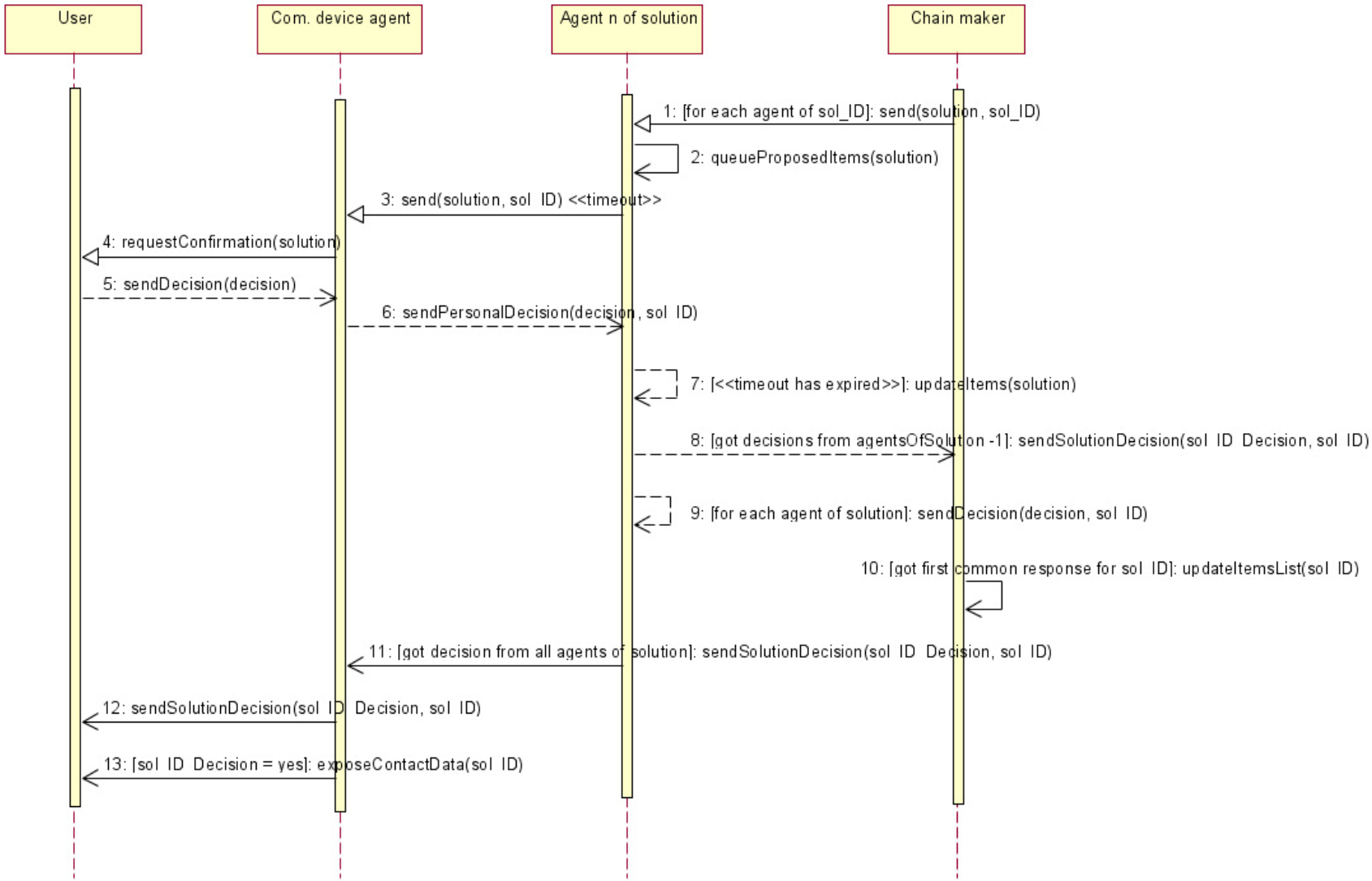

Figure 4.6 – Solution confirmation

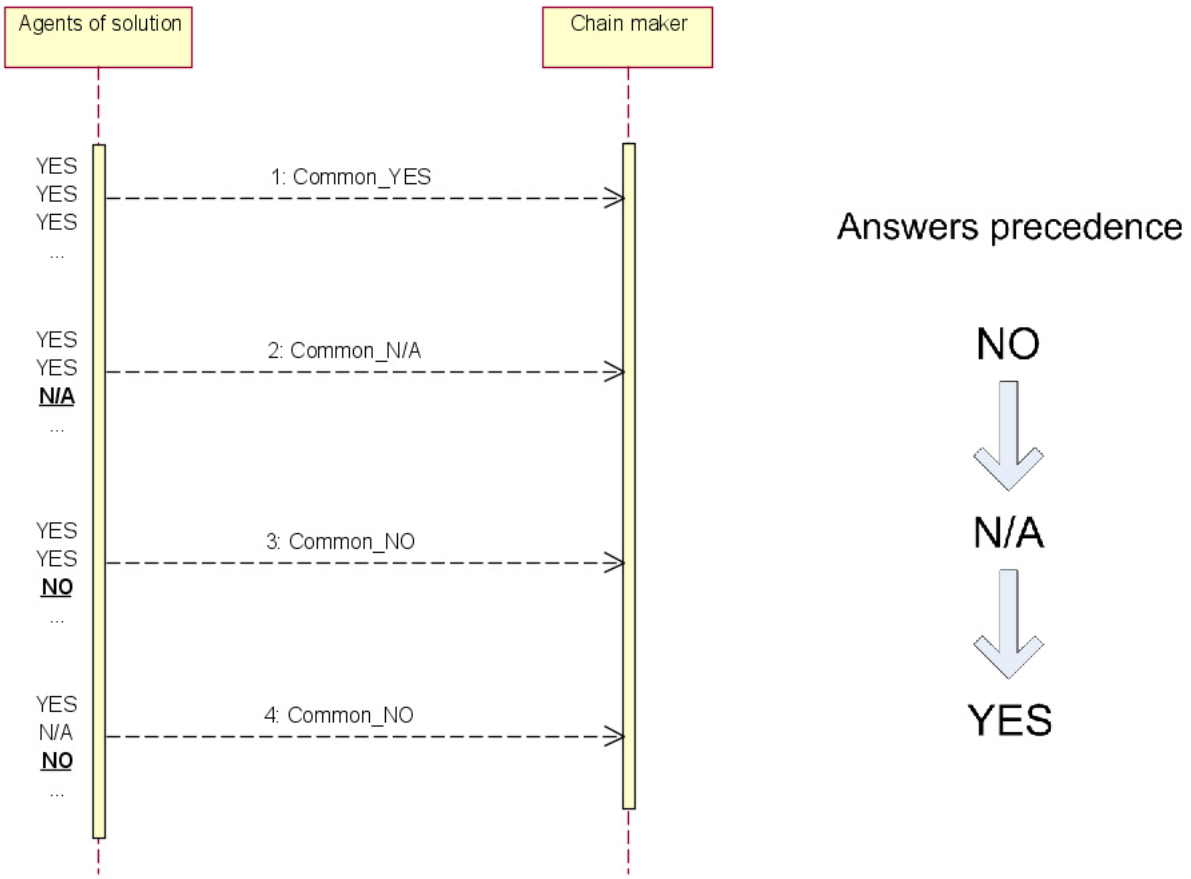

Figure 4.7 – Precedence of answers for common decision

Common decision for a solution will be joint decision of all agents of that solution considering answers precedence (fig. 4.7):

1) if any of agents will have negative answer then the common decision will be negative;

2) if there negative answers will be absent, but there will be any unknown answer then the common decision will be unknown;

3) the common decision will be positive if all answers will be of least precedent type: positive.

Due to asynchronous communications between agents, most probably there will be only one agent that will have responses from all other agents and will not send its own response by that time. That agent will be the first one in a barter chain which will know common decision, whether all users of the chain want to execute it or not. This agent will send common decision for proposed barter chain to corresponding chain maker and its response to other agents of the solution. Chain maker will react to a common decision for proposed solution in similar manner to personal agent, except operations with common items lists (not personal) and additional action: on common refusal it will add the solution for refused list (see fig. 4.8). Other agents will get the response and will see that there is already agent of their barter chain that should notify chain maker of a common decision, so all other agent will

not try to contact chain maker on that solution. This is done to simplify negotiation between agents and reduce inter-agent communications.

Even if there will be more than one agent that will try to report common decision on a given solution, chain maker will consider the first message and disregard all subsequent messages on the same solution.

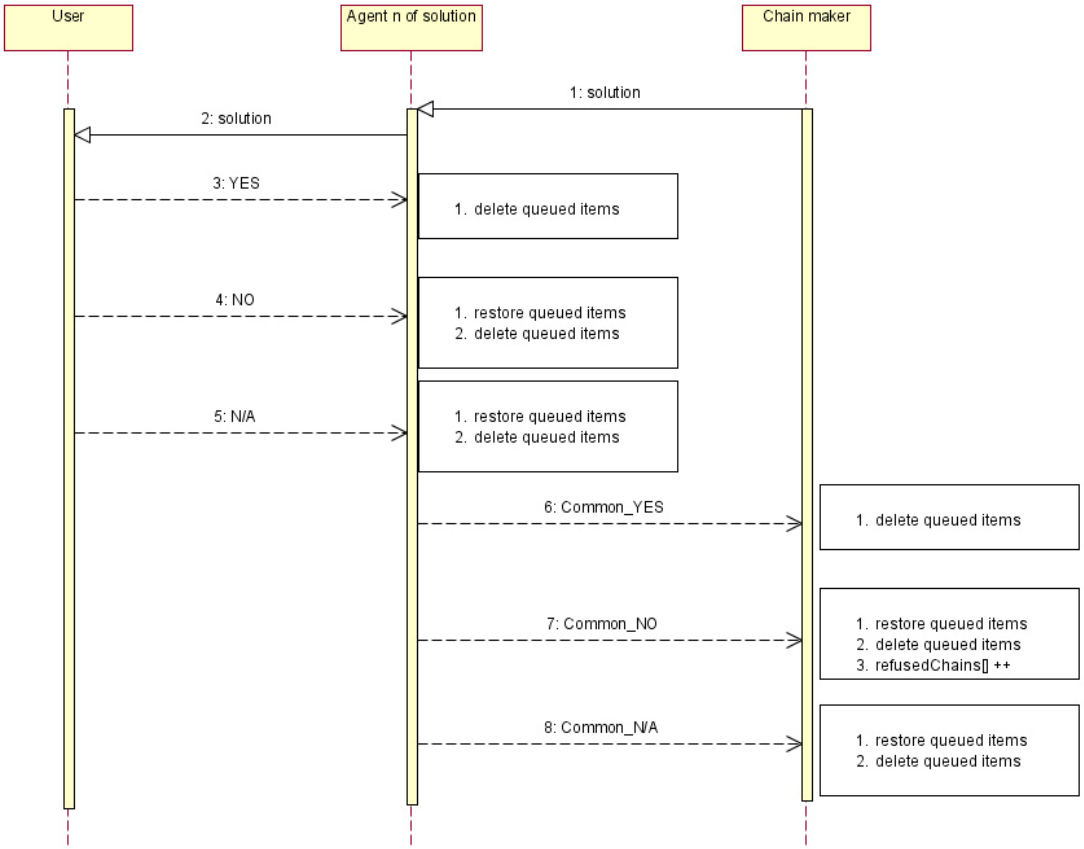

Figure 4.8 – Reactions of agents on solution acceptance decisions

Negative decision for a given solution will make chain maker restore those items involved into proposed barter chain and put the chain into refusedChains[] while positive decision will still keep corresponding items outside of common items lists (see chapter 4.5.3) and will not be considered by the chain maker in finding of next optimal solutions. Thus even if agents of the solution will come to agreement and will not be able to tell about it to their chain maker, it will not affect system's functionality: barter chain can be executed while chain maker will correctly search for other optimal barter chains with remaining items.

ID of a barter chain will be a sequence of agents IDs involved into the chain. This helps to successfully handle situation when chain maker will accidentally finish its service after it will propose an optimal chain. Any new chain maker that will receive common decision message will be able to put that barter chain into refusedChains[] (in case of negative decision) or it will do nothing if the chain's users will accept proposed barter.

This demonstrates soundness of our protocol to tolerate possible communication problems between agents of multiagent system.

Summary (solution confirmation)

Object name	Description
User	Active user of the system
Com. device agent	Agent of a user that resides either in mobile device or PC of its user and can communicate with a personal agent residing in multiagent system
Agent n of solution	Set of agents that are involved into proposed optimal solution
Chain maker	Agent that is responsible for optimal solution finding in a current system

Message No	Message type	Description
1	synchronous	Informing all agents of the optimal solution.
2	asynchronous	Deletion of proposed items from corresponding user items lists.
3	synchronous	Request to confirm participation in a proposed barter chain. Limited by definite period of time to response.
4	synchronous	Informing of user of proposed barter chain.
5	return message	Response of user to the proposed barter chain with decision regarding participation in it.
6	return message	Response to corresponding personal agent user decision on his participation in proposed barter chain.
7	return message	Upon given timeout expiration (or having corresponding user decision) personal agent will update user items lists and will set the solution decision from its user (either that of user or default one if user response will not be received after timeout).
8	return message	Upon having solution decisions from all other agents of the solution, but having not yet send own (user) decision to them, personal agent will infer whether proposed solution's common decision is positive or negative. That common decision will be sent to a ChainMaker (as a response to message No 1).
9	return message	Sending of user decision on the solution to other agents of the solution.
10	asynchronous	Upon having a common decision on proposed solution, ChainMaker will update its queued chains list.
11	asynchronous	Upon having common decision, personal agent will send it to corresponding device agent.
12	asynchronous	Informing of user on common decision.
13	asynchronous	In case of positive decision user will get necessary contact data of other users to make proposed barter.

4.6.5 New agent arrival

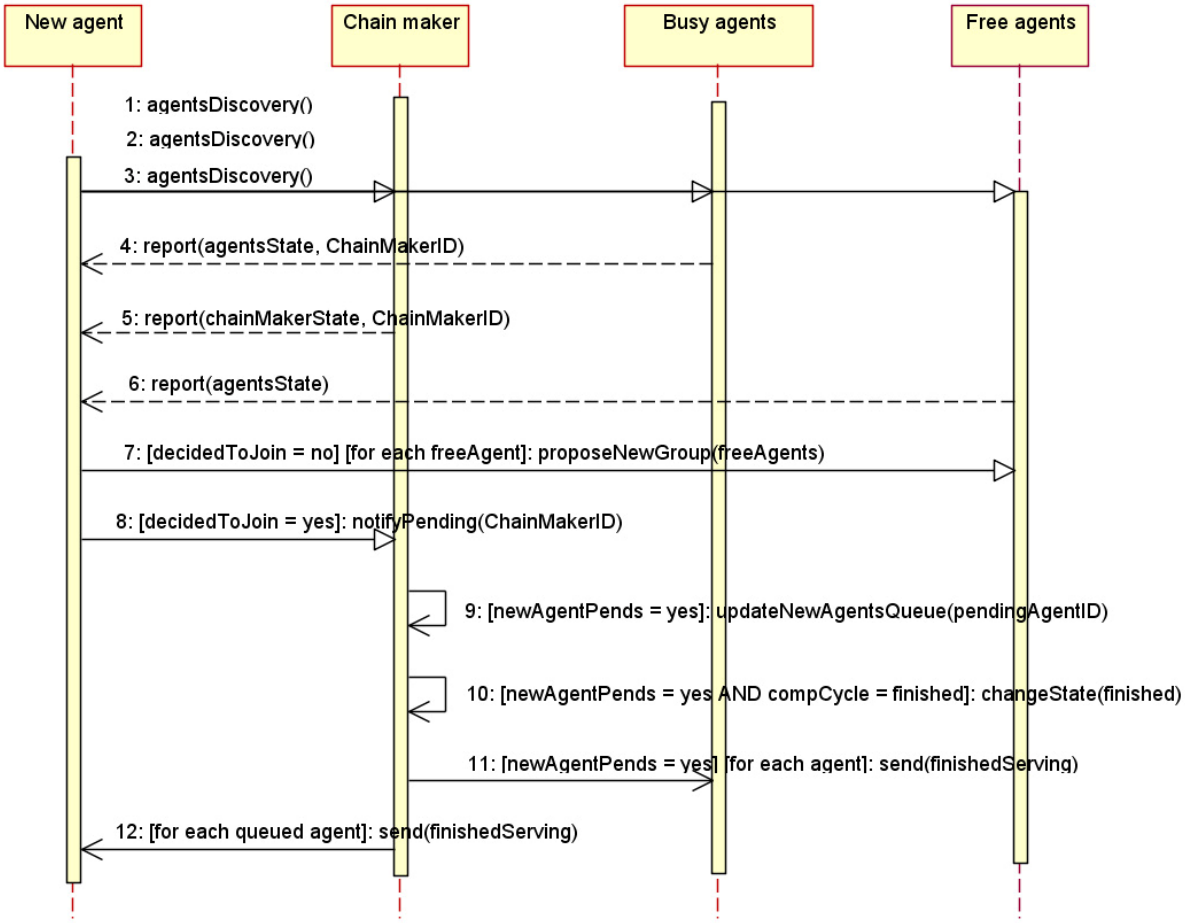

Figure 4.9 – New agent arrival

In our architecture there is a central registry for all agents of a given MAS. Each newly registered agent will be able to get a list of already registered agents in the MAS. All the time each software agent will be active in the MAS it will provide discovery service for other agents. This means that any agent, knowing reference ID of other agent in the system, can send a message to that agent asking for its current state. Whatever active agent will do, it will respond to such message providing agent discovery service.

For new agent other active agents involved into process of optimal solution finding will have 3 roles: ordinary agents, chain makers and free agents. Ordinary agents must answer to discovery request reporting their state (“waiting”) with ID of their chain maker. Chain makers will report their state (“processing data”) and own ID which can be further used by new agents as reference to existing communities. Free agents will report their state only (“free”). Each message sent from one agent to another by default will contain standard low-level communication information (such as sender, recipient, etc.).

Having information on state of surrounding agents, new agent will have to decide whether it will join existing group or it will try to organize new group of agents (in the same MAS) that will potentially consist of free agents (for the moment of discovery phase for the new agent). For details on chain maker's selection and decision making see chapter 4.5.1.

If new agent will decide to create a new community, it will make such proposition to all found free agents. If all agents will accept such proposition, new group will be created and chain maker selected.

If new agent will decide to join existing group of agents then it will send notification to selected chain maker of its desire to joint corresponding group. In case of such notification the chain maker will update its queue of waiting agents. Upon completion of ongoing computational cycle the chain maker will check its queue of waiting agents. If it will be non empty, then the chain maker will change its state to "finished serving" and send a corresponding message to all agents of its group and to all waiting agents. After this all notified agents and chain maker will start agents' discovery procedure.

Summary

Object name	Description
New agent	Agent trying to find appropriate community to joint it (with a state of free agent)
Chain maker	Agent that is responsible for optimal solution finding in a current system
Busy agents	Agents that are already in established community (are in a process of optimal solution finding)
Free agents	Agents that are active in a MAS but are not involved in any of agents' communities and are trying to find their appropriate community

Message No	Message type	Description
1	synchronous	Discovery of surrounding active agents in a current MAS.
2	synchronous	Discovery of surrounding active agents in a current MAS.
3	synchronous	Discovery of surrounding active agents in a current MAS.
4	return message	Response of busy agents with notification of their status and their current ChainMaker's ID.
5	return message	ChainMaker's response with notification of its status and own ID.
6	return message	Free agents response.
7	synchronous	In case is new agent will decide to not join to any existing agents' community, it will send proposition to all discovered free agents proposition to create a new community.
8	synchronous	If new agent will decide to join to one of existing communities, it will notify

		corresponding ChainMaker.
9	asynchronous	In case if ChainMaker will get a notification of at least one new agent's desire to joint to ChainMaker's community, it will update its queue of pending agents.
10	asynchronous	Provided that computational cycle of ChainMaker is finished and presence of at least one agent pending to joint ChainMaker's community, it will finish its service.
11	asynchronous	Provided that at least one agent pending to joint ChainMaker's community, ChainMaker will send to all agents of the community notification of service finishing.
12	asynchronous	After finish of its service, ChainMaker will notify pending agent of this fact, so it can start negotiation process with other agents of community.

4.7 Sample scenario

To see how the BarterCell's algorithm works let's look at a live example.

In the following example for simplicity we have used designation of item as a numeric value. Depending on a context the same item can be represented as a desired item (for example for item "5" we have D_5) and offered item (O_5).

In our example we have a user that wants to get rid of his belongings (named O_1, O_2 and O_8) and wants to have an item called D_3 (that is item of Strict demand for that user). If nobody will have a D_3 then user is agreed to have D_5 or D_6 instead of his belongings (group of flexible demands of the user). Moreover, using ontology the user has specified that he has specific areas of interest. Those areas are represented in this case with item D_9 (fig. 4.10).

All these items, their descriptive information and other relevant user data will be stored in a device agent. That agent will try to pass this information to a corresponding personal (residing in a multiagent system). Let's call this agent A_1.

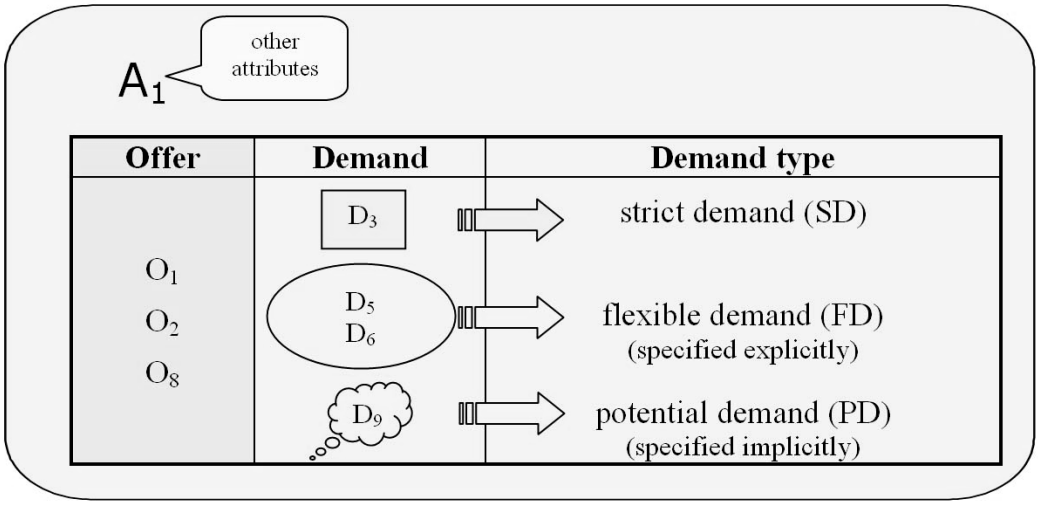

Figure 4.10 – Sample agent with own O- and D-lists

After personal agent will accept user request, it will start broadcasting to find other agents in its environment. Suppose, it has found other 5 agents: A_2 – A_6. Those agents have their own sample item lists (see fig. 4.11).

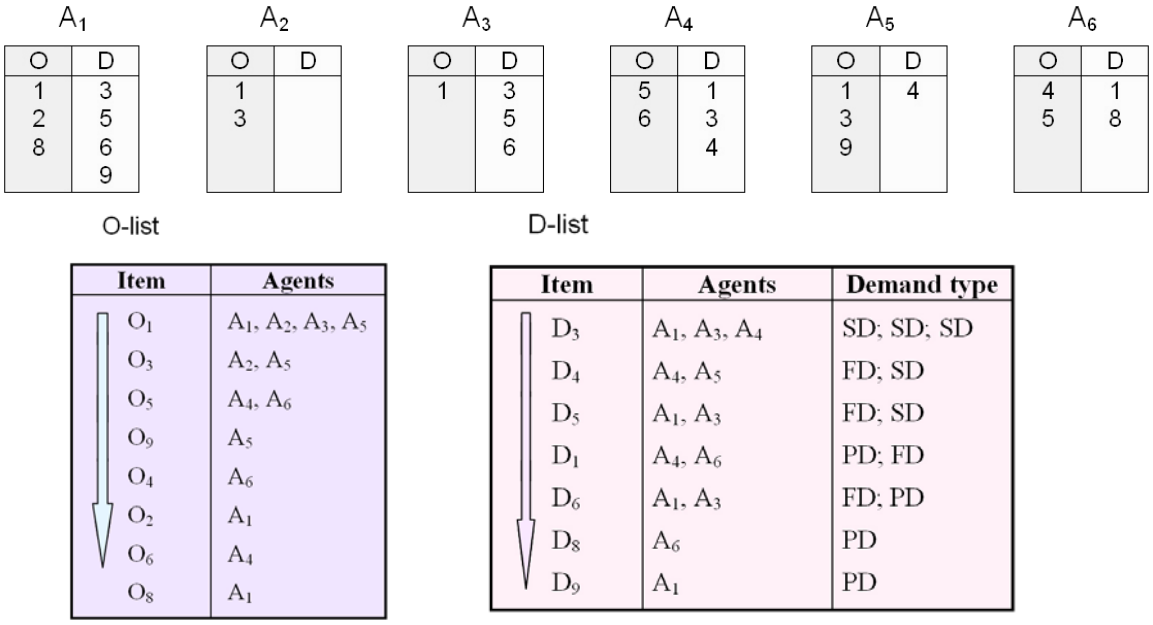

Figure 4.11 – Chain maker's designation

Each agent will send its own items list to all other discovered agents (suppose, that all agents are in a close community, that is list of discovered agents is the same for every agent of the community – for details see chapter 4.5.1). Based on information of items lists of each agent in the community, every agent will compose 2 lists of demands and propositions that will be common for the community.

List of propositions (O-list) will contain items and corresponding agents proposing it. The list will be sorted in descending order of amount of agents proposing items.

List of demand (D-list) will contain items that are demanded by agents of the system with corresponding demand type for each demanding agent. The list will be also sorted in descending order with consideration of agents' amount and demand type (in case of equality of agents' amount).

Looking at the common D-list, each agent will notice that item "3" (D_3) is in a greatest demand among all agents of the given system. According to our algorithm this means that agent who can provide with this item will most probably be involved in optimal barter chain. Looking at O-list every agent will find the first agent that can offer item "3" – it's A_2 in our case. We called such agent ChainMaker (for selection details see chapter 4.5.2).

Having finished standard preprocessing procedure, every agent of the system will know which agent is a ChainMaker at a given moment. ChainMaker will accept its role in the system and will start searching for an optimal barter chain (fig. 4.12).

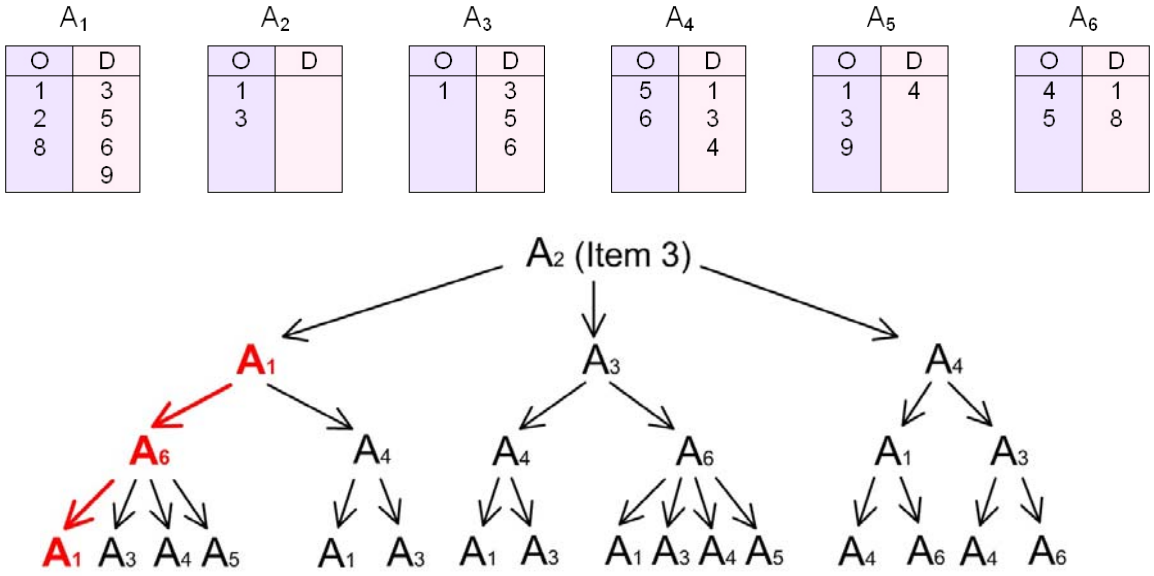

Figure 4.12 – Candidate chain selection

Searching for optimal barter chain, A_2 will start building a barter tree. Barter tree represents all possible barter options in a given agents' community.

Root of the tree will be an agent that will have the most chances to be in the optimal chain at the moment of start of the tree building process. At the beginning it will be A_2, but then, depending on which items will be exchanged, another agent can be placed into the tree root (and that tree can be still built and processed by already selected ChainMaker – A_2 or another ChainMaker if A_2 will then refuse from its role).

Firstly the tree will be built considering only Strict demands (for demand types see chapter 4.2) of agents. After this Strict + Flexible and then Strict + Flexible + Potential demands will be considered.

Let's suppose, that it was impossible to find any barter considering only Strict demands and tree of with demand types Strict + Flexible should be built.

Root agent in the tree will have (for the tree) most demanded item – in our case it's "3". Its children (A_1, A_3 and A_4) have in their own demands list item "3". From this step all subsequent nodes (representing agents) will consider all offered items matching at least one of item in somebody's demands list. In case of matching demanding agent will become a child with parent as an offering agent. In our example A_1 can offer items "1", "2" and "8". At least one of them is required by A_6 and A_4 which become children of A_1.

As practice showed it will be enough to have a tree of depth 3 or 4 to find and optimal chain. Such chain can involve up to 3 or 4 agents that will be able to mutually satisfy their needs by exchanging by their items. More agents in a chain (and thus, more corresponding users in a real life) will be ineffective from a practical point of view. Moreover, tree of depth more than

4 will require significant computational resources (as we showed in our results).

Having built a tree up to defined level, ChainMaker will then analyze it by searching for the shortest closed barter chain. In our example (fig. 4.12) agents A_1 and A_6 compose such chain (according to the tree they can barter in this way: A_1-> A_6-> A_1).

The shortest closed barter chain will be a candidate for the optimal barter chain. It will be then checked for conformance of barter parameters of all agents in that chain. Such parameters will include validation of:

1) coincidence of a barter point;
2) desired maximal barter chain length;
3) proper prices of offers and requests between found agents;
4) expiration time of all items in the chain.

These parameters will be set by each user before he can make a request to his personal agent. The chain will be considered valid if each connection (A_1 -> A_6 or A_6 -> A_1) will have valid at least one barter alternative (fig. 4.13).

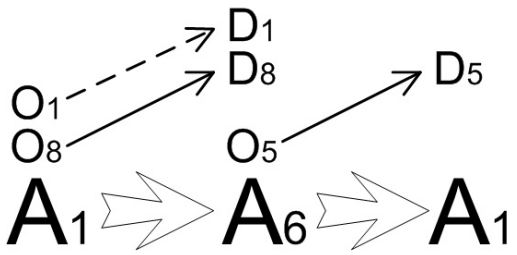

Figure 4.13 – Chain validation

In our example barter of item “1” is not valid between agents A_1 and A_6, but they can barter with items “8” in exchange for “5”.

Suppose, next tree (of demand types Strict + Flexible + Potential) will result into a shortest valid chain A_1-> A_4-> A_1. The ChainMaker will have to choose the optimal chain among shortest valid ones. In our case we have the following situation:

- SD tree: no chains
- SD+FD tree: A1-> A6-> A1
- SD+FD+PD tree: A1-> A4-> A1

Since chain A1-> A6-> A1 was the shortest valid one and considering demand types precedence (Strict > Flexible > Potential), it will become an optimal barter chain in current operational cycle of a ChainMaker. Having defined the optimal chain, ChainMaker will inform all agents in the chain (A1 and A6) that their users can potentially barter between them, having the most optimal barter option in the system for that moment. After this agents A1 and A6 will ask their users whether they will accept proposed barter option or not (having declared that user wants to change something will not

obligate to do it, so he can refuse to barter if proposition will be unacceptable for him).

Having reported to all agents of the optimal chain, ChainMaker will report to all known agents (A_1, $A_3 - A_6$) of a computational cycle finish (if A_2 will decide to be a ChainMaker in the next cycle) or service finish (if A_2 will decide to refuse from its role of a ChainMaker).

On the next computational cycle ChainMaker will operate with modified common O-list and D-list.

5. Experimental results

5.1 Testing platform

As a server we have used PC with AMD Athlon 64 X2 Dual CPU, 2.61 GHz, 3.43 GB of RAM and Microsoft Windows XP SP2 installed on it. Java virtual machine was of version 1.6.0_03 and Eclipse SDE of version 3.3.1.1. As a multiagent platform we have taken JACK 5.0.

To test connectivity of personal agent and device agent we have selected BlueTooth technology and its implementation in a BlueCove java library [31]. To test connectivity we have used 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 with the maximal bit rate support of 723Kbps, and the possibility to concurrently connect up to seven Bluetooth-enabled devices. On the end-user side we have used Nokia 6630 mobile phone with a client part of the application preinstalled in it.

5.2 Analysis of results

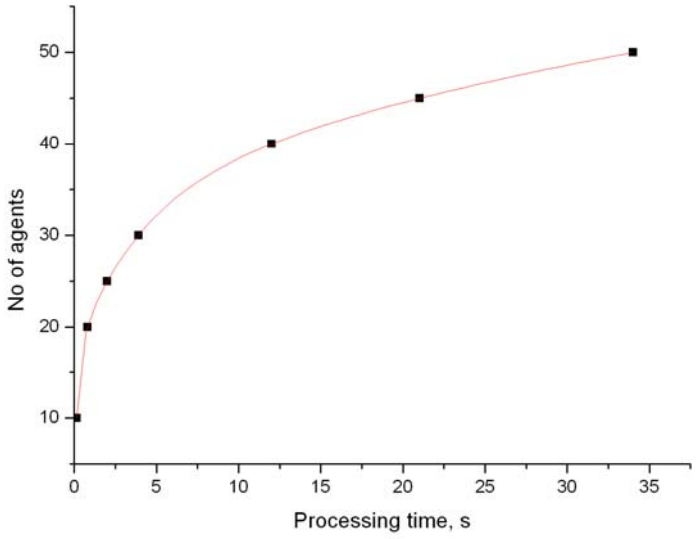

Figure 5.1

Figure 5.1 shows how fast the chain maker will finish searching for all possible optimal chains depending on total number of agents known to it. During this simulation we put the number of O-items to 5 and the number of D-items to 15.

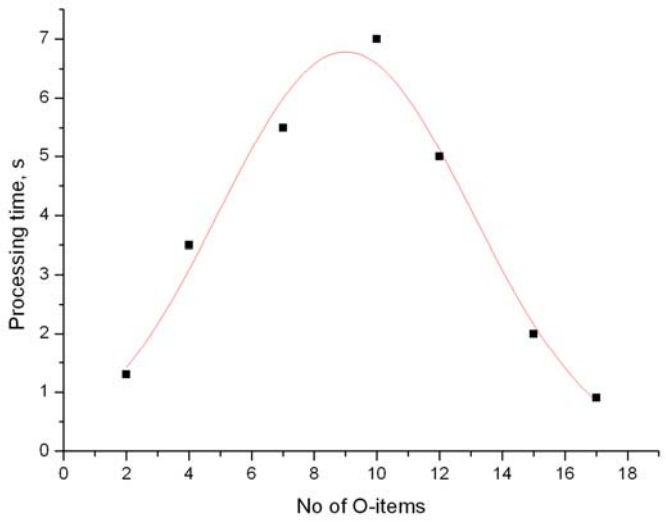

Figure 5.2

Figure 5.2 shows how fast the chain maker will 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 chain maker has the biggest number of possible exchange combinations.

During the simulation we have used the following parameters: max number of items: 20, max number of D-items: 15, number of agents: 30.

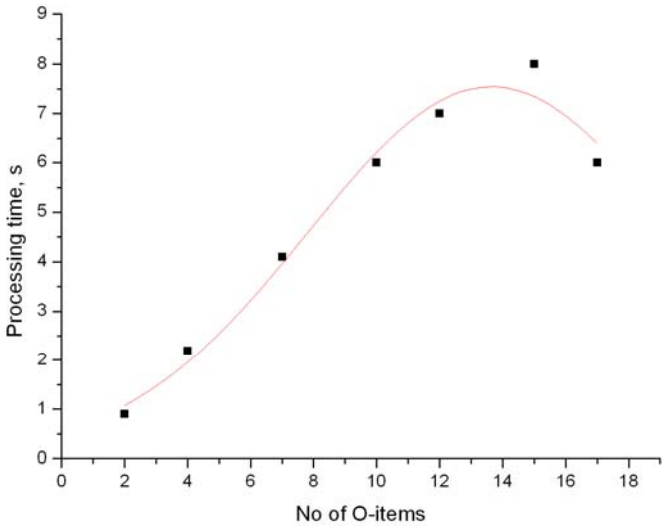

Figure 5.3

The same as previous graph with altered maximum total number of offered + proposed items.

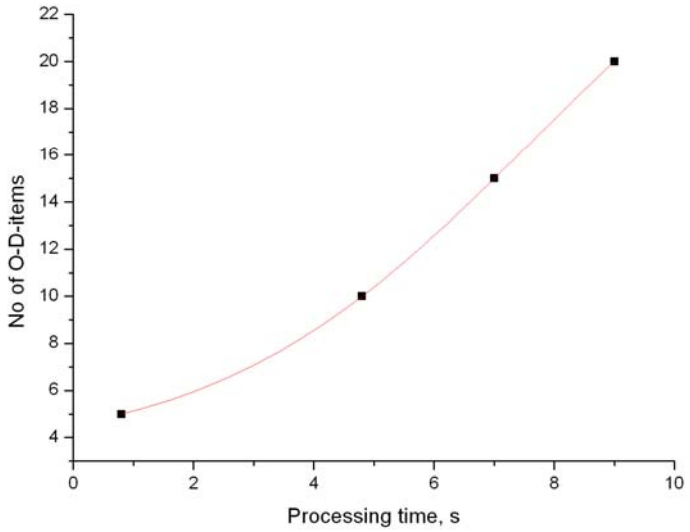

Figure 5.4

Figure 5.4 shows how fast chain maker will finish searching for all possible optimal chains depending on number of items at each known agent. In this case we put number of D-items = number of O-items, number of D-items are only of ”Strict” type and number of agents is 30.

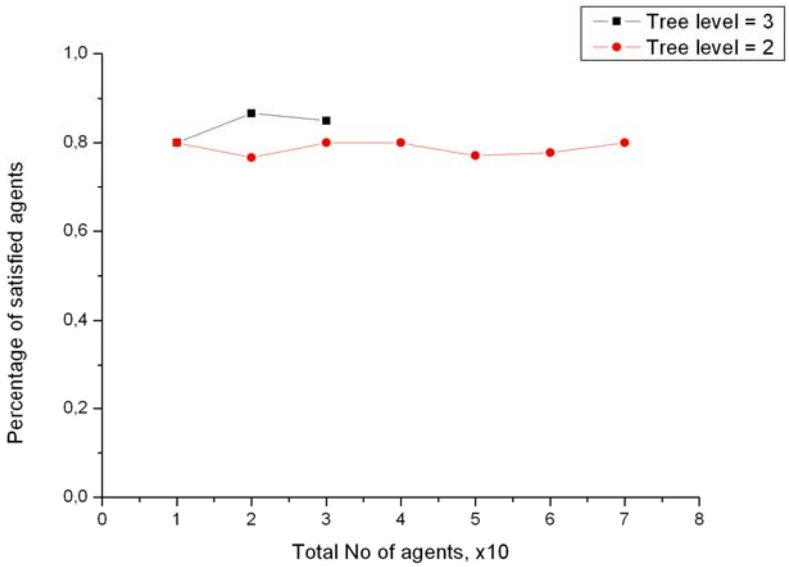

Figure 5.5

Figure 5.5 represents how many agents would be satisfied (i.e. involved in one of optimal chains produced by chain maker) until the chain maker finishes all possible chain-building processes. Depending on the trees level (depth) the percentage of satisfaction will vary. Here we put number of D-items to 20 and number of O-items to 5.

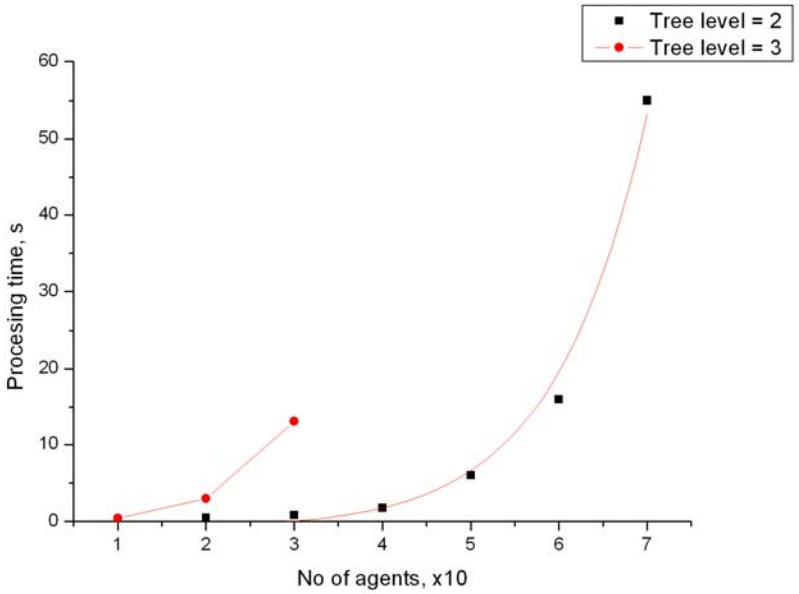

Figure 5.6

In Figure 5.6 we show processing time representation for simulation of agents' level of satisfaction (fig. 5.5).

5.3 Performance comparison with adopted Strategic negotiation model

We have chosen to compare or protocol with the Strategic negotiation model [3] because of its approaches to address problems encountered in distributed data networks that are likely to occur in dynamic mobile environments. The model uses Rubinstein's approach of alternating offers [29, 30]. The strategic negotiation model is divided into three parts which all together form a best case scenario for allocating data in a distributed network. A negotiation protocol where it is assumed that one of the involved agents make an offer to organize the shared task, other agents respond with acceptance, rejection or opting out of the negotiation process. The algorithm terminates in one of the two cases:

1) if all agents accept the proposed solution;

2) if one or more of the agents decide to opt out.

The second part of the model is about the complex utility function that considers factors like storage cost, retrieval cost and distances between servers, which can all be similarly addressed in mobile environments after replacing the concepts of capable servers with limited mobile devices. The third part is about the negotiation strategy of each agent which specifies the actions to be taken in different scenarios.

In the strategic model, there are number of agents $N=A_1 \ldots A_n$, and they are supposed to reach an accepted outcome on a delegated task within certain

period of times T = 0, 1, 2. At each time slot t of the overall process, the algorithm considers the results previously obtained to decide whether to allow another involved agent to make a new offer. The protocol keeps on looping until an offer is accepted by all agents and the proposed solution is then put into practice. Assume that agents involved in a single goal achievement scenario are not informed with the reactions of the others while responding to an offer. If one of the agents rejects the proposed solution but yet none have opted out, the negotiation process moves forward to period t+1.

The strategic negotiation model was built on such assumptions:

1) all of the concerned agents interact according to predefined design norms;

2) agents are not pleased with opting out of any negotiation session and they get committed to whatever results agreed upon;

3) each negotiation process is an independent action that does not relate to any other future or previous processes.

Since we have chosen Strategic negotiation protocol as a benchmark to evaluate and measure the performance of our algorithm with respect to existing ones, we have made some slight customizations for it in order to adjust it to the minimum requirements of our application and thus to make it comparable to our protocol. We made each involved agent seeks to match all of its O-items with other agent's D-items. The adopted algorithm finishes when each concerned agent has defined its completing agent(s), which it can make the barter with (a chain of services exchange with a maximum length of 3 agents).

Table 1. Comparison of performance of adopted Strategic Negotiation algorithm and BarterCell.

No of agents in a chain	Strategic negotiation algorithm	BarterCell algorithm
2	$O(n^4)$	$O(n^2)$
3	$O(n^7)$	$O(n^3)$
4	$O(n^8)$	$O(n^4)$
5	$O(n^9)$	$O(n^5)$

The table compares our solution with the Strategic negotiation model which made it easy to see how great difference in number of required interaction between agents is there. Here we have shown theoretical comparison of performance of our solution with performance of solution implementing Strategic negotiation protocol. For example, for creation of a chain of length 3 it must make $O(n^7)$ interactions between agents of a given system, (n^2) (n-2) (n-2) (n-1) (n-1) (n) where:

$O(n^2)$ agents must communicate with each other to exchange their lists of resources;

O(n-2) agents will communicate with their succeeding peers regarding resources that they can obtain from their preceding peers;

O(n-2) if there will be at least one chain of length 3, then during this communication agent, that initiated chain of resources giving will get back information of how its chain can be executed;

O(n-1) that agent will report to every agent in the chain about coalition formation availability;

O(n-1) every agent received message of the coalition formation availability will decide whether it will be in the coalition or not; decision will be sent to every agent of prospective coalition;

O(n) all agents will send message that will initiate their chain.

We have compared results of this protocol implementation with those received after simulation of our (tree-based) algorithm that searched for first optimal chain of length 3. During the simulation we have put number of D-items to 15 and number of O-items to 5. Comparative results of our simulation see at fig. 5.7.

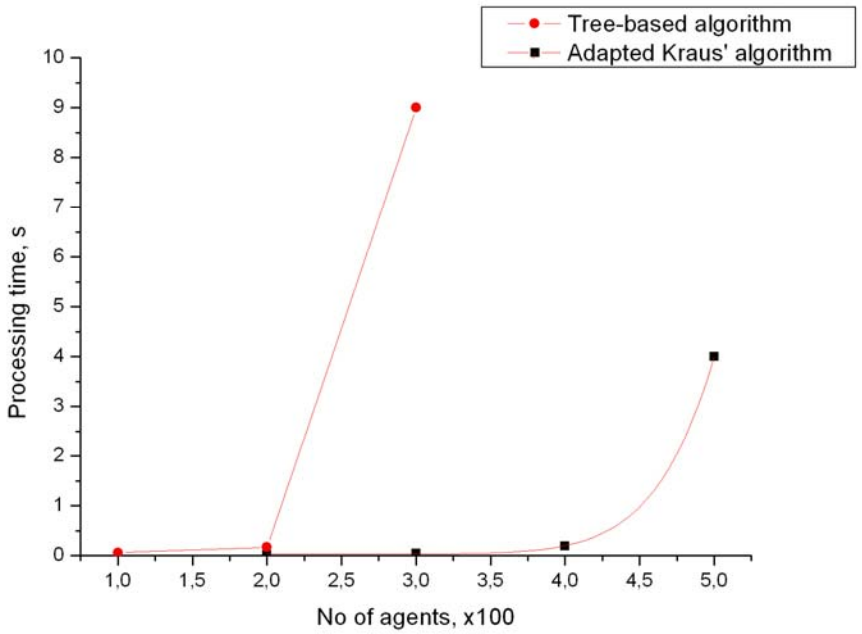

Figure 5.7

We used JDots for tree building, which is an object oriented software component where each node of a tree represents an object with its own fields and methods. Our algorithm works slower with a huge amount of agents (more than 300) because the chain maker needs to build 3 of such object-based trees. But nevertheless, simulating with less than 200 agents these two algorithms give similar results in time, having variations in quality of results

and further improvements are foreseen (work with object trees vs. dataset of matching agents' pairs).

6. Conclusion

In this work we have introduced the system that can wisely implement solution for existing problem – optimal distribution of available resources in a social network. In particular we have presented our approach that is based on multiagent systems. We have developed communication protocol for software agents that will simulate user behavior in the community. Our algorithm helps to discover options of resources distribution in the community. It was tested and analyzed with different parameters that can reflect real-life situations with limitations of time and computational resources. Autonomy of agents, their ability to represent user behavior in real life and their intelligence (that is defined by implemented algorithm) allow to have the most efficient barter between users of the system.

Efficiency of our negotiation model was compared with results of simulation of the most famous negotiation protocol – Strategic negotiation. Though theoretical negotiation model of the BarterCell beats Strategic negotiation protocol, we have received the opposite practical results. That was because to build solution trees we have used a complex object structures (JDots) in our implementation. Those structures require huge computational resources because each node of the solution tree represents an object with its own fields and methods. This can be potentially used to represent an agent as a node. As an alternative another (custom) data structure will be required to overcome problem of lack of computational resources. Such complex object representation can be still useful in future works that will also incorporate search for another communication technology.

In our architecture we have used BlueTooth for communication of pocket devices with a multiagent system. The range of a Bluetooth connection is up to 20 meters, which make obligatory the presence of users within certain area. Moreover, slow data transmission rate and authentication instability brought problems to a desired functioning of the system.

Utmost goal of our future work is to make a pure peer-to-peer pocket devices-based system without central server. This ambitious task will require development of a mobile multiagent framework that will consider recent requirement of dynamic mobile environment.

Bibliography

1 J. Walker. *Mobile Information Systems*. Artech House Publishers, June 1990.

2 M. Wooldridge. *An Introduction to MultiAgent Systems*. John Wiley & Sons, 2002.

3 S. Kraus. *Strategic Negotiation in Multiagent Environments*. The MIT Press, September 2001.

4 R. Stair and G. Reynolds. *Fundamentals of Information Systems*. Course Technology, January 2007.

5 R. G. Smith. *The contract net protocol: High-level communication and control in a distributed problem solver*. IEEE Transactions on Computers, C-29(12):1104-1113, December, 1980.

6 S. Kraus. *Negotiation and cooperation in multi-agent environments*. Artificial Intelligence journal, Special Issue on Economic Principles of Multi-Agent Systems, 94(1-2):79-98, 1997.

7 J. S. Rosenschein and G. Zlotkin. *Rules of Encounter: Designing Conventions for Automated Negotiation among Computers*. The MIT Press, 1994.

8 P. Cramton, Y. Shoham and R. Steinberg. *Combinatorial Auctions*. The MIT Press, January 2006.

9 J. von Neumann and O. Morgenstern. *Theory of Games and Economic Behavior*. Princeton University Press, 1980.

10 Bangqing L., Yulan M. *An Auction-based Negotiation Model in Intelligent Multi-agent System*. In proceedings of the International Conference on Neural Networks and Brain (ICNN&B'05), Beijing, China.

11 Howard Raiffa. *The Art and Science of Negotiation*. Belknap Press, 1982.

12 R. F. Sproul and D. Cohen. *High-level protocols*. Proc. IEEE, vol.66, pp. 1371-1386, Nov. 1978.

13 J. Leslie King and K. Lyytinen. Information Systems: The State of the Field. Wiley, May 2006.

14 M. Beer, M. Invemo, M. Luck, etc. Negotiation in Multi-agent System. Knowledge Engineering Review, 14(3):285-289, 1999.

15 N. R. Jennings, P. Faratin, A. R. Lomuscio, etc. Automated Negotiation: Prospects, Methods and Challenges. International Journal of Group Decision and Negotiation, 10(2): 199-215, 2001.

16 Benameur H., Chaibdraa B. and Kropf P. Multi-item auctions for automatic negotiation. Information and Software Technology, Volume 44, Number 5, 1 April 2002, pp. 291-301(11).

17 P. Klemperer, Auction theory: A guide to the literature, Journal of Economic Surveys 13 (3).

18 Agorics Inc., Going Going Gone! A Survey of Auction Types, available online at http://www.agorics.com/new.html.
19 P. Milgrom, Auction Theory for Privatization, Cambridge University Press, 1998.
20 A. Chavez, P. Maes, KABASH: An agent marketplace for buying and selling goods, in: 1st Int. Conference on Electronic Commerce, IECE98, Soeul, Korea, 1998.
21 O. Bucur, P. Beaune, and O. Boissier. Representing context in an agent architecture for context-based decision making. In Proceedings of the Workshop on Context Representation and Reasoning (CRR'05), Paris, France, 2005.
22 M. Bombara, D. Cali, and C. Santoro. Kore: A multiagent system to assist museum visitors. In Proceedings of the Workshop on Objects and Agents (WOA2003), Cagliari, Italy, pages 175-178, 2003.
23 Ben Coppin, "Artificial intelligence illuminated", 2004.
24 S. Abdel-Naby, S. Fante, and P. Giorgini. Auctions negotiation for mobile rideshare service. In The 2nd International Conference on Pervasive Computing and Applications (ICPCA07), Birmingham, UK, July 2007.
25 V. Bryl, S. Fante, and P. Giorgini. Toothagent a multi-agent system for virtual communities support. In the 8th International Bi-Conference Workshop on Agent-Oriented Information Systems (AOIS'06), Japan, May 2006.
26 M. Clark. JACKTM Intelligent Agents: An Industrial Strength Platform, chapter 7, pages 175-193. In Multi-Agent Programming, ed. Rafael H. Bordini, 2005.
27 FIPA TC Communication. FIPA ACL Message Structure Specification. Technical report, Foundation for Intelligent Physical Agents, Geneva, Switzerland, December 2002.
28 T. Finin, R. Fritzson, and D. McKay. A language and protocol to support intelligent agent interoperability. In The Proceedings of the CE&CALS Conference, Washington, USA, June 1992.
29 Rubinstein A. 1982. Perfect equilibrium in a bargaining model. Econometrica, 50:97-109.
30 Rubinstein A. 1985. Abargaining model with incomplete information about time preferences. Econometrica, 53:1151-1172.
31 Java library for Bluetooth connectivity, http://bluecove.sourceforge.net/
32 Celeste C., Directory Facilitator and Service Discovery Agent, contribution to the white-paper "Agents in Ad Hoc Environments", 2002.

Attachments

Attachment 1. Barter Service

Algorithm 1: Barter Service

```
/*
Input parameters: runAgent(void)
Variables used:
registered_agents list of registered agents in current MAS. MAS supplies
                  each requesting agent of the system with such list on its
                  request get_all_registered_agents()
agreedAgentsList set of agents that commonly accepted all agents of the list
                  as a new group
cDList            list of demands for all agents of a given system
cOList            list of offers for all agents of a given system
ChainMaker        ID of an agent that will make barter chains
currentMDItem     currently most demanded item in a system
currentAgent      ID of an agent running the instance of a code
agentsList        list of all active agents in a given system and their
                  states: - “waiting + ChainMakerID” – for ordinary agents
                          - “processing data + ChainMakerID” – for chain maker
                          - “free” – for the rest of active agents
notRespondingAgents list of agents that did not respond to request of
                  current agent
dG                set of agents which offers currently most demanded item
oS                set of agents which demands currently most offered item
optimalChain      set of agents that are able to make an optimal
                  barter chain in a given system at particular time
itemsQueue        queue of items of proposed barter chain, removed from
                  user lists of offers and propositions till the moment of
                  user decision regarding the chain execution
*/
```

```
1  currentAgent = agent.ID
2  while (currentAgent.isAlive) do
3     registered_agents = get_all_registered_agents()
4     cDList = cOList = currentMDItem = MOItem = ChainMaker = NIL
5     currentChainDecision = optimalChain = notRespondingAgents = itemsQueue = NIL
6     do
7        agreedAgentsList = false
8        agentsList = get_agents_states(registered_agents)
9        if ((0 < agentsList.BusyAgents < N) OR (agentsList.FreeAgents < 3)) then
10          requestingChainMaker = select(agentsList, notRespondingAgents)
11          send(requestingChainMaker, pendingToJoin)
12          waitAndListen(timeout, requestingChainMaker)
13          notRespondingAgents = notRespondingAgents + requestingChainMaker
14       else
15          for all a_i ∈ {agentsList.FreeAgents - currentAgent} do
16             send(agentsList.FreeAgents, formNewGroupProposition)
17          end for
18          wait(timeout)
19          if (received [agentsList.FreeAgents]-1 propositions) then
20             if (each foreignAgentsList.FreeAgents == agentsList.FreeAgents) then
```

```
21                agentsList = agentsList.FreeAgents
22                agreedAgentsList = true
23             end if
24          end if
25       end if
26    while (agreedAgentsList <> true)
27    for all a_i ∈ {agentsList - currentAgent} do
28       send(a_i, currentAgent.offers, currentAgent.demands)
29       cDList = updateCommonDemandsList(a_i.demands, cDList)
30       cOList = updateCommonOffersList(a_i.offers, cOList)
31    end for
32    currentMDItem = findMostDemandedItem(cDList)
33    dG = findMDGivers(currentMDItem, cOList)
34    MOItem = findMostOfferedItem(cOList)
35    oS = findMOSeekers(MOItem, cDList)
36    ChainMaker = ChainMaker(cDList, cOList, currentMDItem, dG, oS)
37    if (ChainMaker == currentAgent) then
38       runChainMakerService(agentsList, cDList, cOList, currentMDItem, dG, oS)
39    else if (receivedChainMakersMessage("new cycle start")) then
40       T = initTimer()
41       while (agentProvidesService(ChainMaker, T)) do
42          optimalChain = getResults(ChainMaker, T)
43          if (currentAgent ∈ optimalChain) then
44             queueItems(optimalChain, itemsQueue)
45             userCurrentDecision = getDecision(optimalChain, currentAgent.user)
46             if (userCurrentDecision <> "Yes") then
47                restoreQueuedItems(optimalChain, itemsQueue, currentAgent.offers,
currentAgent.demands)
48             end if
49             deleteQueuedItems(optimalChain, itemsQueue)
50             currentChainDecision = getCommonDecision(optimalChain, agentsList,
currentAgent)
51             sendOptChainDecision(ChainMaker, currentChainDecision)
52             sendOptChainDecision(currentAgent.deviceAgent, currentChainDecision)
53          end if
54       end while
55    end if
56    end if
57 end while
```

Attachment 2. ChainMaker Selection

Algorithm 2: ChainMaker selection

```
/*
Input parameters: ChainMaker(cDList, cOList, currentMDItem, dG, oS)
Variables used:
cDList          list of demands for all agents of a given system
cOList          list of offers for all agents of a given system
ChainMaker      ID of an agent that will make barter chains
currentMDItem   currently most demanded item in a system
dG              set of agents which offers currently most demanded item
oS              set of agents which demands currently most offered item
*/

1  ChainMaker = currentMDItem = NIL
2  for all item_i ∈ cDList
3     case (dG)
4        dG == 1: ChainMaker = oS
5           return ChainMaker
6        dG > 1:
7        case (dG ⋂ oS)
8           dG ⋂ oS == 1: ChainMaker = dG ⋂ oS
9              return ChainMaker
10          dG ⋂ oS > 1: ChainMaker =  findOldestAgent(dG ⋂ oS)
11             return ChainMaker
12          dG ⋂ oS == 0: ChainMaker =  findOldestAgent(dG)
13             return ChainMaker
14       end case
15    end case
16    currentMDItem = findNextMostDemandedItem(currentMDItem, cDList)
17    dG = findMDGivers(currentMDItem, cOList)
18 end for
19 return ChainMaker
```

Attachment 3. ChainMaker Operation

Algorithm 3: ChainMaker Operation

```
/*
Input parameters: runChainMakerService(agentsList, cDList, cOList,
currentMDItem, dG, oS)
Variables used:
agentsList      list of all active agents in a given system and their
                state
cDList          list of demands for all agents of a given system
cOList          list of offers for all agents of a given system
currentMDItem   currently most demanded item in a system
dG              set of agents which offers currently most demanded item
oS              set of agents which demands currently most offered item
chains[]        set of shortest barter chains for 3 combinations of
                demand types: 1) Strict; 2) Strict + Flexible;
                3) Strict + Flexible + Potential
refusedChains[] set of optimal barter chains, proposed by ChainMaker
                and refused by at least one of participants of each chain
queuedChains[]  set of optimal barter chains, proposed by ChainMaker
                that are pending for having common decision for each chain
newAgentsQueue  set of new agents willing to join existing group of agents
optimalChain    set of agents that are able to make an optimal
                barter chain (in a given system at particular time) and
                unique ID of that chain
treeRootAgent   agent in a system, ID of which will be taken by ChainMaker
                and put into a root of barter trees that will be
                further built
*/
```

```
1   refusedChains[] = queuedChains[] = newAgentsQueue[] = optimalChain = NIL
2   treeRootAgent = currentAgent
3   while (currentAgent.demands <> NIL)
4      chains[] = NIL
5      newAgentsQueue[] = searchNewAgents(agentsList)
6      if (newAgentsQueue[] <> NIL) then
7         for all a_i ∈ {agentsList - currentAgent} ∪ {newAgentsQueue[]}  do
8            inform(a_i, "service finished")
9         end for
10        return NIL
11     else
12        for all a_i ∈ {agentsList - currentAgent} do
13           inform(a_i, "new cycle start")
14        end for
15     end if
16     for all chain_i ∈ queuedChains[] do
17        if (hasDecision(chain_i)) then
18           if (chain_i.decision <> "Yes") then
19              cDList = restoreCommonDemandsList(chain_i)
20              cOList = restoreCommonOffersList(chain_i)
```

```
21              if (chain_i.decision == "No") then
22                 refusedChains[] = refusedChains[] + chain_i
23              end if
24           end if
25           queuedChains[] = queuedChains[] - chain_i
26        end if
27     end for
28     chains[] = findShortestChain(agentsList, treeRootAgent, refusedChains[], "S")
29     chains[] = chains[] + findShortestChain(agentsList, treeRootAgent,
   refusedChains[], "SF")
30     chains[] = chains[] + findShortestChain(agentsList, treeRootAgent,
   refusedChains[], "SFP")
31     if (chains[] <> NIL) then
32        optimalChain = chooseOptimalChain(chains[])
33        for all a_i ∈ optimalChain do
34           informOptChain(a_i, optimalChain)
35           cDList = removeFromCommonDemandsList(a_i.demands, cDList)
36           cOList = removeFromCommonOffersList(a_i.offers, cOList)
37        end for
38        queuedChains[] = queuedChains[] + optimalChain
39        for all a_i ∈ {agentsList - optimalChain - currentAgent} do
40           inform(a_i, "cycle finished")
41        end for
42     else
43        for all a_i ∈ {agentsList - currentAgent} do
44           inform(a_i, "no barter chain")
45        end for
46        return NIL
47     end if
48     if (includesFOrPDemands(optimalChain)) then
49        if (existNextAgent(treeRootAgent, currentMDItem, cDList)) then
50           treeRootAgent = nextAgent(treeRootAgent, currentMDItem, cDList)
51        else
52           if (existNextItem(currentMDItem, cDList)) then
53              currentMDItem = nextItem(currentMDItem, cDList)
54           end if
55        end if
56     end if
57     treeRootAgent = ChainMaker(cDList, cOList, currentMDItem, dG, oS)
58  end while
59  for all a_i ∈ {agentsList - currentAgent} do
60     inform(a_i, "service finished")
61  end for
```