Mobie: Personalized Automated Recharge of Prepaid Mobile Phones

J. Burgett[1], S. Hussain[1], C.M. Jonker[2], A. Razaq[1], K. Silz

[1] AMS Inc, 51-55 Gresham Street, London EC2V 7JH, JeBu@acm.com
[2] Department of AI, Vrije Universiteit, De Boelelaan 1081a, 1081 HV Amsterdam,
jonker@cs.vu.nl, http://www.cs.vu.nl/~jonker

Abstract. Prepay usage as a percentage of overall mobile phone access has increased sharply over the past several years. However, the recharging process is still largely manual with personalization provided by the user. The MOBIE system described in this paper shows the future service of automatically recharging the prepaid account of a mobile phone in a personalized manner. Special attention is given to the use of industry technology to implement the MOBIE system. The paper addresses some of the difficulties encountered in that process.

1 Introduction

Telecommunication is an interesting domain that offers both opposing and common interests and that is in full development due to the integration of mobile phones, the Wireless Application Protocol (WAP, http://www.wapforum.org), and the Internet. Consumers are interested in accessibility, security, and personal beneficial intent. The telecom industry is interested in improvement of the relationship with its consumers, because this will lead to improvement of market share, and loyalty of good consumers (i.e., customer retention). They are further interested in improved operating efficiencies, and increased revenue.

In this paper the design and implementation of MOBIE an intelligent system to support the telecommunications industry is described. The focus of the MOBIE system lies on the (relatively simple idea of) personalized automated recharge of mobile phones. This choice is motivated by the practical benefits of such an application. Simple reactive variants of automated recharge are already available with some telecom providers. The simple automated recharge that is already provided entails only the automated recharge of the account if the account drops below a certain threshold.

The intelligence provided by MOBIE is new. MOBIE monitors your actual usage patterns and compares that to the expectation that the user has of his usage. In case of a discrepancy, MOBIE enlightens the user. The user can then decide whether his expectation was incorrect, or whether some unapproved usage of the account is in progress.

The multi-agent paradigm enables the modeling of opposing and common interests in one system, called MOBIE♣. The interests of each party are modeled in their own agent, thus ensuring that the opposing interests are respected to the full. Cooperation between the agents representing the different parties enables the system to serve the common interests of all parties. The MOBIE multi-agent system consists of personal assistant agents for the consumers and business agents for the mobile telecommunication service providers.

2 Personalized Automated Recharge

On one hand, the user is interested in having an account as low as possible (because s/he will not accrue interest over the money located at the service provider). On the other hand the user wants to call unhindered by an account that is too low. This implies that from a financial point of view it makes sense to recharge his/her mobile account often. However, having to recharge the account is (still) time consuming; having to do this often is inconvenient. The telecom provider can increase consumer comfort by offering personalized automated recharging of the prepaid account by way of a personal assistant agent for each customer. The customer can also use a personal assistant agent offered by another party.

The core functionality of the personal assistant agent required to automate recharging consists of 5 elements. The business agent is designed to match at least the same capabilities. The personal assistant agent has to create and maintain a profile of the customer. The profile contains at least the criteria that tell the agent when to recharge the account and information needed to execute recharging (like the amounts it can use, and payment information). The profile also contains the expected usage pattern for the account. The personal agent has to match the criteria against the actual balance of the prepaid account, and the actual usage pattern against the expected usage pattern. The personal agent has to request the necessary information from the business such as the balance of the prepaid account and the actual usage pattern of the phone for a specified period of time. The personal agent is capable of recharging the prepaid account. The personal agent alerts the user in case of a discrepancy between the actual and expected usage patterns.

The customer initiates the personalized automated recharge by directly selecting the “Recharge” activity. Upon initiation of the recharge process, the Personal Agent retrieves the profile information related to recharging the phone. The Personal Agent replicates the latest account activity from the current service provider to its own Account History Database. The Personal Agent reviews the account activity and determines its actions. If a recharge is needed but the maximum recharge amount was already used up, the personal agent alerts the customer. If a recharge is needed and the maximum recharge amount has not been used up yet, then the personal agent uses

♣ MOBIE has been developed by the Intelligent Agent Initiative Team at American Management Systems (AMS).

knowledge rules to determine the appropriate recharge amount (these rules are proprietary information).

If the recharge amount, together with the amount recharged so far this week, exceeds the maximum recharge per week, then the customer is alerted. If a recharge was decided, the personal agent asks the business agent to recharge the account of the phone with the specified amount. If the recharge fails, the customer is alerted. If the recharge is successful, then, in due time, the personal agent notifies the customer about the recharge. The personal agent writes all activities in the agent activity log.

3 Pre-Design Decisions

Multi-agent technology supports the development of distributed systems consisting of heterogeneous components that by interacting produce the overall service required [2], [4]. The nature of the domain requires that the MOBIE system consists of heterogeneous components. It should at least consist of a personal agent system, a business agent system, and a number of databases. The databases are needed to store information on the users of the personal agent system and to store information on the consumers but now from the perspective of the business, and there is a billing system database; see Figure 1. The last two of these are already in use by the business, a good reason to keep them as separate entities.

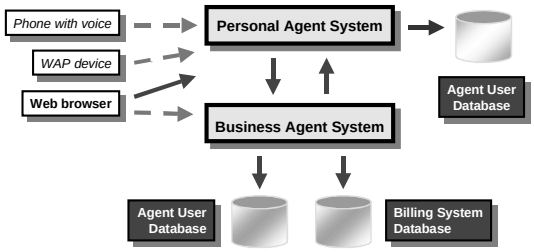

Fig. 1. The Multi-Agent System Functionality

The application should be able to access enterprise systems and enterprise data. Each agent has to be accessible through a number of different access channels: web-based, WAP and voice. Due to inherent restrictions of current WAP implementations (such as slow connection speeds and incompatibilities) and of mobile devices in general (such as small screen and limited input capabilities), a voice-enabled interface has high potential. However, the pilot version of the system only implements web access. The system has to be scaleable, reliable, and easy to maintain. Each agent has to be constructed on the basis of a domain-independent code framework (necessary for rapid and flexible agent development).

Based on an evaluation of the existing industry standards and techniques a decision was made to use some of the Java 2, Enterprise Edition (J2EE; http://www.javasoft.com/products/OV_enterpriseProduct.html) APIs because J2EE is

the de-facto industry standard for implementing and scaling Java on the server. The use of J2EE in the development of an intelligent agent framework helps ensure scalability and maintainability. The available agent communication languages were studied (e.g., KQML), but a full ontology was not required for this application. XML (http://www.xml.org) as the industry standard format for data exchange was chosen as the input/output format for MOBIE's agents.

In the design of MOBIE no logic resides on the client side. Running any code on the client side requires the client to have a runtime environment that can execute the code, enough resources to run the code, downloaded and installed the code on the client's machine, and sophisticated security software against malicious attacks. The most popular client-side scripting language for web pages is JavaScript. WAP devices use the Wireless Markup Language (WML) for page description. WMLScript is the client-side scripting language (http://allnetdevices.com/faq/?faqredir=yes). A voice-enabled access channel does not have any scripting at all. An interesting client-side logic alternative to JavaScript that we did not investigate is Macromedia's Flash technology (http://www.macromedia.com/software/flash/). In our system, the MOBIE character on the screen is a Flash animation.

Investigation of commercially available Java agent platforms, namely AgentBuilder (http://www.agentbuilder.com) and Jack (http://agent-software.com.au) revealed a number of disadvantages: no built-in J2EE or XML support, proprietary communication protocol, hardwired reasoning engine, proprietary development environment (AgentBuilder) and high resource consumption. Instead of taking these products and adding missing features, an agent framework has been developed that focuses on scaleable, e-commerce capabilities. The reasoning engine of the framework is OPSJ from PST (http://www.pst.com), a Java rule engine with small memory footprint. VisualAge for Java (http://www.software.ibm.com/ad/vajava/) was used to develop the Java code, because from our experience it is the best available Java IDE.

MOBIE makes use of the following J2EE (http://www.javasoft.com/j2ee) APIs: JavaServer Pages (JSP), Servlets, Enterprise Java Beans (EJB), the Java Naming and Directory Interface, Java Database Connectivity, and the Java Transaction API. Servlets [3] are server-side Java components that can be accessed from HTML pages. Once loaded by the application server, they reside in memory. The application server automatically spawns a new servlet instance in its own thread for every incoming request. A servlet can use the entire Java API and has built-in support for session management (carrying user data while navigating through a web site).

In the design of MOBIE all three tiers of the Enterprise Java Application Model (http://www.javasoft.com/j2ee/images/appmodel.jpg) are used: the client tier ("Client-Side Presentation"), the middle tier ("Server-Side Presentation" & "Server-Side Business Logic") and the data tier ("Enterprise Information System").

4 The Backbone Architecture

In this paper a generic architecture for multi-agent systems is presented that can be accessed over different channels. The architecture is especially suited for multi-agent systems in which there are at least two parties playing a role.

MOBIE has consumers and businesses. The consumer deals with the business indirectly by allowing his/her personal agent system to act on his/her behalf. The consumer can interact with the personal agent through three different access channels: by phone (using ordinary speech), by accessing the web site of the business on the Internet or by using a WAP (device such as a phone or a PDA). The system is designed according to one generic open system's architecture that is presented in Figure 2; it follows the J2EE recommendations by Sun. Each agent is embedded in its own EJB.

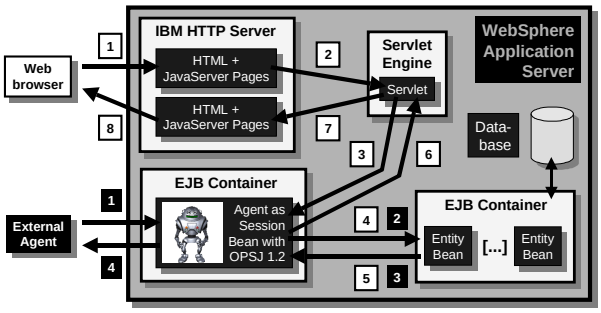

Fig. 2. The MOBIE Backbone Architecture

The upper half shows the control flow of a user – agent interaction. Firstly, the user enters some data into a web page (either an HTML page or a JSP; ①). The servlet extracts the information (②), converts it from Java into XML and sends it to the agent (③). The agent processes the information and then requests data through an entity bean (④). The entity bean is delivered (⑤) and used by the agent. After some more processing, the agent comes up with a result and returns it to the servlet as an XML document (⑥). The servlet examines the results, converts it into a Java object and sends it to the appropriate web page (either an HTML page or a JSP) in the appropriate form (⑦). The user then views the page (⑧).

The lower half shows the agent – agent interaction. The outside agent sends an XML request to the agent (❶). In the same way as before, the agent processes the information and then requests data through an entity bean (❷). The entity bean is delivered (❸) and used by the agent. After some more processing, the agent comes up with a result and returns it to the external agent as an XML document (❹).

In order to formalize and simplify the Java integration with XML, Business Objects (BO) were used. These special Java classes represent the XML data and support XML creation and parsing.

5 Agent Design

Usability engineering tasks included design of the screen look & feel, agent onscreen look, agent onscreen animation, agent onscreen movement, dialog interaction, and agent sound. First the generic agent architecture is introduced, and then the working of the agent is described in the form of a scenario.

5.1 Generic Agent Architecture

The main goal of the agent architecture was to build an industrial strength, domain-independent framework for huge scale e-commerce applications. As a general approach, an EJB like design (the so-called agent container) was chosen. The container offers services to the agent through static methods in the `Services` class. Each agent has to implement one Java interface. The container services were implemented as static methods since OPSJ rules only have access to objects in their working memory and to static methods (see section 6.1.2). Instead of inserting a service object and referencing it in every rule condition, the agent container uses static methods.

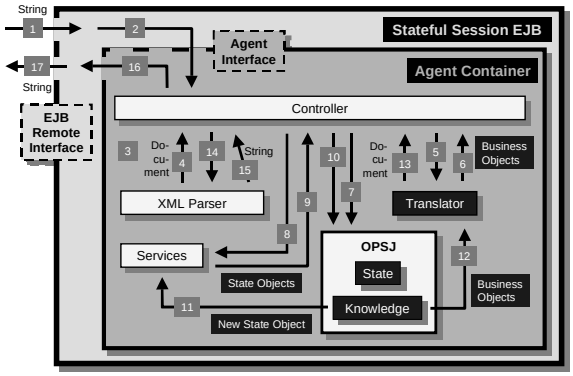

Fig. 3. The Agent Architecture

Figure 3 depicts a more detailed design of the agent (the numbers are referred to in the next section). The agent container is embedded in a stateful session EJB (application specific classes are depicted in white on blue). The EJB classes are just a wrapper around the container; there is little functionality in the EJB. There are two methods in the Agent Interface (and in the EJB remote interface) that trigger the agent reasoning: `String reason( String )` and `Document reason( Document )`. Both methods expect and return a BO as an XML document – either flattened to a string or as an XML Document Object Model (DOM) tree. In both cases, the EJB just passes the messages on to the agent. Since XML is the data exchange format of the agent, the agent has an XML parser to parse strings into documents and to flatten documents to strings. The business logic resides in the Java rule engine OPSJ. The knowledge (i.e. the rules and Java methods) and the state (i.e. objects in the working

memory of the rule engine) are application specific. The agent container uses the application specific `Translator` class to convert between XML and BOs. The `Services` class offers functionality through static methods.

As a result of the architecture, there is very little that sets one specific agent apart from another one: a list of rule set classes, an object derived from the `Translator` class and (implicitly) the BOs used. Each specific agent's remote interface extends (without any changes or additions) the generic agent EJB's remote interface. Similarly, each specific agent implementation class extends the generic agent EJB's implementation class and must implement two methods that return the aforementioned list of rule set classes and the `Translator` instance.

5.2 A Scenario

The steps in the scenario refer to figure 3. Application specific classes are depicted in white on dark. With regards to Figure 2, this details steps ③ and ⑥ (without the entity bean interaction in steps ④ and ⑤).

The scenario starts with an XML document (stored in a flat string) passed into the agent interface (1). The EJB passes it on to the agent (2) where the controller takes charge. The XML Parser decodes the string into an XML document using the Java class `Document` (3 and 4). The controller asks the `Translator` to transform the document into BOs (5 and 6). After the transformation the OPSJ working memory is reset. Then the controller takes each `BO` from the Translator list and inserts it into the OPSJ working memory. In the next step, it gets a list of the (externally stored) state objects of the agent (8 and 9, see the previous section) and are inserts them into the OPSJ working memory (10). The OPSJ inference process starts. As a result of the reasoning, a BO is stored in the `Translator` where it is converted into a `Document` (12). The controller picks it up (13) and the XML parser flattens it to a string (14 and 15). This string is returned to the EJB (16), which then returns it to the EJB client (17).

6 Discussion

Systems can consist of both straightforward and complex tasks. In our opinion, straightforward tasks (e.g., logging activities) should be implemented conventionally (e.g., in plain Java). Complex tasks that require autonomous intelligence (for example, negotiating a better price plan) can be implemented with agents. We found use cases to be an effective way of partitioning system functionality into tasks and assigning them to agents.

The main result of our project is that industry standard methodologies, technologies and tools can be used to build multi-agent systems; software developers can utilize their investments in these areas. Furthermore, in industry, new systems have to coexist and co-operate with legacy systems. Building the new systems with proven tech-

nology in the form of industry standard technology eases the process of the integrating the old with new systems. We demonstrated that for heterogeneous systems requiring autonomous intelligence in a naturally distributed setting, a de-coupled, distributed, XML based architecture like MOBIE eases software development considerably.

However, agent development with industry standard methodologies does have its disadvantages. Those methodologies are based on a different paradigm, which complicates designs. The lack of scalable agent tool integration (e.g., rule engines) makes agent implementation more complex than it should be (e.g., no integrated source code debugger as in Java tools). Also, the concept of (agent) autonomy is absent from object-oriented methodologies.

We imagine that future multi-agent projects will use extended versions of "regular" software system development tools, i.e., an evolutionary improvement - not a revolutionary one. The main difference will be that specific encapsulated roles will be identified, modeled and implemented as intelligent agents. Just as objects brought a higher level of abstraction to software analysis and design, role modeling for agent-based systems will bring a higher level of abstraction than objects [1].

Since the construction of the multi-agent system described here, the world has not stood still. In the mean time the big prepaid mobile phone operators already have some sort of assistance for account recharge. For example, D2 Vodafone in Germany allows a recharge of either 50 DM or 100 DM through an Interactive Voice Response (IVR) service number with direct debit from a bank account. Telecom Italy has an even more advance rang of offerings: IVR, SMS, ATMs, and with an automatic recharge by a fixed amount (through direct debit) if the account falls below a certain threshold. The research presented in this article is therefore of direct applicability.

The chosen domain for the MOBIE project is more in the area of business to consumer (or vice versa) than business to business. However, the results are easily transferable to other applications.

References

1. Ferber, J., Gutknecht, O., Jonker, C. M., Mueller, J. P., and Treur, J., Organization Models and Behavioural Requirements Specification for Multi-Agent Systems. In: *Proceedings of the Fourth International Conference on Multi-Agent Systems*, ICMAS 2000. IEEE Computer Society Press. In press, 2000. Extended version in*: Proceedings of the ECAI 2000 Workshop on Modelling Artificial Societies and Hybrid Organizations*, 2000.
2. Giampapa, J., Paolucci, M., and Sycara, K., (2000), Agent interoperation across multi-agent system boundaries. In: *Proceedings of the Agents 2000 workshop on Infrastructure for scalable multi-agent systems*.
3. Hunter, J., and Crawford, W., (1998), *Java Servlet Programming*, O'Reilly and Associates.
4. Jennings, N., Sycara, K., and Wooldridge, M., (1998), A roadmap of agent research and development. In*: Autonomous Agents and Multi-agent Systems*, Kluwer Academic Publishers, Boston, pp. 275-306.