Embedding Stationary Service Agents in a Mobile Agent Environment[1]

Nataša Ibrajter, Dragoslav Pešović, Zoran Budimac

Department of Mathematics and Informatics, Faculty of Science,
University of Novi Sad, 21000 Novi Sad, Serbia & Montenegro
Email: {natasha,dragoslav,zjb}@im.ns.ac.yu

Abstract. The paper presents one solution that is adopted in an implementation of workflow management systems using mobile agents. Mobile agents occasionally need additional services that are implemented as stationary agents (for example, mail service, ftp service, database connectivity...) The paper presents one possible solution that uses existing implementations but that fit into a mobile system in such a way that compatibility with mobile workers is achieved. The paper discusses in more details the implementation of the stationary mail agent.

1 Introduction

Using the workflow management systems is a modern trend in modeling and implementation of an information flow and business processes in a company [3, 9, 15, 26]. Well organized and fully implemented, a workflow can replace a large part of a classical information system. In a focus of a workflow, there is "work" (information, task, document, announcement, data, etc.) that flows through different points in a company. Deadlines and conditions of a work transition are defined in order to achieve the flow of work. When a condition is met, work is transferred to a next stage, where the next condition has to be fulfilled. At the end of its itinerary, the work is done.

According to most general definition, a mobile agent is a program that is able: to stop executing at one node in a computer network; to transfer itself to another node in a network; and to continue execution there. More precise definitions include additional requirements for a piece of code to be a mobile agent. All definitions however, stress the important feature of mobile agents - autonomous behavior. A mobile agent autonomously decides when and where it will be transferred. Mobile agents are a fresh research area in the field of a distributed programming and artificial intelligence [1, 4-8, 13, 16-19, 24, 25]. A mobile agent is today often an object of some object-oriented programming language. Advantages of mobile agents with respect to classical techniques of distributed programming are numerous [14].

[1] This work is partially supported by the Ministry of Science, Technologies and Development, Republic of Serbia, through the project no. 1844: *Development of (intelligent) techniques based on software agents for application in information retrieval and workflow*.

We used mobile agents to implement a workflow management system (WFMS) [2, 10, 21, 22]. This approach has several main advantages over more classical approaches in organization and implementation of WFMS: uniformity of organization and implementation, simpler flow management, flexibility, scalability, reliability, mobility, support of unstructured business processes, support of inter-organizational work, etc.

This paper concentrates on embedding stationary services into our 'mobile' WFMS. Mobile agents acting as workers often need services like database access, FTP, mail service, ... that would be too big to be included into mobile agents themselves. Therefore, we had to implement them as stationary services residing on some nodes in the network. The problems we had to solve were: a) how to implement them quickly and efficiently such that we can produce new services routinely on a weekly basis, and b) how to make those services compatible with the rest of the system thus enabling easy communication. The paper presents solutions to these two problems taking as a case study an email stationary service.

The rest of the paper is organized as follows. In the next two sections we introduce mobile agents and our workflow management system that is based on them. Fourth section introduces the stationary service and answers the questions how to find them and how to communicate with them. Fifth section describes e-mail service and JavaMail API that we choose to use, while in the sixth section the architecture of the e-mail service agent is surveyed. Section seven concludes the paper.

2 Mobile Agents

A mobile agent [6, 7] is a program, which may migrate from one node to another in a heterogeneous computer network. The mobile agent can suspend its execution at any time, transport itself to another computer in the network, and continue the execution on that new network location. On the target computer, an agent does not restart its execution from the beginning – it continues where it left off.

Mobile agents also require some kind of an execution environment installed on potential hosts to run on. All mobile agent systems have an *agent server* running on potential host machines in the network. Its primary task is to provide an environment in which mobile agents can execute. The agent server also provides the functionality for agents to migrate, to communicate with each other as well as to interact with the underlying computer system. Furthermore, this infrastructure has to provide security mechanisms preventing malicious agents to attack other agents or the underlying computer system on one hand and avoiding manipulations of the hostile agents by a malicious computer system on the other hand. It may also provide some support services, which relate to the agent server itself, services to support access to other mobile agent systems etc.

In Mole, the platform we use [1, 24], there are two different kinds of agents, the system agents and the user agents. User agents are ordinary mobile agents, programmed and employed by the user. They have absolutely no access to the underlying system. System agents are agents with special privileges for access to system resources, providing controlled, secure abstractions of these resources inside the agent

system. System agents are immobile and may be started only by the administrator of the location. User agents may only communicate with other agents and have no direct access to system resources.

3 Workflow Management System

The current architecture is essentially two-part, consisting of work-agents (workers) and worker-hosts.

Workers. Abstract class *Worker* represents an abstract work in the proposed workflow system. This class is a descendant of a class that represents a mobile agent in the mobile agent system Mole. Objects of the class *Worker* thus become mobile as well. The class has attributes containing work identification, work deadlines, work owner, and possibly other important information. The class also contains an itinerary.

Every worker is required to follow the itinerary, which is a list of triples of the following form: (node, condition, methods). A node represents the address of the current network location, where only methods enlisted in the list methods are available to the user on that node. Worker will transfer itself to the next node when and if the condition (a logical function) is met. It is the responsibility of the abstract worker itself to check the condition periodically or at any other appropriate moment.

Concrete workers are descendants of the abstract Worker and contain attributes that describe the work and methods that can be used to process the work. Worker's behavior is almost entirely defined by its itinerary.

Agent server. Mobile agent system Mole, being the basis of our workflow system, takes care of all the agent-related activities of workers. For instance, Mole engines, residing on every node in the system, have the following tasks:

- Listen the designated port, wait for incoming agents, internalize them, put them into a list of agents on that host, and activate them.
- At the end (before switching off the computer), externalize all agents that are currently under their supervision.
- At the beginning of their execution (after the computer is switched on), internalize all saved agents.

Worker-host. Every node in the network contains a worker-host that is implemented as a stationary system agent, having special privileges for the access to host system resources. Mole agents and thus workers are forbidden to access any system resources directly but only by communicating with system agents i.e. worker-hosts. Worker-host represents the interface component between the (operating) system, workers, and users. Every worker-host has a user interface, which enables the interaction with the user of the workflow system.

Every worker is automatically placed in the list of workers residing on the current node, and the corresponding icon appears on the user interface of the host. If the icon of the worker is selected, the list of all methods available on the current node is presented to the user, allowing him to invoke any of them.

If the method of a worker is required to present a user interface, it cannot show the instance of the *Window* class on the screen directly. The method should instead just initialize the adequate interface and pass the reference to the local worker-host, which will recognize the reference and show it on the screen.

During its visit to the node, worker itself will periodically check if the condition for the transition on the next node in its itinerary is fulfilled. When this condition is met, the worker reports its departure to the host, resulting in the corresponding icon being removed from the user interface. The worker is then transferred to the next node specified in its itinerary. When the itinerary is exhausted, worker's journey is complete, and the worker is removed from the system.

Workers and work-hosts are the only software components that are really necessary for the workflow system. However, if the system is to be user-friendlier and more flexible, additional tools and specialized agents need to be included.

Although workers are almost fully autonomous, they may need additional services to finish their work. Those services cannot be embedded directly into the workers as this would prevent keeping workers as small as possible. The services are therefore implemented separately, as stationary agents.

4 Introducing Stationary Services/Agents

To increase efficiency of the whole system, workers should be of minimal possible size. Therefore, "standard" and additional features of the whole system must be implemented as separate stationary agents. These agents can lower the degree of worker autonomicity, because now workers partly depend on external services that may be not available all the time. To diminish the bad consequences of external functionalities, the additional services are widely spread over the whole workflow network. Among stationary services, our system currently has services for e-mail, FTP, and database access.

Service agents do not need to be started explicitly every time the underlying machine is turned on. Once they are started, they exist on the location being automatically placed in the external storage every time the underlying machine is turned off, and subsequently resumed from the storage when the machine is turned on.

4.1 How to Find Service Agents on the Location?

In order to find stationary agents that are available on the location, workers use built-in Mole's mechanism - directory service. A directory service is an electronic database that contains information on entities. In the Mole system, there is a simple local directory service that maintains information about agents providing a service denoted by a string. This local directory service exists on every place.

If an agent provides a service, it can register itself locally by submitting a string identifying the service to the directory service.

Another agent wanting to use this service first asks the directory service. The directory service returns a list containing names of all agents providing the service. This

list is either empty, or contains one or more agent ids. The agent can choose one of the ids and contact the appropriate agent.

Worker-host registers itself for the service of “HOSTING”. Mailer’s service however is registered under the label “MAILING”.

Workers keep *id*s of service agents that are needed on the location in their private attributes. After arrival on the new location, these attributes are set to null. If and when a service agent is needed for the first time on the location, the worker will perform a query and keep the name returned in the adequate attribute. Every other time the service agent is needed on the location, worker will not perform a search again, but will use the saved id to contact the service agent.

4.2 How They Communicate?

Workers communicate with stationary agents by invoking their public methods. This is done through Mole’s built-in RPC (remote procedure call) mechanism, which uses Java RMI (remote method invocation).

In order to initiate RPC, one of several offered *call* methods is invoked. All the variants of this method as first three parameters take: the name of the calling method, ID of target agent, and the current location of the agent. Additionally, most of the *call* methods take at most five serializable objects, which are passed to the remote method as parameters. One variation of this method takes an array of serializable objects, allowing remote method to be invoked with unlimited number of parameters. All versions of the *call* method pass back the result of the remote method, or *null* if the calling method is declared void. Mole catches all the exceptions possibly produced by the remote method, and sends them back to the agent’s location to be locally handled.

Worker-host has four methods that workers can invoke (Fig. 1). They are all used for bookkeeping of workers by worker-hosts:

- **arrival** will be called by workers when they arrive on the node. Host will first check it and then add the worker to its list of active workers.
- **departure** will be called by worker just before they leave the node.
- **workCompletion** is similar as above, but called when the worker has finished its work (and it will be destroyed).
- **migrationFailure** will be called when worker cannot migrate to the next node in its itinerary.

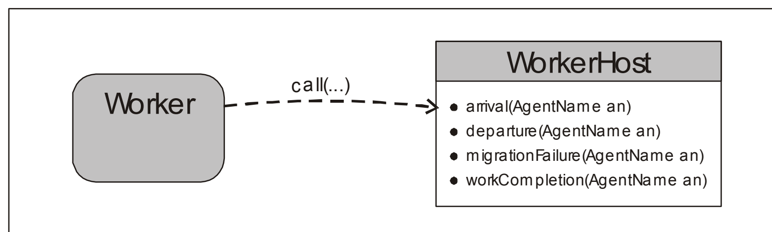

Fig. 1. The worker invokes host’s methods

5 E-mail and JavaMail API

Most people start using Internet by using electronic mail (*email*). The email does almost the same thing as the classical post service: message is put in the addressed envelope and sent. For every email envelope is an email header. Header consists of few fields: *To*, *From*, *Subject*, etc. When post service is not able to deliver the letter to the recipient, letter is returned back with the information about unsuccessful delivery. Email does the same: email message is returned back together with the list of all nodes that tried to deliver the message.

Electronic mail is much wider spread than Internet, and because of that, email message can reach much more nodes than, for instance, FTP. Nevertheless, email has one disadvantage: it is only textual medium. It means that email is able to transfer only messages that consist of characters. Because of that, sending binary files is not natural for email, contrary to sending textual files. In order to overcome this problem, it is necessary to enable computers of both sender and receiver to convert binary files into some sort of ASCII representation. All UNIX systems have this ability: uuencode, btoa, BinHex for Macintosh etc.

Networks had evolved, and in one moment, it was necessary to make email able to send all kinds of files. MIME (*Multi-purpose Internet Mail Extension*) is the specification that makes possible automatic sending of non-textual objects via email. MIME frees user from explicit converting files into ASCII representation. It is possible to send any form of file if the sender and receiver both have software that supports MIME specification. MIME redefine message format and allows non-ASCII character to appear in header fields and content. It has dynamic set of different formats for non-textual message content, enabling email to send animations, pictures, applications, sound, as well as combination of all of them.

RFC 822 [20] (*Request for Comments: 822*) foregoes MIME specified message in following manner: message consist of header fields and content. Content is sequence of ASCII character lines. Headers and content are separated by empty line (only CRLF). The most important header fields are: *To*, *From*, *Subject*.

5.1 JavaMail Application Programming Interface (JavaMail API)

Java is one of the most popular programming languages for creating platform independent, network safe applications. That is the reason why Java is so widely used for network (Internet) and distributed programming. Java is also a programming language that Mole (our mobile agents platform) has been implemented. Therefore, we had to search for a suitable solution in Java.

JavaMail API makes possible for Java developers to enable their applications for sending, receiving and manipulating electronic mail. JavaMail API was intended for evolving simple application with email-sending functionality as well as for developing software packages for Internet providers. Because of that, it had to be simple enough for the first users, and powerful enough to satisfy the needs of the latter. API has basic abstract classes and interfaces, which model typical mailing system. There are classes which implement these abstract classes, and which can enable applications for simple email manipulation. The abstract classes can be implemented in any that

way developers of big applications find convenient, in order to satisfy their specific needs. In both cases, end user always sees the same interfaces.

5.2 JavaMail API architecture

JavaMail API can be divided in three layers:

- **abstract layer** declares classes, interfaces and abstract methods meant for implementing functions for handling electronic mail, supported by every mailing system. API elements of this layer are intended for augmentation in order to support standard data types and to offer interface to transport and message access protocols, if needed,
- Internet **implementation layer** implements part of abstract layer in accordance with Internet standards – RFC 822 and MIME,
- JavaMail uses JavaBeans Activation Framework [12] (JAF) for wrapping and handling message data. Data are manipulated through JAF JavaBeans, which are not part of JavaMail API.

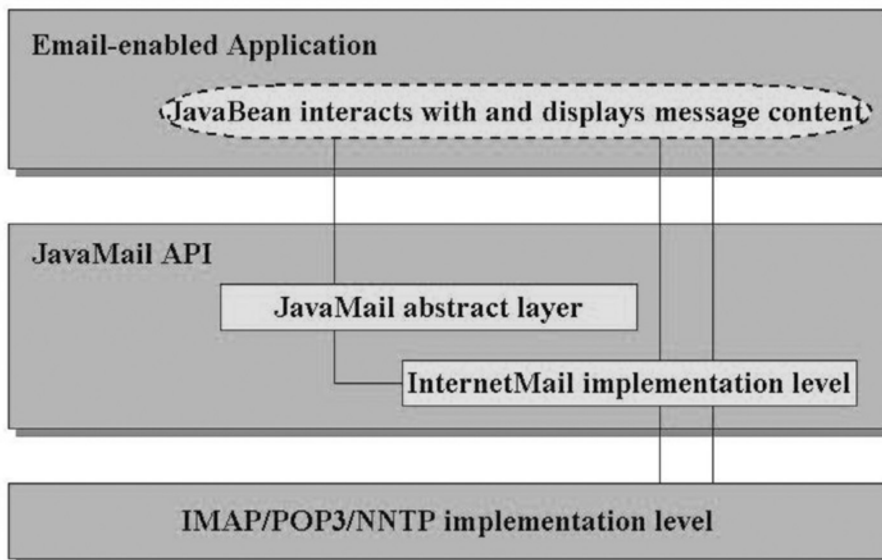

Fig. 2. The place of JavaMail API in email-enabled application

Figure 2 on top level presents email enabled application, and within it is a JavaBean which abstracts message content handling: creation, editing and access. Middle layer is JavaMail API layer which consists of abstract classes and classes which implements some of the abstract according to Internet standards. Application uses JavaMail API, if the offered implementations of abstract classes suite its needs, or it can use classes which implement JavaMail API like IMAP, POP3, or NNTP provider.

5.3 Messages and Message Handling

Message is an abstract class which implements *Part* interface (Fig. 3). *Part* interface consists of attributes, which describe content of the *Message* object. It interfaces mail system and manipulates message content (defines standard headers set, message content data types, etc...). This interface would not be necessary if the only class implementing it would have been *Message* class. But, abstract class *BodyPart*, as well as interface *MimePart* implements *Part* interface. *MimePart* extends *Part* interface, adding it RFC822 and MIME semantic and attributes. Figure 3 shows *MimeMessage* and *MimeBodyPart* classes using *Part* interface indirectly, through *MimePart* interface. *Message* class adds attributes needed for email sending: *From*, *To*, *Subject*, *Reply-To*, etc. Classes extending *Message* class, as *MimeMessage* class, add their own specific attributes.

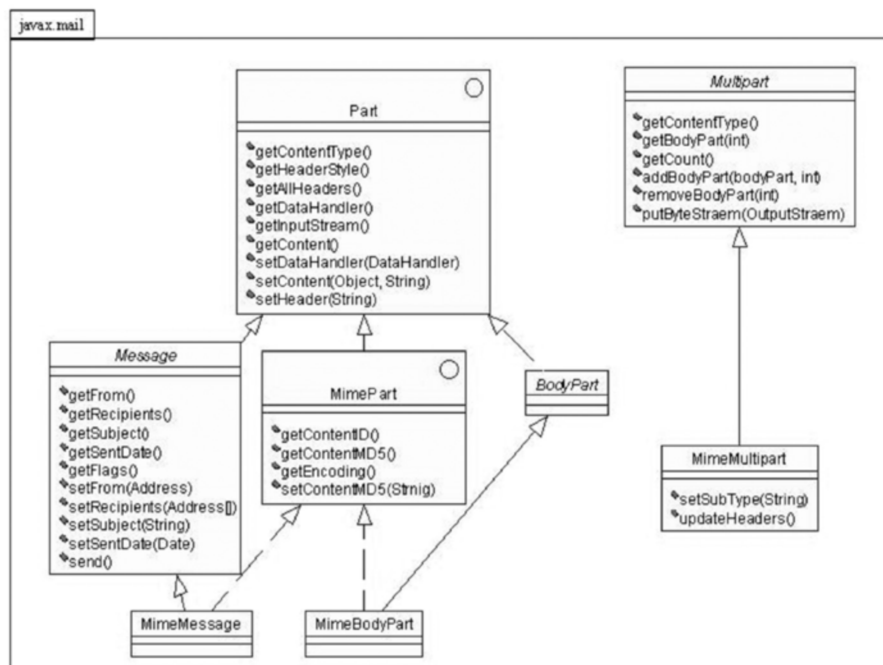

Fig. 3. JavaMail package

JavaMail knows nothing about email message's data format or type. *Message* object accesses its content through intermediary – JAF. In this manner, *Message* objects handle their content using the same API methods, regardless of content type and format. JavaMail API components access message content like: input stream, *DataHandler* object, or like Java object.

Message object can contain multiple parts, and every part is described by its own set of attributes, and has its own content. Content of a multipart message is *Multipart* object. It contains *BodyPart* objects. *BodyPart* objects represent separate parts of

multipart message. *BodyPart* is an abstract class and it implements *Part* interface, only defining what was declared in *Part* interface. This class does not add attributes *From*, *To*, *ReplyTo*, or other headers, like *Message*. *BodyPart* objects are intended for insertion in *Multipart* container.

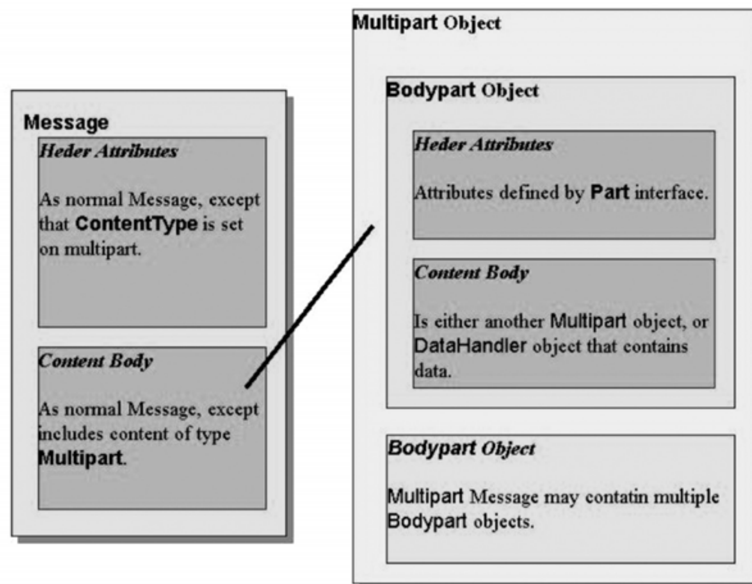

Fig. 4. Multipart message

5.4 Mail Session and Message Transport

If JavaMail API is a factory, than we could tell that *Message* class would be product of the factory, and class *Session* would represent the production. During production, the following are changed: product which is manufactured (different *Message* objects), tools (access and transport protocols), even the one who is carrying the production out. In advance are known the tools which could be used and the type of a product which is to be produced. Session set the preconditions, like configuration options, user authentication (used in later communication with mailing system), etc.

JavaMail API supports multiple sessions, even in the same time. Every session can use more than one submission and transport protocols (in JavaMail API factory, every production can use more than one tool. Tools are predefined by user: environment options, especially mail.store.protocol, mail.transport.protocol, mail.host, mail.user, and mail.from.

Transporting message is handled by *Transport* object, which is instance of some of the subclasses (implementations, for instance, of the SMTP) of abstract *Transport* class. *Transport* object is rarely explicitly created, and is usually obtained from *Session* class. Abstract class *Transport* defines interface to message submission and transport protocols.

6 JavaMail Stationary Agent

Agent is implemented as a class named *Mailer*. Agent itself is realized by extending *SystemAgent* class, from mobile agent system Mole. System agent has access to the host resources and it is stationary (it can not migrate). The method that carries most of the functionality is named *sendMail*.

Worker is calling method *sendMail* with an array of parameters: the kind of e-mail to be sent *(text/plain, html, file)*, to whom the message is sent (email address of the receiver), subject of the message, the name of the file with the message content (or the content is passed as a string), the name of the file which is going to be attached to the message. Depending on arguments which mobile agent passes to stationary mail agent, methods *plain* (send plain message - only text), *fajl* (send message with attached file) or *html* (send message in a HMTL format) will be called.

Figure 5 shows UML diagram of the class *Mailer*. Class *Mailer* imports JavaMail package *javax.mail*, *javax.mail.internet*, and *javax.activation*. The last package is needed only because of handling message content. In the first one, there are most important classes, like *Session*, *Message* and *Transport*. *Mailer* implements *SystemAgent* method *prepare*, setting in that way environment properties, obtaining *Session* object, and registering *Mailer* object as stationary agent that offers some services. Protected methods *collect* and *collectHtml* collect content of the file with the intended content of the email and return *StringBuffer*, that is turned into *String*, and added to the message as message's content. Message is sent by calling *Transport.send(msg)*, where *msg* is message to be sent.

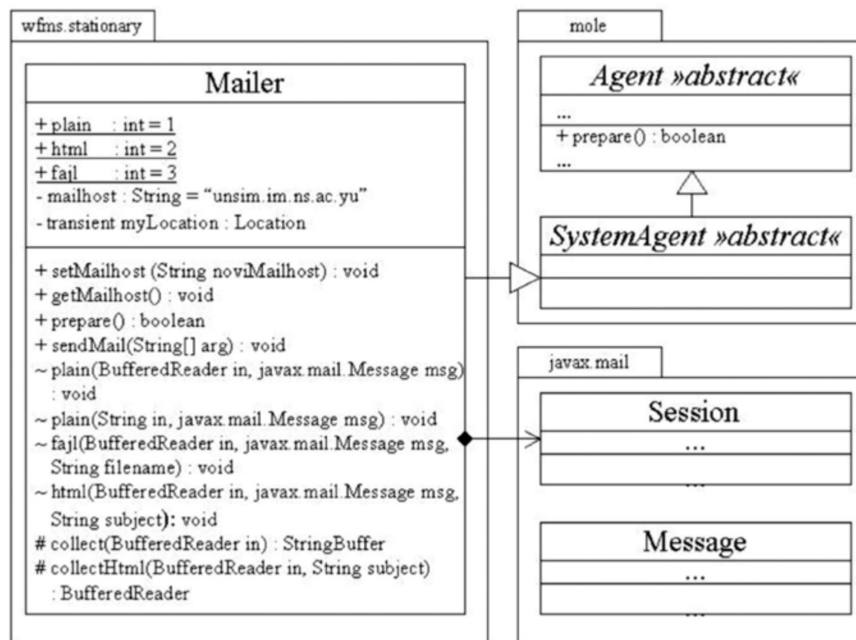

Fig. 5. *Mailer* implementing *SystemAgent* and using JavaMail API

The life cycle of the mailing stationary agent on a host:

- if the machine is turned on for the first time, then the agent should be explicitly started on,
- otherwise, service agent would be obtained from the external storage where it was stored when the machine went down,
- service agent registers itself with the directory service,
- when worker (mobile agent) gets to the location, it asks for the list of service agents that provide desired service on that host,
- it chooses one service agent from the list, and keeps ID of the chosen stationary agent in its internal list, using that stationary agent as the service provider for desired service the rest of its life-time on that host,
- worker invokes public method of the chosen stationary agent, using *call* method.

7 Conclusion

The paper presents our solution to the problem of inclusion of stationary services into a mobile agent system (email, database connectivity, FTP...). This problem is rarely addressed in consideration of mobile agent systems and is usually left to the end-users of such systems to be solved separately.

By inheriting a stationary agent class from an underlying mobile agent system (Mole) and by including JavaMail API classes, we got the best of two worlds: we got the stationary services that are fully compatible with the rest of mobile agent system (communication and search for services, among others) and a way to rapidly produce new services using existing solutions.

References

1. Baumann J., Hohl F., Rothermel K., Strasser M.: Mole – Concepts of a Mobile Agent System. University of Stuttgart Homepage, (1997
2. Budimac Z., Ivanović M., Popović A.: Workflow Management System Using Mobile Agents. *Proceedings ADBIS Conference*, Springer LNCS Vol.1691, Maribor, Slovenia, (1999) 169-178
3. Debenham J.: Constructing an Intelligent Multi-agent Workflow System. *Proceedings AI Conference*, Brisbane, Australia, (1998) 119-126, (1998)
4. General Magic: Introduction to Odyssey API. Homepage of General Magic.
5. General Magic: Mobile AgentsWhite Paper. General's Magic Homepage, (1997)
6. Gray R.S.: Agent Tcl: a Flexible and Secure Mobile Agent System. Ph.D. thesis, Dartmouth College, Hanover, NH, (1997)
7. Green S., Hurst L. at all: Software Agents: a Review. IAG Report, Trinity College, Dublin, (1997)
8. Harrison C., Chess D., Kershenbaum A.: Mobile Agents: are they a Good idea? Homepage of IBM T.J. Watson Research Center, (1995)
9. Hollingsworth D.: Workflow Management Coalition – the Workflow Reference Model. The Workflow Management Coalition Specification - Doc. no. TC00-1003, (1995)

10. Ibrajter N., Budimac Z.: Stationary Mail Agent. *Proceedings PRIM Conference*, Zlatibor, Yugoslavia, (2002)
11. JavaMail API Design Specification, Sun Microsystems Inc. (1988) http://java.sun.com/products/javamail
12. JavaMail Guide for Service Providers, Sun Microsystems Inc. (1998) http://java.sun.com/beans/glasgow/jaf.html
13. Karnik N.M., Tripathi A.: Design Issues in Mobile Agent Programming Systems. *IEEE Concurrency,* Jul-Sep.:52-61 (1998)
14. Lange D., Oshima M.: Seven Good Reasons for Mobile Agents. *Communications of the ACM* 42(3):88, (1999)
15. Leymann F., Roller D.: *Production Workflow – Concepts and Techniques*. Prentice Hall New Jersey, (2000)
16. Lingau A., Drobnik O.: An Infrastructure for Mobile Agents: Requirements and Architecture. Homepage of Jochan Wolfgang Goethe Univeristy, Frankfurt am Main, (1997)
17. Marwood D.: Extending Applications to the Network. M.Sc. thesis, Department of Computer Science, Univ. of British Columbia, (1998)
18. Mitsubishi Electric: Mobile Agent Computing – White Paper. Homepage of Horizon Systems Lab, (1998)
19. Morreale P.: *Agents on the Move*. IEEE Spectrum, April:34-41 (1998)
20. Network Working Group, Request for Comments
21. Pešović D.: Joint-Paper Worker. *Proceedings PRIM Conference*, Zlatibor, Yugoslavia, (2002)
22. Pešović D., Budimac Z.: An Advanced Joint-paper Worker. *CCS journal*, to appear
23. Pešović D., Budimac Z.: A Comparative Analysis of Several Mobile Agent Systems. *Novi Sad J. Math.* 30(2):95-111 (2000)
24. Strasser M., Baumann J., Hohl F.: Mole – a Java Based Mobile Agent System. Home-page of Stuttgart University, Stuttgart, Germany, (1996)
25. Venners B.: Under the Hood: the Architecture of Aglets. *Java World*, (1997)
26. Workflow Management Coalition – Terminology and Glossary, Homepage of Workflow Management Coalition, (1999)