The Development of a Mobile Agent Platform

A. Petric, M. Kusek and G. Jezic
Department of Telecommunications / Faculty of Electrical Engineering and Computing, University of Zagreb
Zagreb, Croatia
{ana.petric, mario.kusek, gordan.jezic}@fer.hr

Abstract— **This paper describes the software engineering process used in the development of the Crossbow2 agent platform. Crossbow2 is a light – weight agent platform that enables the basic agent functionalities: agent creation, migration, detection, communication and destruction. The platform was developed with the Extreme Programming (XP) approach. The paper elaborates the activities included in the software development process and the XP practices used in the requirement specification, system design, implementation and testing.**

I. INTRODUCTION

The motivation for the development of a new mobile agent platform was the fact that existing platforms do not achieve the best performance when only basic functionalities, such as agent creation and migration, were needed. The first version of mobile agent platform Crossbow [1,2] was developed as a core platform, which can be used for researching complex mechanisms such as agent communication, cooperation, security, etc. Its fundamental requirements were to provide the basic functionalities (agent creation and migration), perform well, be easy to use and be executable in a heterogeneous environment. Although the platform fulfilled these basic requirements, recently the need for a structurally developed agent platform according to FIPA standards appeared.

After analyzing different types of software development process models, we decided to make the Crossbow2 platform using the incremental model. The incremental model [4] combines the linear sequential model with the iterative philosophy of prototyping. The result of the first increment in the process of developing Crossbow2 was an agent platform that could create an agent. In the second increment, the platform was extended with the functionality of locating agents. In the third increment, agent migration was enabled and, finally, in the fourth increment agent communication and destruction were added.

The paper is organized as follows. Section II describes the requirements that the platform needs to fulfill. Section III describes the design of the platform. Finally, Section IV describes how the platform was implemented and tested.

II. REQUIREMENTS

Software requirements are used to describe the functionalities, needs and constraints that are associated with a software product. Correctly defined requirements are the first technical step in the software development process and one of the main conditions for high quality software.

As a result, the first crucial step of the project was to define the software requirements. We analyzed existing platforms and determined that agent platform Crossbow2 [5] should have basic functionalities such as agent creation, migration, communication, detection and destruction. After defining the requirements, it was necessary to specify them with more details before going on to design phase.

An agent is created after a platform receives a request from a user or from another agent. Since a user and an agent can both give tasks to other agents it is important that they can create agents for that purpose. Unique agent identification information has to be returned to the user or the other agent, depending who demanded agent creation. An agent's home platform is the agent platform it was created on. An agent must be able to migrate to another agent platform so it could perform a task that requires its presence on a remote location. At the same time, the agent must notify its home platform about its current location in the network. Agents and users must be able to locate other agents by contacting the agent’s home platform and requesting information about its current location in the network. An agent can move to another location on its own request or upon receiving a request from another agent or user. An agent can communicate with the user or other agents regardless of its and their current location in the network. Communication initiator must know unique agent identification information of destination agent in order to start communication with it..For security reasons, an agent can be destroyed only upon a user’s demand.

III. DESIGN

In software design, software requirements are analyzed in order to obtain the internal structure of the software that will be used as a construction basis. Software design describes the architecture of the software and how it is decomposed and organized into components. One of the main requirements in developing Crossbow2 was that the new agent platform must be developed according to FIPA standards. For that purpose, the FIPA agent management specification was carefully analyzed. The FIPA agent platform is described in Section III.A. Additional components that were build in the Crossbow2 agent platform are described in Section III.B and the messages used for communication are described in Section III.C.

A. FIPA agent management specification

The agent management reference model [6] consists of the logical components described in this section.

An **Agent Platform (**AP**)** ensures the physical infrastructure is convenient for deploying agents. The AP consists of the machine(s), the operating system, the agent support software, FIPA agent management components (Directory Facilitator, Agent Management System and Message Transport Service) and agents.

An **agent** is a computational process that implements the autonomous, communicating functionality of an application. Agents are the fundamental actors on an AP. An agent must have at least one owner; it can be the user or another agent, depending on organizational affiliation. An agent must support at least one notion of identity known as the Agent Identifier (AID) which distinguished him unambiguously within the Agent Universe.

A **Directory Facilitator (**DF**)** is an optional component of the AP. The DF provides yellow pages services to agents. Agents can register their services with the DF or they can find out from the DF what services other agents offer. An **Agent Management System (**AMS**)** is a mandatory component of the AP. There can be only one AMS in an AP. It is the job of the AMS to control and supervise access and use of the AP. The AMS also offers white pages services to other agents. The **Message Transport Service (**MTS**)** is the default communication method used between agents on different APs. **Software** describes all the non-agent executable collections of instructions that can be accessed through an agent.

B. The Crossbow2 architecture

The Crossbow2 agent platform is based on the FIPA architecture but is not FIPA compliant yet since it does not support ACL messages. In Figure 1 components taken from the FIPA agent management specification are shown in regular text**,** while new components are marked **bold**. Those components which were not implemented are marked *italic*.

Each agent has exactly one owner, which is a user or another agent, depending on who demanded its creation. Each agent also has exactly one *Agent Identifier* (AID) that is composed of the *Agent Platform Identification* (APID) and the time of creation. The APID consists of the IP address of the platform, the TCP port at which the platform is running and the type of connection (socket or SSLsocket). An agent can be contacted only on the platform at which it is currently present. Agents communicate by exchanging messages.

There is only one AMS in Crossbow2 AP. It maintains a directory of *Agent Envelopes* (AE). An AE is the agent's only contact with the rest of the AP; it consists of the agent and its *Message Repository*. The AMS also has a *Location Repository* (LR) that maintains a directory of AIDs from agents created on that platform, along with their current location in the network. The LR also maintains a directory of AIDs from agents currently present on that platform. AMS has an *IDFactoy* whose task is to create AIDs and the APID.

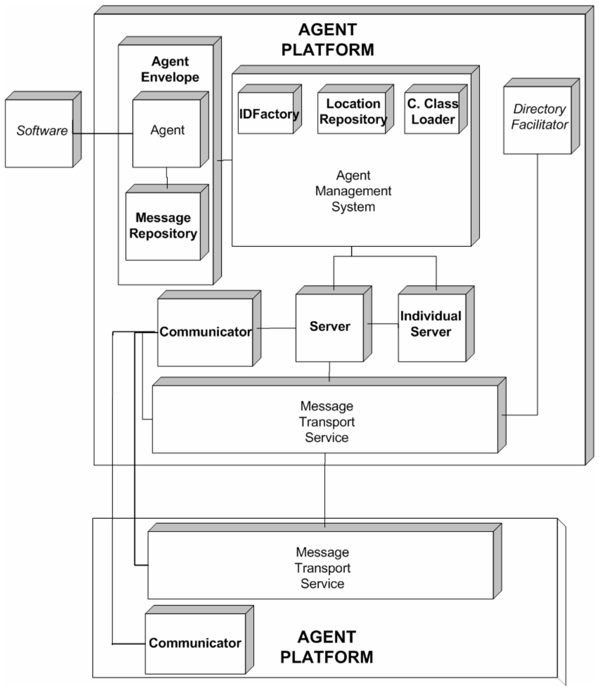

Figure 1. Crossbow2 arhitecture

The MTS waits for connection requests from other platforms or users and establishes the corresponding connection by creating a *Communicator* object. This object handles any further communication between the two platforms or user-platform communication. Communication between platforms is pluggable. It is possible to use either socket or the SSL socket protocol to communicate with another platform or user. Also, it is possible to implement some other protocol if this is requested.

There is no additional software for agents. It is assumed that the developers which are going to implement other agent functionalities will create the necessary software.

The *MessageRepository* (MR) contains all of the agent's messages. After a message arrives it is put at the end of the list. An agent processes the first message in the list and then deletes it.

The *Server* is responsible for sending outgoing messages and receiving messages from other platforms. The *Server* also creates *IndividualServers* (IS) that disburden the server by handling specific tasks such as processing received messages.

C. *Messages*

Messages are the basic means of communication in the Crossbow2 AP. There is a total of twelve different types of messages. Every message has an ID. For every message, the sender and the receiver are known. In case the *agent* sends (receives) the message, it contains the agent's ID and its current location. If the *agent platform* sends (receives), the message then contains only the APID.

A user or agent can send a *CreateNewAgentMessage* to the AP in case they want to create a new agent. After the platform receives such a message, it creates a new agent, along with its AE and MR. After the new agent is created, the user or the agent receive an *AgentCreatedMessage* that confirms the creation of the new agent and contains its AID.

The server creates a *LocationRequestMessage* in case an agent or user wants to communicate with another agent, but the Server doesn't know the agent’s current location. The server always sends the *LocationRequestMessage* to the home platform of the agent whose location it is interested in. The home platform of that agent looks into the LR for the agent's current location and returns that information as a part of the *LocationResponseMessage*.

An agent or user can send a *RequestToMoveMessage* to an agent if they want to move him to another platform. This message contains the location to where the agent will be moved. The message is put into the agent's MR. The agent can move only after it processes the messages that arrived before the *RequestToMoveMessage*. When an agent moves to another location it sends two messages. The first one is a *MoveMessage* that contains the AID and URL from where the *Agent* class is going to be loaded, and the second one is an *AgentMessage* which will perform agent creation after it arrives at the destination platform. Every time an agent arrives to a platform, it sends a *CurrentLocationMessage* to its home platform. After this message arrives to the agent's home platform, its current location is stored in the platform's LR in case other agents or users are interested in communicating with him.

Agents communicate with users and other agents by sending a *CommunicationMessage.* The communication initiator can request to receive a response to its message.

A user can send a *DestroyAgentMessage* in case he wants to destroy an agent. After receiving it, the platform destroys the agent, together with its AE and MR. When the agent is destroyed, the platform that destroyed him sends an *AgentDestroyedMessage* to its home platform so that the destroyed agent is removed from its home platform LR.

A *GoodbyeMessage* is sent when two platforms are ending their communication. The *Communicator* objects that were used for that communication are destroyed. This message enables the recycling of communication resources.

IV. IMPLEMENTATION AND TESTING

Extreme Programming (XP) is a new light-weight approach to developing software [8]. It is used by small to medium-sized teams. Writing tests before implementation is one of its main characteristics and that was the main reason we decided to perform our implementation and testing using XP.

The Crossbow2 AP was developed by two people through 5 weeks. Analyzing mobile agent platform Crossbow and the first iteration of Crossbow2 development lasted 2 weeks. The second, third and fourth iterations each lasted one week.

Crossbow2 is written in Java using the Eclipse IDE version 3.0 [7]. The XP practices [8] used in Crossbow2 development are: small releases, simple design, unit testing, refactoring, pair programming, continuous integration, collective ownership, on-site customer and coding standards.

Small releases – simple functionality was implemented and put into function; subsequent versions were released in short cycles, enabling the customer to always see what was done. We think this practice is good because the customer was pleased to see the progress and this had a positive impact on the programmers’ efficiency. Furthermore, this practice had a positive effect on our motivation, since we started to see the results of our work very soon and were even more motivated to continue with the next iteration.

Simple design – the system was designed as simply as possible at the given moment. Since the time predicted for completing the platform was limited, this practice saved us valuable time so we had enough time to implement all the basic functionalities. This practice was very useful since it taught us that simple solutions are often the better ones. At the beginning, we started to develop a complex system but very soon it became difficult to find out why certain components were not doing what we wanted them to do. It became particularly more difficult to locate where exactly the mistake was made.

Testing – unit tests were continually written to increase our confidence in the written code Testing was done with JUnit 3.8 [9] to ensure automatic and quick testing. This practice caused chaos at the beginning; it was very unusual and confusing to write these tests at first. As a result of this, the first week we wrote tests after the implementation and than slowly started to adjust and write tests before the implementation. Once we got used to it, our productivity increased because we didn’t spend a lot of time debugging code in case of a mistake.

Refactoring – the system was restructured to remove duplication, add flexibility and simplify without changing its behavior. There was no actual refactoring in the first increment. We got a real idea what refactoring really represents when we started with the second increment. By that time, some of the existing basic functionalities needed to be restructured so that they could be used for more complex functionalities. We realized that it is easier to modify the existing functionality than to write a new one that is just slightly different. Since we had tests, we ran them after each small modification to ensure that the old functionalities were still working properly. In case the tests failed, repairing the mistakes didn't present a problem.

Pair programming – two programmers wrote the code at one machine. This practice was used to increase productivity and it has its good and bad sides. The good side is that both programmers learn from each other, although this is not always the case. Since one of the programmers was more experienced, the "weaker" programmer had more profit from pair programming by learning a lot more than the "stronger" one. The bad side of this practice is that when one programmer has a bad day, it reflects on the work atmosphere and causes tension between the programmers. This claim has shown to be "the truth, the whole truth and nothing but the truth".

Collective ownership – both programmers could change the code at anytime. This practice was not questionable since there were only two programmers, both with equal status and, thus, one could not forbid to the other to modify the code.

Continuous integration – the system was built and integrated several times a day. Every time a task was completed, integration was performed using CVS. This practice is better when there are more pairs working on the same code. In our case, CVS was used primarily as a backup of the latest accurate version.

On-site customer – the customer was always available to answer all of the developer's questions and to see if the current version was functioning correctly. This practice was of the highest importance to us since it saved us a lot of time and energy that would have been spent developing certain functionalities in the way we thought was correct.

Coding standards – the code was written according to the rules. The main purpose was to improve communication and lighten the code understanding. The Maven CheckStyle Plugin [10] was used to check the code's compliance with the coding standards. This was very annoying at first because we had our own code style. The number of errors according to the coding standards was very high the first few times we ran the style check. However, after spending at least an hour on correcting the code style we started to write code according to the standard. This required a lot of self-control but we got used to it.

The result of the first increment was a platform that could create an agent. First we wrote the tests and then the implementation of the AMS and *IDFactory*. The next step was creating an AE and the agent. When the platform receives a *CreateNewAgentMessage,* an agent is created; The MTS and *Server* were created to enable sending and receiving messages. The functionality of creating new agents was added to the agent. An *AgentCreatedMessage* was made to return the AID of the created agent to its owner. In the first increment, an agent could create another agent only on the platform at which it was currently present.

In the second increment, we enabled agent location. For this purpose, an LR was created along with the *LocationRequestMessage* and *LocationResponseMessage* messages. We added the functionality of sending a *CurrentLocationMessage* after an agent is created so that the agent would be added into the LR. Since a *LocationRequestMessage* can be sent to another AP, the next step was to enable creating *Communicator* objects.

The task of the third increment was to enable agent migration with the help of *RequestToMoveMessage*, *MoveMessage* and *AgentMessage* messages. Since the *RequestToMoveMessage* is put in the MR, we first tested and implemented the MR. After the agent is moved to the AP, it also sends a *CurrentLocationMessage* to its home platform.

In the fourth and last increment, communication between agents and between an agent and user was enabled. A *CommunicationMessage* was introduced into the platform, and ISs were added to disburden the *Server*. *DestroyAgentMessage* and *AgentDestroyedMessage* messages enabled the destruction of an agent. The last functionality added was recycling communication resources with the help of a *GoodbyeMessage*.

CONCLUSION AND FUTURE WORK

The Crossbow2 Agent Platform was developed by using XP practices and it's the architecture was made according to FIPA standards. XP practices were shown to be very helpful and had a significant impact on increasing programmer productivity. Practices such as testing, on-site customer and simple design were especially helpful. Pair programming and the coding standard were useful and helped the programmers increase their personal skills. Since Crossbow2 is a light-weight Agent Platform, in the future it could be used in mobile connections for personal services. In order to be FIPA compliant, communication with the ACL messages is going to be implemented in the following iterations .

REFERENCES

[1] D. Voncina, V. Vyroubal "Crossbow manual", Faculty of Electrical Engineering and Computing, Zagreb 2002.

[2] M. Kusek, D. Voncina, V. Vyroubal "Design and Implementation of Mobile Agent Platform Crossbow", MIPRO 2004 – CTI (Information & Telecommunication), Opatija, Croatia, 2004, pp. 82-87.

[3] I. Crnkovic, M. Larsson "Building Reliable Component-Based Software Systems", Artech House, 2002.

[4] R. S. Pressman "Software Engineering: A practitioner's Approach", McGraw-Hill Publishing Company, London 2000.

[5] V. Gostovic, A. Petric "Agent Platform – Crossbow2", Proceedings of the Fourth Summer Camp 2004, Zagreb July 5 – September 17, 2004, pp. 67-87.

[6] Foundation for Intelligent Agents "FIPA Agent Management Specification" document number SC00023K, http://www.fipa.org/specs/fipa00023/SC00023K.html

[7] Eclipse Integrated Development Environment, www.eclipse.org

[8] K. Beck, C. Andres "Extreme Programming Explained: Embrace Change", Addison-Wesley Professional, 2004

[9] JUnit, Testing resources for Extreme Programming, www.junit.org

[10] Apache Maven Projects, maven.apache.org