Mobile Intelligent Agents in Erlang

Stefan Mandl, Raymond Bimazubute, and Herbert Stoyan

Department of Computer Science 8 (Artificial Intelligence)
University of Erlangen-Nürnberg,
am Weichselgarten 9, D-91058 Erlangen, Germany
{mandl,rbimaz,hstoyan}@cs.fau.de

Abstract

We present a vision for distributed application programming in peer-to-peer networks based on a modified version of the Procedural Reasoning System that we call ePRS. The original PRS is expanded to provide mobility between nodes, which may be spread over a network of computers. The node concept is strongly linked to the concept of computation environments. We show that the goal-based nature of ePRS agents makes them useable even in environments that were unknown at the time they were defined. The Erlang language, which is used as implementation language provides the basic networking technology. An example from the domain of database consistency is proposed.

1 Introduction

By definition, in peer-to-peer networks every machine potentially plays two roles, that of a client and that of a server. Therefore there is no single location that hosts an application's information and in addition, there is no single location that hosts the program logic. This means that distributed applications in peer-to-peer networks have to deal with incompatible servers and clients, either because the network's software update strategy is insufficient or because of principal reasons of the architecture. For example, certain operations may not be possible on certain platforms. Commonly, solutions to these problems rely on protocols and versioning to ensure that only compatible peers communicate and that the messages have the same semantics to both of them.

In contrast, intelligent mobile software agents offer the possibility to operate in the described inconsistent environments by 1) abstracting from the particular implementation of operations and 2) continue operation on a different hosts without stopping the computation.

Distributed application programming comes to its strengths when the functionality of the application can be split into mostly independent sub parts. The higher the dependence between the sub parts, the more communication overhead will be involved. A classic example for a distributed system is that of a movie renderer that assigns picture rendering tasks to various machines in a compute-cluster thereby gaining a speed-up that is proportional to the number of machines employed. In such a system, the "worker-nodes" run equal programs and serve as a means to distribute the load.

Another typical scenario for distributed application programming is the case when some resources that are needed are not available on any system the application is supposed to run on. In section 6, we present an example which deals with a very valuable resource: Users, that are sitting at different locations, each working on the same application. This kind of application is commonly referred to as "groupware" or collaborative application. Such systems appear to be implementable in a very natural way by choosing an agent-based implementation.

A very important part of real-world distributed applications is security and error-handling. These aspects have been excluded from the following but shall receive more attention in the future.

The next section gives a short review on intelligent agents. Section 3 explains the idea behind a distributed application. Section 4 introduces the idea of computation environments and abstract actions. Section 5 presents the implementation of ePRS. Section 6 reviews the design of a proposed distributed application. Section 7 gives a conclusion and hints for future research.

2 Intelligent Agents

There are many different definitions of "agent" in the literature [4]. In this paper, we assume agents as autonomous entities perceiving and acting in some kind of environment. Intelligent agents are agents that act in an intelligent way. To avoid a philosophical discussion on the term "intelligent", we state the simplifying working

definition, that any agent that does some amount of reasoning to select the most appropriate action in the current situation shall be called intelligent.

To this end, Shoham attributed mental capabilities like commitments and beliefs to an agent's data structures [12]. This makes it easier to evaluate the appropriateness of an agent's actions, as the judgment on the appropriateness involves categories that are well known to humans. For example, it is easier to state whether or not an agent chose a clever action when it is known what goals and intentions the agent has.

In BDI (belief, desire, intentions) systems [8], the mental capabilities are divided into

Beliefs: The agent's view on the state of the world
Desires: The agent's goals
Intentions: The agent's current plans to bring about a desired goal

Using these categories, an intelligent agent could be defined as an agent that selects its intentions in such a way that the agent's goals are reached. In contrast, non-intelligent agents select their actions in a predefined manner.

A very important characteristic of intelligent agents is adaptability, which means that the agent reacts appropriately to changes in the environment. This could for example cause a goal to be dropped as it is not reachable anymore. Obviously, in a peer-to-peer network, this property is of great importance when a peer is temporarily offline, causing certain actions to become unavailable. In that case, the agent is able to postpone the intentions that rely on these actions and productively continue working on goals that are not affected by this circumstance. When the network is reachable again, the agent could re-awake the postponed intentions.

3 Distributed Applications

An *application* is a program that gives a computer instructions that provide the user with tools to accomplish a task. A program written so that the processing can be divided across multiple computers over a network is called a *distributed* application. Taking the two definitions together, and assuming that a set of intelligent agents may be called a program, distributed applications could be implemented by a number of agents running on a number of computers in a network. This means that both, the program and the data (embodied in the agents beliefs) are spread over the network. The actual way this distribution is handled is dependent on the application. Typical examples are settings, where certain resources that are crucial to the completion of a task are not available on the same computer system. For example, to produce a piece of cloth, there is need for a weaving machine and a after that a dyeing machine, which may be located at different factories; to book a flight ticket at the ticket counter, the clerk needs an easy-to-use user interface through which he is accessing a large flight database concurrently with hundreds of colleagues at different locations.

The examples show the nature of distributed applications. The implementation of such systems is mostly client-server based with a strict seperation between clients and servers, which makes it necessary to maintain large bodies of code and keep them consistent. Also, the server often marks a single point of failure, degrading the reliability of the system. Additionally, a single server is a performance bottleneck for two reasons: Two many requests and network traffic.

The employment of intelligent mobile agents, offers the flexibility needed for such large scale applications.

4 Environments: Abstracted Actions

We define an ePRS agent's computation environment as the set of possible actions it could take, the set of possible plans it could follow, and the number of other agents that it is aware of. The major contribution of ePRS is the idea to make the actions and plans part of the environment instead of being internal features of the agent. For the agent programmer, this leads to a different style of agent programming, as the actual procedure to reach a certain goal is not accessible to her. Only the initial beliefs and goals have to be specified. A major part of the implementation of such a system is the implementation of the computation environments.

For the future, we plan to implement more of the original PRS's meta-level reasoning, allowing agent-local plans to update the goals and influence goal-handling in a agent local way, thus giving the agents more different qualities.

In order to implement computation environments, we adopt the node concept of Erlang (more on Erlang in section 5. A node is a entity that is capable of running ePRS agents and computation environments. Nodes may reside on different computers in a network; there can be multiple nodes per machine. Every node provides exactly one computation environment, thus the only way an agent can change its computation environment is to move to a different node where another environment is available.

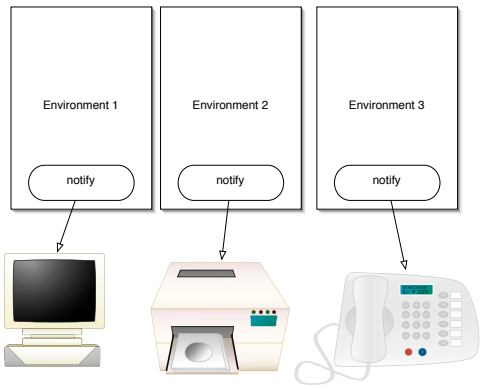

Fig. 1. Different environments offering different implementations

If any node in the system provides the same environment, the agents can act in a load-balanced way, thereby gaining efficiency, but this provides no conceptual advantage over a centralized solution. If on the other hand the different nodes provide different environments, it is possible to maintain different aspects of the application on different nodes.

This means that the same goal will possibly be mapped to different actions, depending on the environment the agent is running in (see figure 1).

The fact that also the plans are provided externally by the computation environment has an interesting consequence. When an agent intends a plan, this plan is put into the agent's intention structure. When the agent migrates to a new environment, it takes the intended plans from the old environment with it. This makes it possible to import plans from other environments. As plans do not consider action specifications that would depend on the environment, but only specifications of new sub goals, the imported plan might very well be followed in the new environment. If one of the goals can not be satisfied in the new environment, the intention is postponed and may be re-awaked when the agent is moving to a different environment.

If the goal that is not satisfy-able is the only active goal remaining, the agent may decide to migrate to a different host, where the goal can be handled.

5 ePRS Implementation

The Procedural Reasoning System (PRS) is the classical BDI language. It was defined by Georgeff and Lansky in 1987 [6] and is in part based on experiences from the previous Procedural Expert System (PES) [5] and [7]. It serves as a reasoning system situated in a dynamic world and operating under real-time constraints and resource limitations. A PRS agent has four data structures: beliefs, goals, plans, and the intention structure. The intention structure contains plans that the system has decided to execute. This decision is made based on the agents beliefs and goals. The PRS system interpreter activates plans and executes them. Plans are called *Knowledge areas* in the original PRS or *Acts* in CL-PRS [11]. We use the terms "plan" and "act" interchangeably.

Erlang is a mostly functional programming language which was designed at the Ericsson and Ellemtel Computer Science Laboratories in Sweden [1]. There exists a commercial implementation for which professional support is available. In addition, there is an open source implementation that is available free of charge for non-commercial purposes [3]. Erlang has support for several high-level features like distributed programming, multi-threading, advanced communication, and error-handling. We call such a language an "Agent Implementation Language", as these features make it very attractive for agent programming, but it is lacking the agent concept at the language level, e.g. there is no primitive called `agent`.

No practical programming language—even a functional one—is free of side-effects. To avoid side-effects helps in the implementation of mobility. In Erlang, there are only three kinds of side-effects:

– Each thread (Erlang process) has an environment dictionary that can be altered by the commands `put` and `get`. For ePRS, we don't use these commands.

– Message-passing between Erlang processes has the obvious side-effect of sending a message. As we want to model distributed communicating intelligent agents, we can not do without.
– External functions can be linked to the Erlang system. Commonly these functions will have side-effects. We may use this feature in the future for accessing external resources.

In section 5.1, we explain the implementation of the ePRS interpreter. Section 5.2 describes how agent migration is realized.

5.1 Basic Operation

Every ePRS agent follows a certain procedure that is motivated by the execution cycle of PRS[6]. This procedure is the same for every ePRS agent. It handles the updating of the agents beliefs, goals and intentions. We like to call it an abstract machine for executing goal-oriented programs with procedural knowledge. In PRS, the procedural knowledge is made up of a number of plans an agent could take to reach a certain goal. Plans have a two part representation. The first part consists of a cue which indicates, whether or not a plan is to be considered for a certain goal, an environment part, which is mainly used to initialize variables from the agent's beliefs, and a resource section which describes which resources have to be acquired in order to execute a plan. The second part of a plan is the so-called plot, which is a directed graph, which contains step-by-step the sub goals that have to be reached in order to reach the goal the plan was designed for. It is possible to specify parallel plan steps in an "and/or" fashion. There are goals of different kinds: *achieve*, *achieve-by*, *conclude*, ... Our current implementation of ePRS only supports achieve-goals. Plan selection (intention) and execution sets up new sub goals but does not directly invoke actions. This is the task of an extra step in the interpreter. For every primitive action—which is obtained by calling the function `actions()` on the current node—, the system tests whether or not it is applicable in the current situation and providing the solution to a intended goal. If this is the case, the action is chosen for execution, before any compound plans are even considered. If such an action can not be found, the list of available plans—available through the function `plans()`—is searched for a candidate plan to reach a current goals. If such a plan is found, it is put in the intention-structure.

In ePRS, we represent primitive actions as Erlang functions like the one in figure 2. These functions return a tuple of functions. The first component function of the tuple tests whether or not the action is applicable to the current situation, imitating the the first part of PRS plans. The second component function of the tuple actually implements the action. The argument to the primitive action function is a goal specification which possibly contains variables that are preserved in the environment of the test and action functions that are returned.

Plans are represented in a similar way. There is a goal cue, which is the parameter of a function, which returns a tuple of test and plot functions which represent the plan. CL-PRS employs a complex graph library to represent the plot of a plan. We use the fact that actual execution follows a tree. In ePRS, we represent this tree by a list of functions that dynamically determine the index of the next function to be called from the list.

As described in the previous section, act nodes do not cause real action to happen, but change the set of goals and beliefs of the agent.

In ePRS, plans and actions are not part of the agent but provided from the environment. This is different from other BDI implementations like JACK [2], where the reactions to plan steps are part of the Agent. This means, that plans and actions have to be specified once for an environment and no agent specific action methods have to be supplied.

5.2 Mobility

Erlang is a programming language that has distribution built-in. The basic concept of distributed Erlang is the *node*. Nodes have names that can be chosen to be unique in the local network or world-wide. Every process is running on a certain node. It is also possible to launch a new process on a different node. There are additional language features that allow the monitoring of local or remote processes. For an Erlang node to become a ePRS node, it must be running a certain server process, which has to be registered as `eprs_server` on that node. By now, the only kind of message this process can handle is the request to start a new agent. This is done by starting a new Erlang process which executes the agent's main loop. It is clear now that an agent is transfered by starting a new process on a different machine. Thus to be mobile, the complete state of the agent must be explicitly embodied in the arguments to the agent's main loop. So we have to refrain from using the process dictionary to store the agents beliefs or goals. The kind of mobility implemented by our system is commonly called *weak migration*. In contrast to systems like Agent TCL [10], [9], that provide *strong migration*, the system always starts in a certain state on the new location. As ePRS is a goal oriented system, there is

```
migrate_host({achieve,[at,Host]}) ->
     {fun(Beliefs) ->
          true
     end,
     fun(Beliefs) ->
          migrate(Host),
          [at,Host]
     end};
migrate_host(_) ->
     {fun(_) -> false end, fun(_) -> false end}.
```

Fig. 2. Erlang code for the primitive action `migrate_host`.

no predefined flow of execution, thus the difference between weak migration and strong migration is not so obvious. It reduces to the fact that migration routines can not be used in Erlang functions that actually implement the actions.

6 Example: Database consistency

We plan to test ePRS on a proposed application to find inconsistencies in a database containing files of historical persons from the european royal societies from the last several centuries. Current relational database systems offers some support for keeping databases consistent:

- Restricted forms for data input
- Data types for entries
- A simple language to specify constraints on the data
- Triggers which allow for user defined test procedures when data is accessed in the database

It is obvious, that these features are not sufficient to find all errors in such databases as ours, as it contains a lot of real world data from different ages. In addition, it might well be that some of the data has been corrupted with bad faith in the course of history; a fact of great importance and interest to a historian. Obviously, automatic correction of entries in such a database may not be feasible at all. One has to possess a large amount of knowledge and experience to identify and — even harder — to correct wrong entries. In addition, there may be experts on certain kinds of errors (like name spelling) that are novices on other kinds of errors. We envision a system where experts can provide their own knowledge by specifying agent environments, with action implementations that reflect the very properties of their domain, still the expert does not have to actually program the agent as the abstract goals are expected to be similar for every domain.

Additionally, we expect that the implementation of the application will be a long-term issue where different strategies for finding inconsistencies will be tried. Therefore, we like to be able to implement a new strategy for detecting a certain kind of error while keeping the system up and running. By providing agent environments in a network, we won't have to halt the execution of the other parts of the system and still can test our changes by temporarily bringing up a new node containing the new implementation.

One could even imagine some new kind of interactive application development: The system runs in a preliminary version, reaching at a goal that is not satisfy-able. Instead of dropping the goal, the system announces the fact and the programmer may be able to update the environment code in order to serve this goal. It is the goal-oriented nature of ePRS, that suggests such a kind of development process.

7 Conclusion and Future Research

We presented a vision for distributed application programming. An application is made up of a number of agents traveling in a peer-to-peer system, choosing appropriate actions at appropriate location. We identified the existence of actions and plans that are external to the agent as a way to bring this ideal closer. Still, a lot of work needs to be done. There is the need for a high-level system specification language which allows the concise definition of nodes and the environments that are available on them. Additionally, we need a new agent programming language that is able to cope with the concept of external actions.

References

1. J.Armstrong, R.Virding, C.Wikström,M.Williams: *Concurrent programming in Erlang*. Prentice Hall, Englewood Cliffs, New Jersey, Second Edition
2. Paolo Busetta, Ralph Ronnquist, Andrew Hodgson, and Andrew Lucas. *Jack intelligent agents - components for intelligent agents in java*. AgentLink News Letter, January 1999. White paper
3. `http://www.ericsson.se/erlang`, `http://www.erlang.org`
4. Stan Franklin and Art Graesser: *Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents*. Proceedings of the Third International Workshop on Agent Theories, Architectures and Languages, Springer-Verlag, 1996
5. M.P.Georgeff, A.L.Lansky: *A System for Reasoning in Dynamic Domains: Fault Diagnosis on the Space Shuttle*. Technical Note 375, SRI International, Menlo Park, California, 1986
6. M.P.Georgeff, A.L.Lansky, M.J.Schoppers: *Reasoning and Planning in Dynamic Domains: An Experimentation with a Mobile Robot*. Technical Note 380, Artificial Intelligence Center, SRI International, Menlo Park, California, 1987
7. M.P.Georgeff, A.L.Lansky: *Procedural Knowledge*. Technical Note 411, SRI International, Menlo Park, California, 1987
8. M.Georgeff, B.Pell, M.Pollack, M.Tambe, and M.Wooldridge: *The Belief-Desire-Intention Model of Agency*. Proceedings of the 5th International Workshop on Intelligent Agents V : Agent Theories, Architectures, and Languages (ATAL-98), Springer Publishers, 1999
9. Robert Gray: *Agent Tcl: A flexible and secure mobile-agent system*. In The Fourth Annual Tcl/Tk Workshop Proceedings. The USENIX Association, 1996
10. Kotay, K. and Kotz, D.: *Transportable Agents*. In Yannis Labrou and Tim Finin, editors, Proceedings of the CIKM Workshop on Intelligent Information Agents, Third International Conference on Information and Knowledge Management (CIKM 94), Gaithersburg, Maryland, December 1994
11. *Procedural Reasoning System User's Guide*. Artificial Intelligence Center, SRI International, Menlo Park, California, 2001
12. Y.Shoham: *Agent-oriented programming*. AI Magazine, Vol.60, No.1, 1993, p.51-92