1999

An Agent-Based Design for Problem Solving Environments

Dan C. Marinescu

Report Number:
99-017

AN AGENT-BASED DESIGN FOR PROBLEM SOLVING ENVIRONMENTS

Dan C. Marinescu

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

CSD #99-017
May 1999

An Agent-Based Design for Problem Solving Environments

Dan C. Marinescu
(dcm@cs.purdue.edu)
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907, USA

Abstract

In this paper we examine alternative means to exploit the advantages of code mobility, object-oriented design, and agent technology for high performance distributed computing. We describe an infrastructure for Problem Solving Environments based upon software agents.

1 Introduction

A number of new initiatives and ideas for high performance distributed computing have emerged in the last few years. Object-oriented design and programming languages like Java open up intriguing new perspectives for the development of complex software systems capable to simulate physical systems of interest to computational sciences and engineering. The Java Grande initiative aims to add new constructs and to support optimization techniques needed to make the Java language more expressive and efficient for numerical simulation. If successful, this effort will lead to more robust scientific codes and increased programmer productivity.

An important side effect of the use of Java in scientific computing is code mobility. This brings us to another significant development, computing grids. Informally, a computing grid is a collection of autonomous computing platforms with different architectures, interconnected by a high-speed communication network. Computing grids are ideal for applications that have one or more of the following characteristics: (a) are naturally distributed, data collection points and programs for processing the data are scattered over a wide area network, (b) need a variety of services distributed over the network, (c) have occasional or sustained needs for large amounts of computing resources e.g. CPU cycles, large memory, vast amounts of disk space, (d) benefit from heterogeneous computing environments consisting of platforms with different architectures, (e) require a collaborative effort from users scattered over a large geographic area.

Many problems in computational sciences and engineering could benefit from the use of computing grids. Yet, scientific code mobility, a necessary condition for effective use of heterogeneous computing environments is a dream waiting to materialize. Porting a parallel program from one system to another, with a different architecture and then making it run efficiently are tedious tasks. Thus the interest of the high performance distributed computing comunity for Java.

We are cautiously optimistic regarding both aspects outlined above. At the time of this writing, Java code is still inefficient, it runs 10 to 20 times slower than C code. It is rather unlikely that the Java language will ever include a comprehensive support for numerical computations because scientific and engineering applications represent only a relatively small fraction of the intended audience for Java. Even if Java becomes the language of choice for writing scientific and engineering codes, we still have a large body of legacy codes written along the years and an "ab inito" approach, re-writing them in Java is unlikely. We need also to keep in mind that often, parallel codes require new algorithms to execute efficiently, thus code mobility in a heterogeneous system has inherent limitations. Resource management in a network of autonomous nodes and security pose formidable challenges that need to be addressed before computing grids could become a reality.

In this paper we examine alternative means to exploit the advantages of code mobility, object-oriented design, and agent technology for high performance distributed computing, at a time when Java Grande and computing grids are only a promise. To bridge the gap between promise and reality we propose to develop an infrastructure for Problem Solving Environments, PSEs, based upon software agents.

To use a biological metaphor [2], software agents form a nervous system and perform command and control functions in a Problem Solving Environment. The agents themselves relay on a distributed object system to communicate with another. Though the agents are mobile, some of the components of the PSE are tightly bound to a particular hardware platform and cannot be moved with ease.

In this paper we first review basic requirements for designing complex systems like Problem Solving Environments and the role of software agents, then we introduce a software architecture that has the potential to facilitate the design and implementation of PSE and to make the resulting systems less brittle.

The basic design philosophy of the Bond system is described in [1], [?] [2], the security aspects of Bond are presented in [8], an application of Bond to the design of software agents for a network of PDE solvers is discussed in [11] and an in depth view of the design of Problem Solving Environments using Bond is given in [10]. The Bond system was released in mid March 1999.

2 Agents, Problem Solving Environments, and Software Composition

Software agents seem to be at the center of attention in the computer science community. Yet different groups have radically different views of what software agents are, [6], what applications could benefit from the agent technology, and many have a difficult time sorting out the reality from fiction in this rapidly moving field and accepting the representation that software agents provide a "magic bullet" for all problems. The concept of an agent was introduced by the AI community a decade ago, [3]. An AI agent exhibits an autonomous behavior and has inferential abilities. A considerable body of work is devoted to agents able to meet the Turing test by emulating human behavior [9]. Such agents are useful for a variety of applications in science and engineering e.g. deep space explorations, robots, and so on.

Our view of an agent is slightly different [1]. For us a software agent is an abstraction for building complex systems. An agent is an *active mobile object that may or may not have inferential abilities.* Our main concern is to develop a constructive framework for building collaborative agents out of ready-made components and to use this infrastructure for building complex systems including Problem Solving Environments, PSEs. The primary function of a Problem Solving Environment is to assist computational scientists and engineers to carry out complex computations involving multiple programs and data sets. We use the term workflow and metaprogram interchangeably, to denote both the static and the dynamic aspects of this set of computations. We argue that there are several classes of high performance computing applications that can greatly benefit from the use of agent-based PSEs:

- Naturally distributed applications,
- Data intensive applications.
- Applications with data-dependent or non-deterministic workflows.
- Parallel applications based upon domain data decomposition and legacy codes.

Many problems in computational science and engineering are naturally distributed, involve large groups of scientists and engineers, large collections of experimental data and theoretical models, as well as multiple programs developed independently and possibly running on systems with different architectures. Major tasks including coordination of various activities, enforcing a discipline in the collaborative effort, discovering services provided by various members of the team, transporting data from the producer site to the consumer site, and others can and should be delegated to a Problem Solving Environment. The primary functions of agents in such an environment are: scheduling and control, resource discovery, management of local resources, use-level resource management, and workflow management.

Data-intensive applications are common to many experimental sciences and engineering design applications. As sensor-based applications become pervasive, new classes of data-intensive applications are likely to emerge. An important function of the PSE is to support data annotation. Once metadata describing the actual data is available, agents can automatically control the workflow, allow backtracking and restart computations with new parameters of the models.

Applications like climate and oceanographic modeling often relay on many data collection points and the actual workflow depends both upon the availability of the data and the confidence we have in the data. The main function of the agents in such cases is the dynamic generation of the workflows based upon available information.

Last, but not least, in some cases one can achieve parallelism using sequential legacy codes. Whenever we can apply a divide and conquer methodology based upon the partitions of the data into sub-domains, solve the problem independently in each sub-domain, and then resolve with ease the eventual conflicts between the individual workers we have an appealing alternative to code parallelization. The agents should be capable to coordinate the execution and mediate conflicts.

The idea of building a program out of ready-made components has been around since the dawn of the computing age, backworldsmen have practiced it very successfully. Most scientific programs we are familiar with, use mathematical libraries, parallel programs use communication libraries, graphics programs rely on graphics libraries, and so on.

Modern programming languages like Java, take the composition process one step further. A software component, be it a package, or a function, carries with itself a number of properties that can be queried and/or set to specific values to customize the component according to the needs of an application which wishes to embed the component. The mechanism supporting these functions is called *introspection*. Properties can even be queried at execution time. *Reflection* mechanisms allow us to determine run time conditions, for example the source of an event generated during the computation. The reader may recognize the reference to the Java Beans but other *component architectures* exists, Active X based on Microsoft's COM and LiveConnect from Netscape to name a few.

Can these ideas be extended to other types of computational objects besides software components, for example to data, services, and hardware components? What can be achieved by creating *metaobjects* describing *network objects* like programs, data or hardware? We use here the term "object" rather loosely, but later on it will become clear that the architecture we envision is intimately tied to object-oriented concepts. We talk about network objects to acknowledge that we are concerned with an environment where programs and data are distributed on autonomous nodes interconnected by high speed networks.

Often the components are legacy codes that cannot be modified with ease. In this case we can wrap around legacy programs newly created components called software agents. Each wrapper is tailored to the specific legacy application. Then interoperability between components is ensured by federation of agents.

3 An Infrastructure for Problem Solving Environments

Bond, [1], is a distributed-object, message-oriented system, it uses KQML [5], as a meta-language for inter-object communication. KQML offers a variety of message types (performatives) that express an attitude regarding the content of the exchange. Performatives can also assist agents in finding other agents that can process their requests. A performative is expressed as an ASCII string, using a Common Lisp Polish-prefix notation. The first word in the string is the name of the performative, followed by parameters. Parameters in performatives are indexed by keywords and therefore order-independent.

The infrastructure provided by Bond supports basic object manipulation, inter-object communication, local directory and local configuration services, a distributed awareness mechanisms, probes for security and monitoring functions, and graphics user interfaces and utilities.

Shadows are proxies for remote objects. Realization of a shadow provides for instantiation of remote objects. Collections of shadows form *virtual networks* of objects.

Residents are active Bond objects running at a *Bond address*. A resident is a container for a collection of objects including communicator, directory, configuration, and awareness objects. *Subprotocols* are closed subsets of KQML messages. Objects inherit subprotocols. The discovery subprotocol allows an object to determine the set of subprotocols understood by another object. Other subprotocols, monitoring, security, agent control, and the property access subprotocol understood by all objects.

The transport mechanism between Bond residents is provided by a *communicator* object with four interchangeable communication engines based upon: (a) UDP, (b) TCP, (c) Infospheres, (info.net), and (d) IP Multicast protocols.

Probes are objects attached dynamically to Bond objects to augment their ability to understand new subprotocols and support new functionality. A *security probe* screens incoming and outgoing messages to an object. The security framework supports two authentication models, one based upon *username, plain password* and one based upon the*Challenge Handshake Authentication Protocol, CHAP*. Two access control models are supported, one based upon the *IP address (firewall)* and one based upon an *access control list*. *Monitoring probes* implement a subscription-based monitoring model. *An autoprobe* allows loading of probes on demand.

The *distributed awareness* mechanism provides information about other residents and individual objects in the network. This information is piggy-backed on regular messages exchanged among objects to reduce the overhead of supporting this mechanism. An object may be aware of objects it has never communicated with. The distributed awareness mechanism and the discovery subprotocol reflect our design decision to reduce the need for global services like directory service and interface repositories.

Several distributed object systems provide support for agents. Infospheres

(//www.infospheres.caltech.edu/) and Bond are academic research projects, while IBM Aglets (www.trl.ibm.co.jp/aglets/index.html) and Objectspace Voyager (//www.objectspace.com) are commercial systems.

A first distinctive feature of the Bond architecture, described in more detail in [1] is that agents are native components of the system. This guarantees that agents and objects can communicate with one another and the same communication fabric is used by the entire population of objects. Another distinctive trait of our approach is that we provide middleware, a software layer to facilitate the development of a hopefully wide range of applications of network computing. We are thus forced to pay close attentions to the software engineering aspects of agent development, in particular to software reuse. We decided to provide a framework for assembly of agents out of components, some of them reusable. This is possible due to the agent model we overview now.

We view an agent as a finite-state machine, with a strategy associated with every state, a model of the world, and an agenda as shown in Figure 1. Upon entering a state the strategy or strategies associated with that state are activated and various actions are triggered. The model is the "memory" of the agent, it reflects the knowledge the agent has access to, as well as the state of the agent. Transitions from one state to another are triggered by internal conditions determined by the completion code of the strategy, e.g. success or failure, or by messages from other agents or objects.

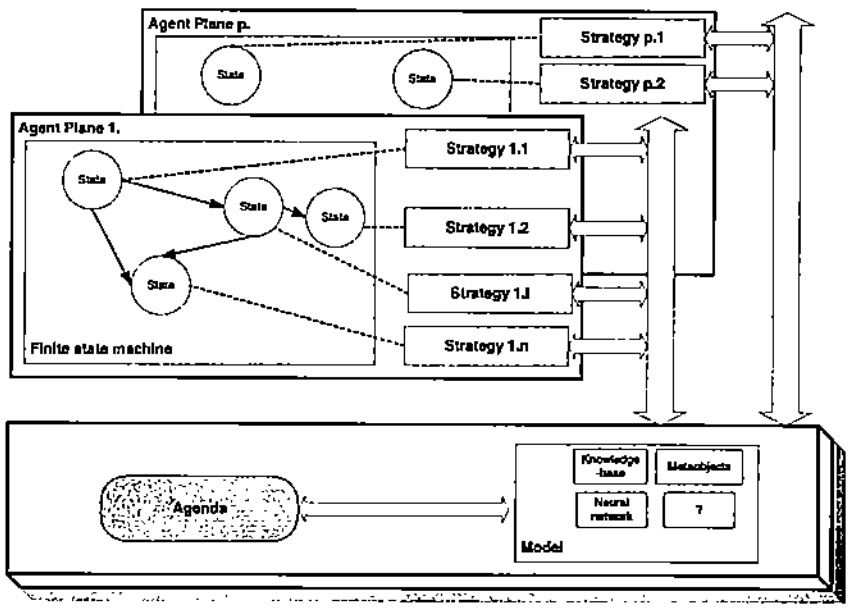

Figure 1: The abstract model of a Bond Agent

The finite-state machine description of an agent can be provided at mul-

tiple granularity levels, a course-grain description contains a few states with complex strategies, a fine-grain description consists of a large number of states with simple strategies. The strategies are the reusable elements in our software architecture and granularity of the finite state machine of an agent should be determined to maximize the number of ready made strategies used for the agent. We have identified a number of common actions and we started building a strategy repository. Examples of actions packed into strategies are: starting up one or more agents, writing into the model of another agent, starting up a legacy application, data staging and so on. Ideally, we would like to assemble an agent without the need to program, using ready-made strategies from repositories.

Another feature of our software agent model is the ability to assemble an agent dynamically from a "blueprint", a text file describing the states, the transitions, and the model of the agent. Every Bond-enabled site has an "agent factory" capable to create an agent from its blueprint. The blueprint can be embedded into a message, or the URL of the blueprint can be provided to the agent factory. Once an agent was created, the agent control subprotocol can be used to control it from a remote site.

In addition to favoring reusability, the software agent model we propose has other useful features. First, it allows a smooth integration of increasingly complex behavior into agents. For example, consider a scheduling agent with a mapping state and a mapping strategy. Given a task and a set of target hosts capable to execute the task, the agent will map the task to one of the hosts subject to some optimization criteria. We may start with a simple strategy, select randomly one of the target hosts. Once we are convinced that the scheduling agent works well, we may replace the mapping strategy with one based upon an inference engine with access to a database of past performance. The scheduling agent will perform a more intelligent mapping with the new strategy. Second, the model supports agent mobility. A blueprint can be modified dynamically and an additional state can be inserted before a transition takes place. For example a "suspend" new state can be added and the "suspend" strategy be concatenated with the strategy associated with any state. Upon entering the "suspend" state the agent can be migrated elsewhere. All we need to do is send the blueprint and the model to the new site and make sure that the new site has access to the strategies associated with the states the agent may traverse in the future. The dynamic alteration of the finite state machine of an agent can be used to create a "snapshot" of a group of collaborating agents and help debug a complex system.

We have integrated into Bond the JESS expert shell developed at Sandia National Laboratory as a distinct strategy able to support reasoning. Bond messages allow for embedded programs written in JPython and KIF.

Agent security is a critical issue for the system because the ability to assemble and control agents remotely as well as agent mobility, provide unlimited opportunities for system penetration. Once again the fact that agents are native Bond objects leads to an elegant solution to the security aspect of agent design. Any Bond object, agents included, can be augmented dynamically with a security probe providing a defense perimeter and screening all incoming and

outgoing messages.

The components of a Bond agent shown in Figure 1 are:

- **The model of the world** is a container object which contains the information the agent has about its environment. This information is stored in the form of dynamic properties of the model object. There is no restriction of the format of this information: it can be a knowledge base or an ontology composed of logical facts and predicates, a pre-trained neural network, a collection of meta-objects or different forms of handles of external objects (file handles, sockets, etc).

- The **agenda** of the agent, which defines the goal of the agent. The agenda is in itself an object, which implements a boolean and a distance function on the model. The boolean function shows if the agent accomplished its goal or not. The distance function may be used by the strategies to choose their actions.

- The **finite state machine** of the agent. Each state has an assigned strategy which defines the behavior of the agent in that state. An agent can change its state by performing *transitions*. Transitions are triggered by internal or external *events*. External events are messages sent by other agents or objects. The set of external messages which trigger transitions in the finite state machine of the agent defines the *control subprotocol* of the agent.

- Each state on an agent has a **strategy** defining the behavior of the agent in that state. Each strategy performs actions in an infinite cycle until the agenda is accomplished or the state is changed. Actions are considered atomic from the agent's point of view, external or internal events interrupt the agent only between actions. Each action is defined exclusively by the agenda of the agent and the current model. A strategy can terminate by triggering a transition by generating an internal event. After the transition the agent moves in a new state where a different strategy defines the behavior.

All components of the Bond system are objects, thus Bond agents can be assembled dynamically and even modified at runtime. The behavior of an agent is uniquely determined by its model (the model also contains the state which defines the current strategy). The model can be saved, transferred over the network.

A `bondAgent` can be created statically, or dynamically by a factory object `bondAgentFactory` using a *blueprint*. The factory object generates the components of the agent either by creating them, either by loading them from persistent storage. The agent creation process is summarized in Figure 2

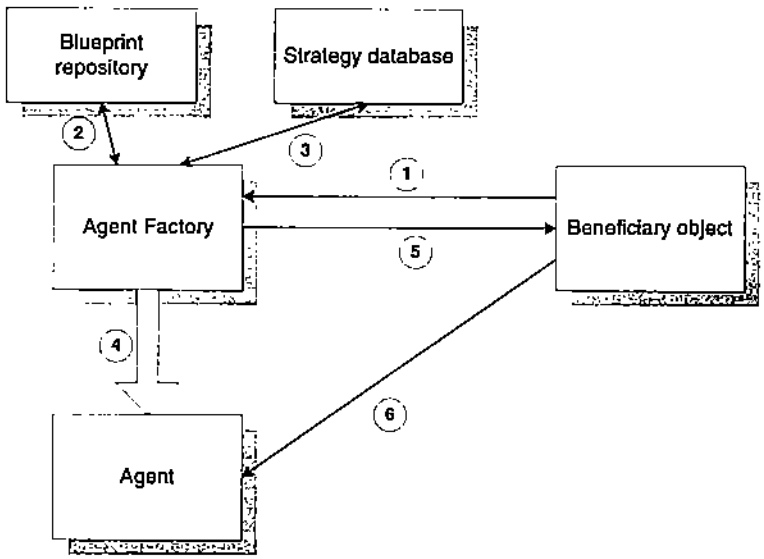

Figure 2: Creating an agent remotely using an agent factory. (1) The beneficiary object sends a create-agent message to the agent factory (2) The blueprint is fetched by the agent factory from a repository or extracted from the message (3) The strategies are loaded from the strategy database (4) The agent is created (5) The id of the agent is communicated back to the beneficiary, and (6) The beneficiary object controls the new agent

4 Conclusions

Bond is a Java written, agent-based, distributed-object system we have developed for the past few years. Bond provides a framework for interoperability based upon (1) metaobjects that provide information about network objects, and (2) software agents capable to use the information about network objects and carry out complex tasks.

Some of the components we propose e.g. the agent framework, the scheduling agents, the monitoring and security frameworks, are generic and we expect that they will be included in other applications like distance learning, or possibly electronic commerce.

The design of a Problem Solving Environment based upon a network of PDE solvers is one of the applications of Bond that illustrates the advantages of a component based architecture versus a monolithic design of a PSE.

A beta version of the Bond system was released in mid March 1999 under an open source license, LPGL, and can be downloaded from our web site, **http://bond.cs.purdue.edu**.

Acknowledgments

The work reported in this paper was partially supported by a grant from the National Science Foundation, MCB-9527131, by the Scalable I/O Initiative, and by a grant from the Intel Corporation.

References

[1] Bölöni, L., and D.C. Marinescu, *An Object-Oriented Framework for Building Collaborative Network Agents.* Kluever Publishers, 1999 (to appear).

[2] Bölöni, L., R. Hao, K.K. Jun, and D.C. Marinescu, *Structural Biolgy Metaphors Applied to the Design of a Distributed Object System*, Proc. Second Workshop on Bio-Inspired Solutions to Parallel Processing Problems, in LNCS, vol 1586, Springer Verlag, 1999, pp. 275-283.

[3] Bradshaw, J. M., *An Introduction to Software Agents*, in J. M. Bradshaw Ed. *Software Agents*, MIT Press, pp. 3-46, 1997.

[4] *The Grid, Blueprint for a New Computing Infrastructure*, Foster, I. and C. Kesselman, Eds., Morgan Kaufmann, (1998).

[5] Finn, T., Y. Labrou, and J. Mayfield, *KQML as an Agent Communication Language*, in J. M. Bradshaw Ed. *Software Agents*, MIT Press, pp. 291-316, 1997.

[6] Franklin, S. and A. Graesser, *Is it an Agent, or just a Program?*, Proceedings of the Third International Workshop on Agent Theories, Architectures and Languages, Springer Verlag, 1996.

[7] Genesereth, M. R., *An Agent-Based Framework for Interoperability*, in J. M. Bradshaw Ed. *Software Agents*, MIT Press, pp. 317-345, 1997.

[8] Hao, R., L. Bölöni, K.K. Jun, and D.C. Marinescu, *An Aspect-Oriented Approach to Distributed Object Security*, Proc. 4-th IEEE Symp. on Computers and Communications, IEEE Press, (1999), (in print).

[9] Jennings, N. R., K. Sycara, M. Woolridge, *A Roadmap of Agent Research and Development*, in Autonomous Agents and Multi-Agent Systems, 1, pp. 275-306, 1998.

[10] Marinescu, D.C., and Bölöni L., *A Component-Based Architecture for Problem Solving Environments*, 1999, (in preparation).

[11] Tsompanopoulou, P., L. Bölöni, D.C. Marinescu, and J.R.Rice, *The Design of Software Agents for a Network of PDE Solvers* Proceedings of Workshop on Autonomous Agents, IEEE Press, 1999 (in press).