Structural Biology Metaphors Applied to the Design of a Distributed Object System

Ladislau Bölöni, Ruibing Hao, Kyungkoo Jun, and Dan C. Marinescu

Computer Sciences Department
Purdue University
West Lafayette, IN 47907, USA
boloni, hao, junkk, and dcm@cs.purdue.edu

Abstract. In this paper we present the basic ideas of a distributed object middleware for network computing. We argue that when building complex systems out of components one can emulate the lock and key mechanisms used by proteins to recognize each other.

1 Overview

A number of biological analogies have found their way into computer science. Neural networks provide an alternative to von Neumann architecture [2–4], genetic algorithms [5] are used to solve optimization problems, mutation analysis was proposed for software engineering. The question we are concerned with is if these analogies can be further expanded to cover the way we build complex systems out of components and emulate genetic mechanisms and the immune system [10].

In this paper we present a design philosophy for network computing middleware, within the larger context of building complex systems out of ready-made components. Inherently, a complex computing system is heterogeneous and accommodating the heterogeneity of the hardware and the diversity of the software is a main concern of such a design. This heterogeneity is a constant source of pain for those intent on solving problems with computers. The alternative to heterogeneity and diversity is uniformity, but is this an exciting or a feasible alternative? Though we thoroughly enjoy the diversity of biological and social systems we complain when faced with the diversity of computer systems. Here we argue that hardware heterogeneity and software diversity can be accommodated provided that we transfer to computers themselves the tedious tasks of this endevour. Nature uses composition to build very complex forms of life and as we understand the principles and mechanisms that form the foundation of life we should try to emulate them to build more dependable and easy to use computing systems.

The cornerstones of the architecture we propose are metaobjects, entities that provide a description of the network objects, and software agents that perform operations on network objects. Examples of network objects are programs, data, hardware components including hosts and communication links, services, and so on. Metaobjects allows us to annotate the network objects.

The Bond project was triggered by a collaboration with structural biologists who provided the problems, the motivation to design a Virtual Structural Biology Laboratory, and need to learn some basic facts about the structure of biological macromolecules. The complex procedures needed for data acquisition, data analysis and model building for x-ray crystallography and electron microscopy are discussed elsewhere [9, 12–14]. Here we only note that processing of structural biology data involves large groups and facilities scattered around the world, complex programs that are modified frequently. Most computations are data intensive, they require the use of parallel and distributed systems.

2 Biological Systems

Nature uses composition to build extremely complex structures [1]. There are 20 amino-acids, the basic building blocks of life. The amino-acid sequence of a protein's peptide chain is called a *primary structure*. Different regions of the structure form local regular *secondary structure* such as alpha helices and beta strands. The *tertiary structure* is formed by packing such structural elements onto globular units called *domains*. The final protein may contain several polypeptide chains arranged in a *quaternary structure*. By formation of such tertiary and quaternary structures amino-acids far apart in the sequence are brought together in three dimensions to form a functional region, an *active site* [1]. The three dimensional structure of a protein determines its function, the disposition in space and the type of the atoms in a region of the protein provide a *lock* that can be recognized by other proteins that may bind to it, provided that they have the proper *key*. Living organisms *mutate*, the atomic structure of their cells changes and a *selection* mechanisms ensures the survival of those able to perform best their function.

Biological cells carry with them *genetic material, RNA and DNA* that describe the sequence of amino-acids in every protein. How this information is used to actually build the protein, the so-called *folding problem* is not elucidated yet. Moreover proteins with different sequences of amino-acids may fold to identical or very similar 3D atomic structures. They may have different properties, e.g. thermal stability, but they will perform the same functions because a fundamental principle is that the *structure determines the function of a biological specimen*. Another fundamental principle in genetics is *genetic economy*. *Symmetry* plays an important role in building complex cells. Spherical viruses have an icosahedral symmetry, they look like a soccer ball, are composed out of wedges with identical structure. The virus core contains the genetic material and has only one copy of the DNA or RNA sequence of an "unit cell", the wedge mentioned above.

3 Composition, Reflections, and Introspection

Let us now turn our attention to software composition. 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.

The set of properties embedded into a metaobject provide a "lock and key" mechanism similar with the one used by proteins to recognize one another. In a *workspace* populated with metaobjects, one can envision mechanisms to link network objects together to form a *metaprogram*, a new metaobject, capable of carrying out a well defined computational task. Example: to link a data object to a software object we need to search the workspace for a metaobject describing a crystallographic FFT program able to compute the 3D Fourier transform of a symmetric object with a known symmetry, given a lattice of real numbers describing the "unit cell", the building block of the object. At each step, the selection process succeeds if the target metaobject possesses a set of "keys", corresponding to the desired properties needed for composition. This is a deliberate example, anyone familiar with FFTs recognizes that creating the metaobjects describing the elements discussed above is a non trivial task.

We expect a metaobject to contain *genetic information*, i.e. a description of all relevant properties of a network object . This information must be in a form suitable for machine processing. For example a metaobject associated with a software component should include a description of the functions and interfaces of that component using a descriptive language like IDL, Interface Description Language. IDL reveals only the interfaces of an object and does not specify how these properties are to be *folded* into an actual implementation. The genetic information associated with a data object should reveal its ancestry, the characteristics of the sensor that has generated the data, or provide links to the program that has produced the data and to its input data objects. The genetic information would allow an autonomous agent to generate an actual implementation of the code in case of a software component, or a human contemplating the results of a sequence of computations to trace back decisions made at some point in the past.

We also need to add new properties to a metaobject. For example when an agent discovers that a processor executes incorrectly a sequence of instructions this new property should be added to the metaobject describing that particular class of processors or processors with Ids within a certain range. A fundamental principle is that acquired traits take precedence over inherited ones.

Both hardware and software are created as a result of an *evolutionary* process. Occasionally, new programs, or microprocessors, are created from scratch, but often, new versions are upgrades of existing objects, that inherit many characteristics of the older versions. Inheritance has the potential to simplify the task of building metaobjects and agents capable to manipulate them intelligently.

Care must be taken to expose in the metaobject only *stable properties* of the network resources. For example the amount of main memory installed on a system is a stable property, though it may change, such changes are likely to be infrequent, while the load placed upon the system varies rapidly, it is a *transient property* and should not be exposed.

Once we recognize that creation of metaobjects is a difficult task we have to ask ourselves if metaobjects and the network resources need to be tightly or loosely coupled with each other. The tightly coupled approach would require that the network object and the metaobject be kept together to ensure consistency. There are fundamental flaws with the tightly coupled argument. First, this "ab-initio" approach would require re-creation of legacy components, software, hardware and data, to fit our scheme. Second, information would be unnecessarily duplicated and confidential information compromised.

By virtue of the arguments discussed above a metaobject should only have a *soft link* to the network object it describes. Different properties of the network object may be accessible on a need to know basis, e.g. the source code may be available only to those who have the need for it. The metaobject associated with a program may contain attributes describing the function of the program, its input and output. It should also provide references or pointers to the: source code, the executables for different platforms, the human readable documentation of the program, the implementation notes, an error log, and so on. Following the principle of genetic economy, components of the metaobject would be shared among all the users of the object.

The price to pay for the distributed object approach is that inconsistency between the metaobjects and the network objects are occasionally unavoidable. We argue that in a network rich environment catastrophic consequences of such inconsistencies are unlikely to occur. Moreover, whenever possible, discovery agents should update the metaobjects.

In summary, we propose to create metaobjects describing properties of network objects. These properties should reveal how objects can be composed together using a lock and key mechanism and support a selection mechanism to eliminate components that do not perform their functions well. Linking metaobjects together can only be done based upon a universally accepted *taxonomy of objects* and properties, and a given context. We acknowledge the fact that a considerable amount of work in the area of knowledge sharing still remains to be done, but we believe that a system built along the principles discussed above will allow a larger segment of the population to use computers to solve complex tasks.

4 An Architecture for Communicating Objects

This section introduces Bond, an agent-based, object-oriented middleware for network computing. The goals of the Bond system are to: (a) facilitate access to network resources, (b) support collaborative activities, and (c) accommodate software diversity and hardware heterogeneity.

Metaobjects and agents are important classes of Bond objects. Metaobjects provide synthetic information about network objects and agents use this information to access and manipulate network objects on behalf of users.

The first question we have to answer when designing an agent-based, distributed-object architecture is the level of integration of agents and objects. Should the agents must be tightly integrated into the architecture, or can they be added on to an existing distributed-object framework? To answer this question we examine two critical issues, communication and the complexity of the system.

Communication is a critical issue in a distributed system and though procedure-oriented and message-oriented paradigms are arguably dual to each other, most distributed-object systems favor remote method invocation and synchronous communication. The conventional wisdom is that remote method invocation is more efficient than message passing and that asynchronous communication is considerably more difficult to program and debug than synchronous one. Thus if we want to design a complex system, capable to evolve and to exhibit a good level of performance the only choice is synchronous remote method invocation.

On the other hand, *agent-oriented programming* favors the message-oriented approach and asynchronous communication. Since agents can exhibit a complex behavior, remote method invocation is too restrictive and inter-agent communication is based upon message passing, using agent-communication languages like KQML. An agent has the ability to make autonomous decisions and should be able to adapt its behavior to the response to a message or to the lack of a response. In a *wide-area distributed system*, components including other agents or objects, and communication links may fail, and the lack of a response should still lead to predictable behavior. The ability to respond to component failure is an important argument in favor of integrating software agents into network computing systems.

An approach often encountered in the literature, is to support remote method invocation for regular objects and message passing for agents. We believe that this dichotomy increase the software complexity and creates additional problems when regular objects and agents need to communicate with each other. Our solution is to design a pure message passing system with all objects communicating using KQML. An agent is just another object extended with the ability to make autonomous decisions, it can communicate with any other object and the communication fabric is common to the entire object population.

Important as it may be, the fact that every object "speaks" KQML, in other words understands the syntax of a KQML message and is able to parse it, does not guarantee that two objects can cooperate with each other. An object should be able to "understand" semantically a message and respond to it in a coherent manner. The granularity of objects can differ widely, ranging from simple objects like an address, a date, or a message, to complex objects like a scheduling agent or an authentication server. A server is an active object capable to provide a well-defined set of services, though it may be very complex a server does not have the autonomy an agent must exhibit. The ability of an object to "understand" a message is also very different.

To solve the problem of semantical understanding of a message we introduce the concept of *subprotocols*. A subprotocol is a "dialect" understood by a subset of objects. A subprotocol is a closed set of messages, an object understands semantically every single message in a subprotocol and it is able to respond with a message in the same subprotocol. This guarantees the ability of agents to cooperate. An agent may use a *discovery subprotocol* and determine a set of common subprotocols with another agent or object. Once a group of agents discover that they all understand a subprotocol any agent sending a message belonging to the common subprotocol is guaranteed to get a meaningful response. An object can determine with ease if it understands a message because every message is stamped with the subprotocol it belongs to. A subprotocol is inherited based upon object hierarchy. When a new object extending an existing one is created, in addition to the subprotocols inherited, new ones can be defined. The only subprotocol understood by all objects is the *property access subprotocol*, all agents understand the *agent control subprotocol*. The ability of an existing agent may be enhanced with *probes*, special objects capable to handle other aspects of a complex system, e.g. security, monitoring, accounting, and so on. We also have the ability to generate subprotocols dynamically once the communication patterns of a group of agents are known.

4.1 Component-based agents

A first distinctive feature of the Bond architecture 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. 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*. 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.

The finite state machine description of an agent can be provided at multiple *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 database. 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.

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, without altering other aspects of its behavior, provided that the model is not altered by 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 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 is to 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.

4.2 Metaobjects

Metaobjects are other critical components of our system, they glue together network objects and agents. Network objects or resources are: hardware platforms, communication links, sensors, programs, libraries, services, data files, and so on. The function of metaobjects is to provide the knowledge needed by the software agents to manipulate network objects.

The metaobjects form a distributed knowledge base and their defining attribute is that besides a name and a unique Id their structure is fluid, it changes with time to reflect our knowledge about a world populated with network objects. The ever-changing relationship amongst them is the other important trait of metaobjects. Metaobjects are suitable to represent knowledge while traditional objects in the object-oriented programming sense have well-defined inheritance and best represent entities with a well-known set of properties. There are no methods specific to a metaobject other than to set and reset their attributes. Agents are capable to manipulate metaobjects and invoke functions to determine if the metaobject provides the information needed by a specific strategy. The metaobjects provide the means to integrate object-oriented and agent-oriented programming.

To illustrate these ideas consider the scheduling agent example discussed earlier. The agent must be able to determine if a particular host is suitable for a computational task. We have the choice of creating a regular object with a number of attributes to reflect the most relevant properties of a host e.g. its IP address, the type and Id of the processor, the number of processors, the amount of main memory, and so on. We may ask ourselves what is a relevant property for a host, for example are the type of the graphics cards, or the number of geometric engines important? The answer is that in some cases the graphics hardware may not be present and in other cases we may not be interested in graphics when scheduling the task. Rather than defining a *host object* with a potentially very large set of attributes we define a metaobject with no other property but a name and add dynamically properties to it as needed.

Some of the traits of the network object are discovered at the time the metaobject was created others reflect changes in the network object itself discovered by some agent at a later point in time. For example a *discovery agent* will fill in the metaobject associated with a host, a number of ¡attribute, attribute-value¿ pairs, including the type and the Id of the processor. A selection strategy of the scheduling agent will match the attributes desired by the task with the ones offered by a specific host as reflected by the corresponding metaobject.

In this paper we describe an object-oriented, agent-based, distributed object system based upon active messages. The system exercises the metaphors discussed above. An alpha release of the Bond system is available for downloading at `http://bond.cs.purdue.edu`.

5 Acknowledgments

The work reported in this paper is partially supported by the National Science Foundation grants BIR-9301210 and MCB-9527131, by the California Institute of Technology, under the Scalable I/O Initiative, by a grant from Intel Corporation, and by the Computational Science Alliance and the NCSA at the University of Illinois.

References

1. C. Branden and J. Toose. *"Introduction to Protein Structure"* Garland Publishing 1991.
2. F. Rosenblatt, *"The perceptron: A perceiving and recognizing automaton"*, 85-460-1, Project PARA Cornell Aeronautical Laboratory Ithaca, NY, 1957.
3. J. Hopfield *Neural Networks and Physical Systems with Emergent "Collective Computational Abilities"* Proceedings of the National Academy of Sciences, 1982 vol.79, pp. 2554.
4. J. L. McClelland and D. E. Rumelhart, *"Parallel Distributed Processing: Explorations in the Microstructures of Cognition, Volume 2: Psychological and Biological Models"*, MIT Press Cambridge, Mass. 1986
5. J. R. Koza, *"Genetic Programming: On the Programming of Computers by Means of Natural Selection"*, MIT Press 1992.
6. L. Bölöni. *"Bond Objects – A White Paper."* Department of Computer Sciences, Purdue University CSD-TR #98-002.
7. L. Bölöni, Ruibing Hao, K. K. Jun, and D.C. Marinescu. *"Subprotocols: an object-oriented solution for semantic understanding of messages in a distributed object system"*, September 1998, (submitted).
8. K. Mani Chandy. *"Caltech Infospheres Project Overview: Information Infrastructures for Task Forces."*
9. M.C. Cornea Hasegan, Z. Zhang, R.E. Lynch, D. C. Marinescu, A. Hadfield, J.K. Muckelbauer, S. Munshi, L. Tong and M.G. Rossmann. *"Phase Refinement and Extension by Means of Non-crystallographic Symmetry Averaging using Parallel Computers."* Acta Cryst D51, pp 749-759, 1995
10. J.C. Fabre and T. Perennou. *"A Metaobject Architecture for Fault Tolerant Distributed Systems: The FRIENDS Approach"* IEEE Trans. on Computers, Vol 47, No 1, pp 78-95, 1998.
11. T. Finin, et al. *"Specification of the KQML Agent-Communication Language"* DARPA Knowledge Sharing Initiative draft, June 1993.
12. R. E. Lynch, D.C. Marinescu, H. Lin, and T. S. Baker. *"Parallel Algorithms for 3D Reconstruction of Asymmetric Objects from Electron Micrographs"* September 1998, (submitted).
13. I. Martin, D.C. Marinescu, R. E. Lynch, and T. S. Baker. *"Identification of Spherical Virus Particles in Digitized Images of Entire Micrographs"* Journal of Structural Biology, 120, pp 146-157, 1997.
14. I. Martin and D.C. Marinescu *"Concurrent Computations and Data Visualization for Structure Determination of Spherical Viruses"* IEEE Computational Science and Engineering, 1998 (in press).
15. R. J. Stroud. *"Transparency and Reflection in Distributed Systems"* ACM Operating Systems vol. 22, no. 2 pp. 99-103, 1993.