1998

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

Ladislau Bölöni

Ruibing Hao

Kyungkoo Jun

Dan C. Marinescu

Report Number:
98-037

STRUCTURAL BIOLOGY METAPHORS APPLIED TO THE DESIGN OF A DISTRIBUTED OBJECT SYSTEM

Ladislau Boloni
Ruibing Hao
Kyungkoo Jun
Dan C. Marinescu

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

CSD-TR #98-037
November 1998

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

Ladislau Bölöni, Ruibing Hao, Kyungkoo Jun, and Dan C. Marinescu
(boloni, hao, junkk, and dcm@cs.purdue.edu)
Computer Sciences Department
Purdue University
West Lafayette, IN 47907, USA

October 30, 1998

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.

Contents

1 Overview

A number of biological analogies have found their way into computer science. Neural networks provide an alternative to von Neumann architecture [2], [3], [4], [5], genetic algorithms [6] 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, to emulate genetic mechanisms and the immune system [14], [19].

The heterogeneity of our computing hardware and the diversity of the computer software are a constant source of pain for those intent on solving problems with computers. 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 heterogeneity and diversity are a true blessing and not a form of modern plague, provided that we learn how to transfer to computers themselves the more tedious tasks needed to accommodate heterogeneity and diversity. 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 alternative to heterogeneity and diversity is uniformity, but is this an exciting prospect?

In this paper we present the design philosophy of Bond, an infrastructure project in scientific computing, 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.

The Bond project was triggered by a collaboration with structural biologists who provided the problems, the motivation, 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 [12], [13], [16], [17], [18]. Here we only note that processing of structural biology data involves large groups and facilities scattered around the world, complex programs that are changed frequently. Most computations are data intensive, they require the use of parallel and distributed systems.

2 Biological Systems, Composition, Introspection, and Metaobjects

Nature uses composition to build extremely complex structures [1]. There are 20 aminoacids, the basic building blocks of life. The aminoacids 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 aminoacids 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.

Let us briefly examine the mechanisms used by the immune system. The human immune system provides a first line of defense against viruses. It first detects the presence of a virus and then produces antibodies that bind to the active site of the virus cell. A virus cell may have several millions atoms. To disable a virus cell, an antibody cell must know the conformation of the binding site(s) of the virus at the atomic level. A man made antiviral drug provides a second line of defense and can only be designed if the atomic structure of the virus is known.

Biological cells carry with them genetic material, RNA and DNA which describe the sequence of aminoacids 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 aminoacids 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.

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 resources* 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 resources to acknowledge that we are concerned with a distributed 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 them together to form a *metaprogram*, a new metaobject, capable of carrying out a well defined computational task. Example: to link a data metaobject to a software metaobject we need to search the workspace for 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 the metaobject to contain a description of all relevant properties of a network resource including "genetic information". This information must be in a form suitable for machine processing by an *autonomous agent*. For example a software metaobject should include a description of the functions and interfaces of that component using a descriptive language like IDL, Interface Description Language, which 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.

Does the effort to build metaobjects seem daunting? We should expect it according to the biological analogy discussed above. The task of abstracting the properties of network resources is monumental, one can only succeed if the network resources, software, data, and hardware, are classified into *disjoint classes* each with well-defined properties and *inheritance* mechanisms. We also need to add new properties to any metaobject. A fundamental principle is that acquired traits take precedence over inherited ones.

We favor the development of software components in an object-oriented language like Java which supports inheritance mechanisms. Building software in an object-oriented framework is essential because it exposes only essential properties of the software component and hides the details of the implementation of each function.

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. The latest release of a program may only have a small number of known new properties and only those need to be included into its information object, the rest of the properties are inherited from its ancestor. For example if the input data format has not changed, the new version will inherit the data format of the old one. If we discover that the latest version of the program in some cases produces incorrect results this fact becomes a new property and an agent will be able to avoid the latest version of the program if it acts in behalf of an application that exercises the incorrect behavior of the program. Similar examples can be constructed for hardware components. A Pentium Pro processor is downwards compatible with a Pentium processor and its information object need not describe the common set of instructions but only new instructions as well as faulty instruction sequences if they exist. Therefore inheritance has the potential to simplify the task of building information objects 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 two components are 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. Every site running Microsoft Word would need to store a huge amount of information including its genetic component (the source code) that Microsoft is unlikely to reveal willingly.

By virtue of the arguments discussed above a metaobject should only have a soft link to the network resource it describes. Different properties of the network resource 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 metaobject approach is that inconsistency between the metaobjects and the network resources are occasionally unavoidable. We argue that in a network rich environment catastrophic consequences of such inconsistencies are unlikely to

occur.

In summary, we propose to create metaobjects describing properties of network resources. 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. In this paper we present an infrastructure supporting the design principles presented in this section.

3 An Architecture for Communicating Objects

This section introduces Bond, a Java-based, object-oriented middleware for network computing. Network computing is a paradigm that emphasizes the use of network resources, computing resources distributed across the network, over local resources. Network resources are hosts, programs and data. Middleware is a software layer that allows developers to mold systems tailored to specific needs from components and develop new components based upon existing ones. 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.

The basic components of Bond are metaobjects and agents. Metaobjects provide synthetic information about network resources and agents use this information to access and manipulate network objects on behalf of users. The basic philosophy of the Bond middleware is to allow developers and users to define metaobjects and agents using the *frameworks* provided by the system.

Bond can be used in stand-alone mode or to connect to a Bond domain. A Bond domain consists of a society of users with common interests and a number of services. In stand-alone mode a user may create metaobjects and agents and execute locally several demos. To create a Bond domain one needs to activate the servers provided with the package.

Bond is a message-oriented system, it uses KQML [15], 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.

3.1 Objects and containers

A *Bond Object* is collection of data fields and methods, [7]. The data component of an object consists of several *fields* and a vector of *dynamic properties*. Each object has a *unique bondId*. The granularity of Bond objects ranges from simple objects, e.g. a message, to complex objects like a directory server. Some objects are *passive*, e.g. a message or a metaobject associated with a data file, others are *active* they have one or more threads of control. Active as well as passive objects can communicate using the `say()` method [7]. All objects are persistent. A segment of the Bond object hierarchy is presented in Figure 1. The full Bond object hierarchy is presented elsewhere.

A *container* is a Bond object that encapsulates a collection of objects. A message for a remote object is always delivered to the container where the object is located and the

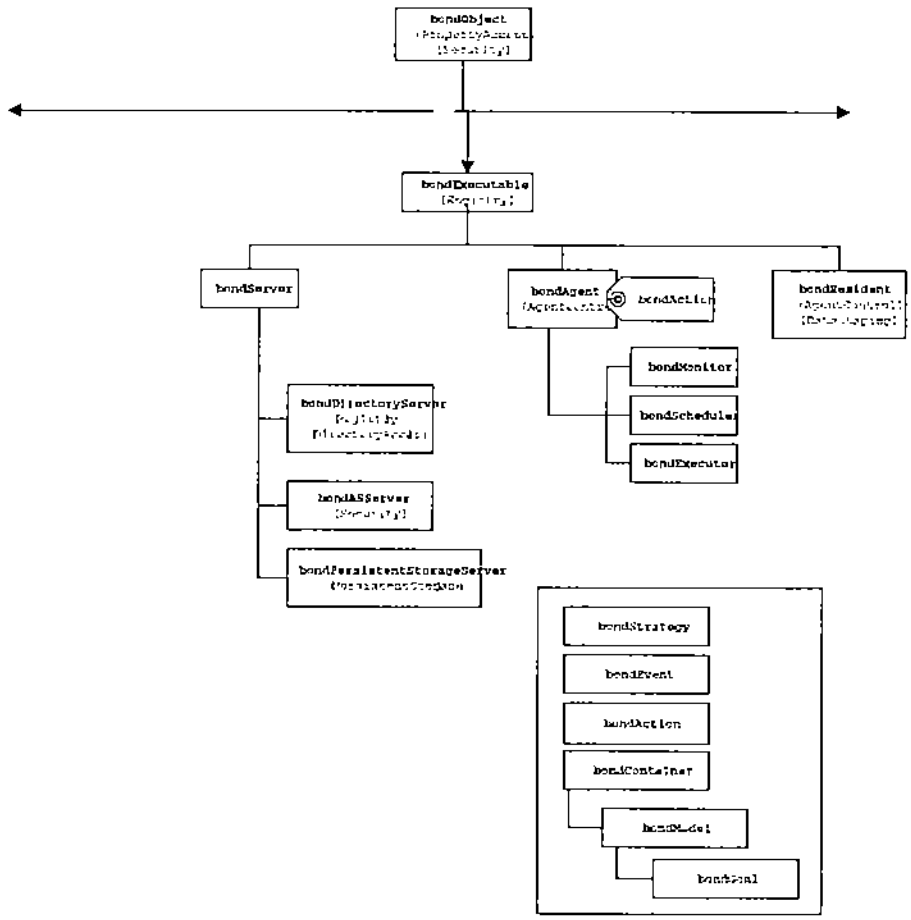

Figure 1: A segment of the Bond object hierarchy and the subprotocols understood by various classes.

messaging thread together with the local directory cause the delivery of the message to the local object. There are two types of containers, *active and passive.* Residents and workspaces are examples of containers. A *resident* is an active container consisting of several threads of control, a local directory and a set of objects. Bond objects with the exception of *light-weight objects* like messages and shadows always register with the local directory upon creation.

Figure 2 shows the structure of a resident. The messaging thread of a resident delivers messages from remote objects to local ones and sends out messages originating from local objects. Several other threads may be running as a side-effect of messages in sub-protocols understood by the resident. An *workspace* is a passive container consisting of several, possibly many objects belonging to one user. An workspace can be saved to and retrieved from persistent storage (or stored on the local host and retrieved by the resident).

3.2 Communication fabric

The system provides a communication fabric allowing objects to interact with one another. The communication fabric is based upon *active messages.* An *active message* is one delivered immediately upon receipt to its destination. A passive message is queued in a buffer for later delivery. Bond favors non-blocking communication, typically the sender continues execution after sending a request and it is interrupted upon the arrival of the reply. There are means to block, waiting for a message, e.g. `waitReply()` function [7].

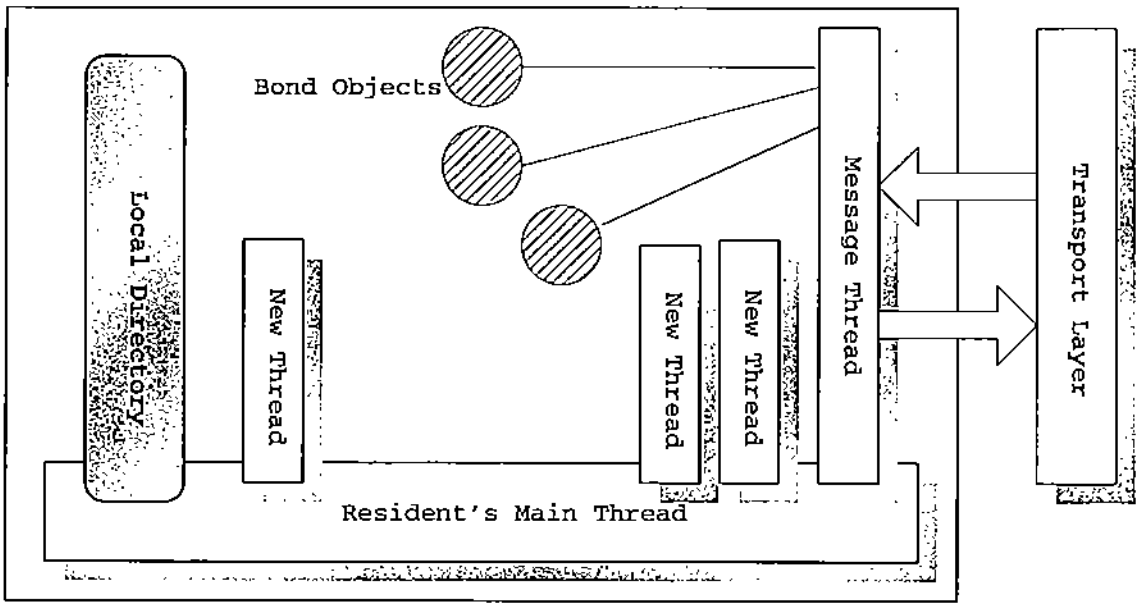

Figure 2: A Bond resident. All objects in the container, excluding shadows are registered with the local directory. The messaging thread of the resident delivers messages to local objects identified by their bondId. Messages for remote objects are sent out by the messaging thread using the address of the remote container provided by the local shadow. The main thread of a resident may start up new threads as requested by various sub-protocols. A resident could be configured as a server or have one or more agents.

A *shadow object* is a proxy for a remote object. A shadow is an abstraction designed to allow distant objects to communicate with one another as if they were co-located. A message delivered to the local shadow is guaranteed to reach the remote object. Shadow objects are the conceptual equivalents of the stubs in CORBA and RMI and can be implemented using stubs. Shadows are the only component of the Bond system that depends upon the underlying transport mechanism. Porting a Bond networking solution to a new transport middleware requires only to re-implement the bondShadow object, while all the other components can be reused even without being recompiled.

A *virtual network* is a local collection of shadow objects. This abstraction is necessary to create dynamic sets of semantically related objects. For example, a directory maintains a virtual network of local directories, a scheduler agent maintains a virtual network of execution agents.

A *subprotocol* is a closed subset of KQML messages, a specialized language needed to perform a specific task. The subset is closed because a subprotocol does not reference outside messages, the reply is a member of the same subprotocol with the question. Examples of generic Bond subprotocols are *property access* subprotocol, *agent control* subprotocol or *security* subprotocol. This abstraction is necessary to introduce a *structure in the semantic space of the messages*. An object can only understand sub-protocols implemented by its own class or inherited from its ancestors.

3.3 Probes, an aspect-oriented approach to complex object design

Bond relies on *probes* to separate various aspects of complex object design. A *probe* is an object that understands one or more specialized subprotocols and can be attach to an existing

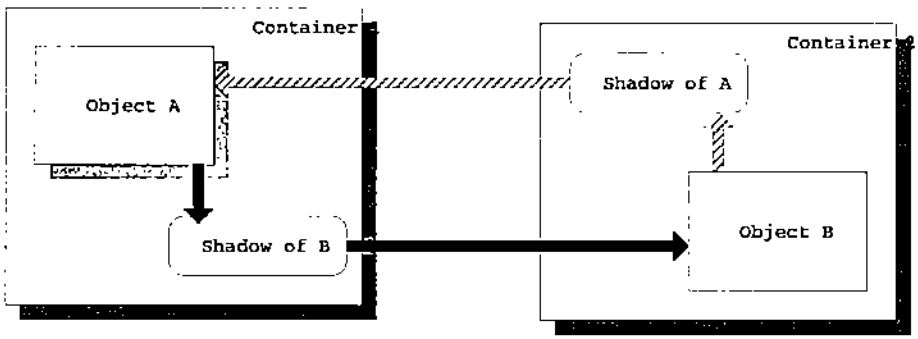

Figure 3: Communication between remote objects using their shadows. A shadow is a proxy for a remote object. To communicate with object B in container 2, object A in container 1 creates a shadow of B using the `find` function. Once the shadow of B is created, A applies the local `say()` method to send messages to it; then the shadow delivers the message to B. To send a message to A, B follows the same procedure, creates a shadow of A then sends messages for A to its shadow.

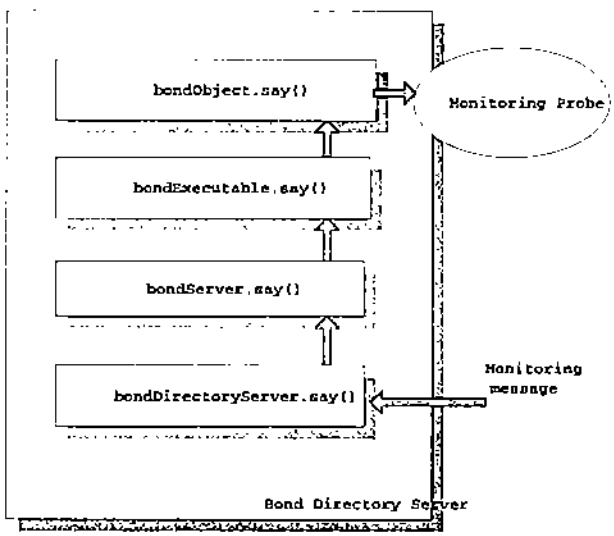

Figure 4: Probes. A `bondDirectoryServer` object does not understand the monitoring subprotocol. A monitoring message received by the `bondDirectoryServer` object message is passed to its ancestors, `bondServer`, `bondExecutable`, `bondObject`, but none of them understands the subprotocol. Then the dynamic properties of the object reveal that a monitoring probe is attached and the message is delivered to it. The monitoring probe understands the monitoring subprotocol and it is able to monitor other objects in behalf of the `bondDirectoryServer` or monitor the `bondDirectoryServer` in behalf of other objects.

object. Security and monitoring probes can be attached to an existing object. Security probes implement several security and access control models. Monitoring probes implement the Bond monitoring model. They allow objects to continuously monitor remote objects and be informed when changes of the properties of remote objects occur see Figure 4.

The system includes agent and metaobject factories and frameworks for monitoring and object security. The agent factory is capable to generate agents based upon blueprints describing the state-machine implemented by the agent,the strategies associated with every state, the agenda and the model of the world used by the agent.

Bond servers support the basic functionality necessary for collaborative activities. A Bond domain is a society of users with common interests. A system monitor starts up the servers and ensures fault-tolerance. Directory Servers allow objects to locate remote objects. The Persistent Storage Server is a repository for *user workspaces.*

Figure 5 shows the architecture of the system and its major components:

- The transport layer - ensures message delivery to remote objects. At this time Bond uses Infospheres [11], to transport messages between Bond sites.
- Core elements - provide the communication fabric, containers and core objects.
- Frameworks - provide the infrastructure to build agents and metaobjects and augment them with monitoring, and security features.
- Extensions - provide blueprints for agents, metaobjects, probes and demos,
- Servers - directory, persistent storage, the system monitor, the QoS monitor, authentication, versioning, are some of the servers that can be activated in a Bond domain.
- Interfaces - GUIs and APIs provide user access to the system.

3.4 Agent Framework

Agents are programs that autonomously pursue their own agenda and are able to cooperate with other agents to accomplish their tasks.

The agent framework in Bond allows a programmer to simplify the programming of an agent because most of the functionality of the agents are already implemented. By default, Bond agents have the possibility to be controlled remotely and to cooperate with each other.

The task of an application programmer is limited to specify the agenda, the finite state machine of the agent, and the strategies associated with each state. Bond also provides a large database of ready-made strategies, so in typical cases a Bond agent can be assembled without programming.

The segment of the Bond object hierarchy related to the implementation of the agent framework is shown in Figure 6.

Events are object that implement the `bondEvent` interface, however all the events currently used in Bond agents are KQML messages, represented by the `bondKQML` object. The programmer may define new types of events.

The finite state machine is implemented by the `bondFiniteStateMachine`, `bondFSMState` and `bondFSMTransition` classes. These classes are general enough that they don't have to be overwritten for any agent covered by this formalism. A `bondFiniteStateMachine` can be constructed dynamically and saved or restored from the persistent storage.

Strategies are defined by any object which implement the `bondStrategy` interface. The model of the world is represented by the `bondModel` object. The information is represented in form of objects attached as dynamic properties of the model object. This approach allows

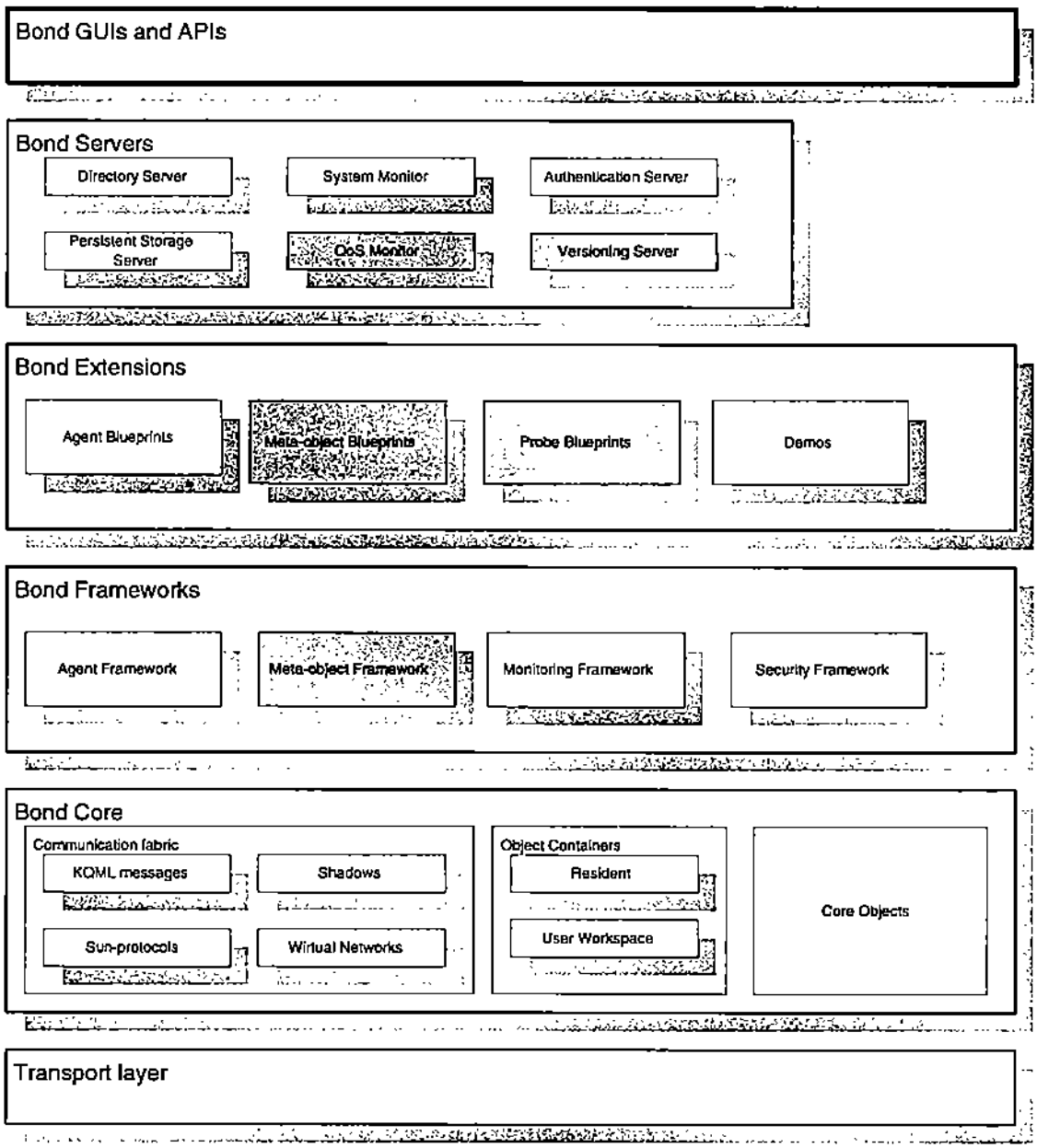

Figure 5: The architecture of the Bond middleware. The transport layer is provided by the info.net class of the Infospheres. Solid boxes are components not yet implemented. Bond servers are necessary to guarantee services in a Bond domain. Bond objects may communicate with each other even without the assistance of servers.

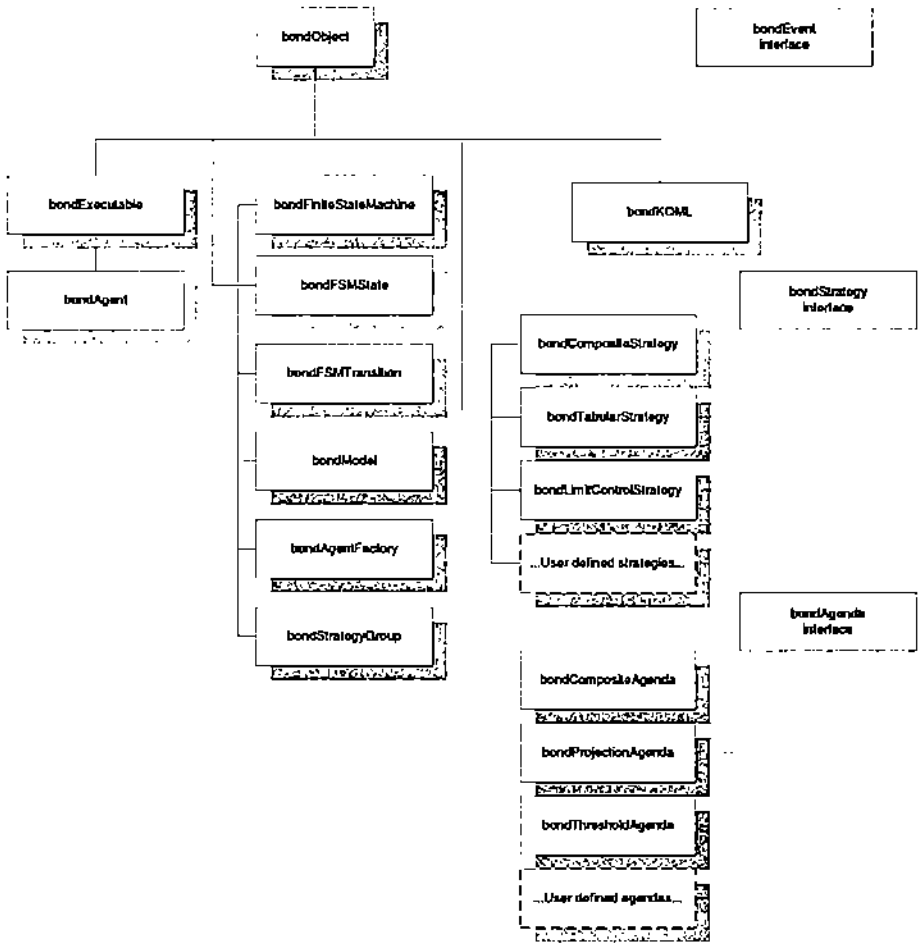

Figure 6: The segment of the Bond object hierarchy directly involved in creating agents

the programmer complete freedom to use any knowledge representation approach he or she wants. We found it useful to impose certain *conventions* on the representation which allows various strategies to work together. A programmer may or may not follow these conventions.

The agenda is an object implementing a `bondAgenda` interface. The `float distance(bondModel)` function should return the distance from the desired state of the world, while the `satisfiedBy()` function returns a boolean value showing if the agenda is satisfied or not.

One particular way of implementing the agenda is when the desired region is a projection of the Model, i.e. some values of the model should match specific values, while others can have arbitrary values. This is implemented in the `bondProjectionAgenda` object. Another possible agenda is when the desired region is an n-dimensional rectangle: we provide specific limits for the values of certain properties. This agenda is implemented by the `bondRectangleAgenda` object.

The `bondCompositeAgenda` object implements an *agenda calculus* supporting boolean expressions on ready made agendas. The ready-made agendas cover most of the useful agendas one would think of using within the agent framework. These agendas can be created dynamically, without the need to write Java code.

Actions in the Bond system are object that implements the `bondAction` interface. Examples of defined actions in the object hierarchy are the `bondMethodCallAction` that allows a particular method of an object to be designated as an action and `bondAgent` which specifies

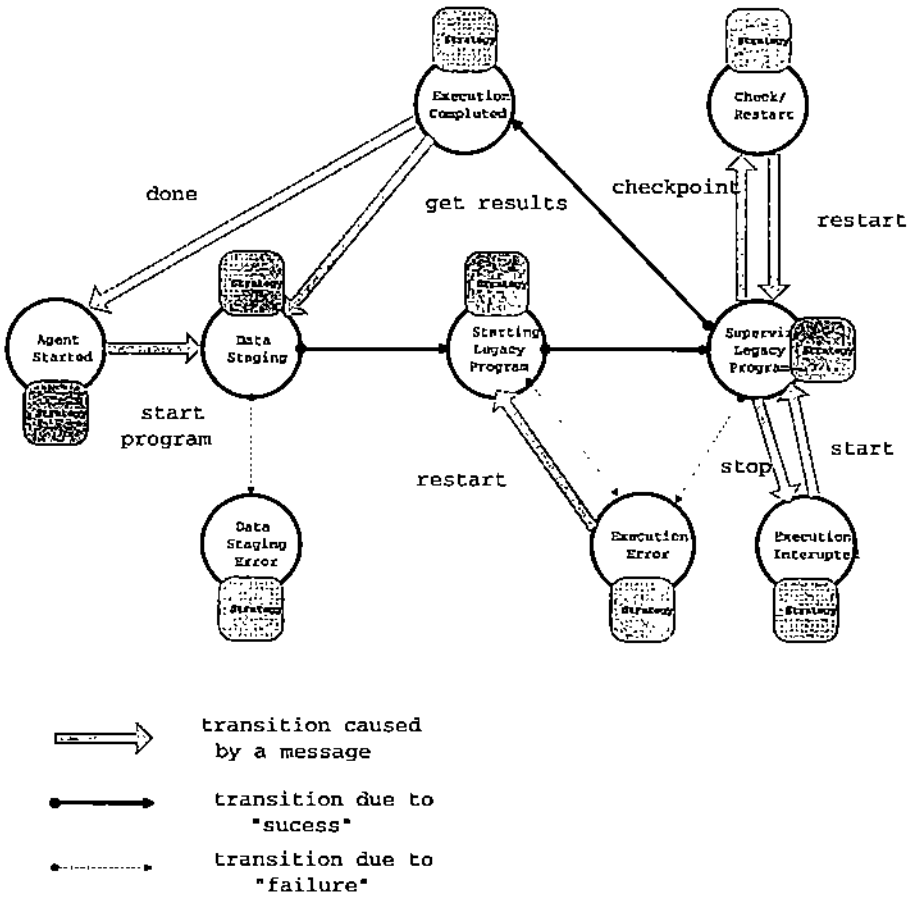

Figure 7: The state transition diagram of the Data Staging and Remote Execution Agent, `bondDSexec`. The functions of the agent are: perform data staging, start-up a legacy program on a remote host, and supervise its execution. State transitions are caused by: (a) external messages, (b) the success/failure to attain the goal of the strategy in a given state.

that starting a new agent can be an action. The user can also define custom actions.

Strategies are object implementing the `bondStrategy` interface, which defines the `nextAction()` function. Strategies can be created using their constructors, created as part of strategy groups, or saved and loaded from persistent storage.

Figure 7 illustrates the states and the transitions of a data staging and remote execution agent. The strategy associated with every state is a sequence of actions, often the result of the execution of a program, or a script. For example the strategy for the "Data staging state" may consist of a script invoking the `ftp` file transfer protocol, the strategy for "Starting legacy program" state invokes a system call to start up a process running the program. The model of this agent consists of variables to store information needed by various strategies, e.g. the path to the executable for the legacy application, the completion code of the strategy for each state, and so on.

3.5 Monitoring Framework

Objects in Bond are cooperative. They provide and request services one another. They are inherently dynamic. Agent objects appear on demand and disappear. Permanent servers change their state. It is important for objects to know the state of other objects and their

changes.

We design a *subscription-based monitoring model* [10]. The following terms are used to describe the model:

- *Property:* an entity consisting of name and its value.
- *Event:* the changes of properties.
- *Subject:* an object whose properties are of interest to other objects and it is willing to provide them upon request.
- *Monitor:* an objects that collects the properties of other objects, analyzes them, and takes actions based upon the result of this analysis.

In this model a monitor subscribes to the properties of interest of a subject. The subject notifies the subscribers of the events. The notifications are sent according to one of the notification models:

- *Time-triggered:* the subject notifies the changes periodically using an agreed-upon time interval.
- *Change-triggered:* any changes in the subscribed properties triggers notification.
- *Hybrid:* it works like change-triggered when changes do occur as time-triggered if there are no changes.

An implicit assumption of our model is that monitors and subjects are aware of the effects of communication latency and processing delays. If we denote by δ the total elapsed time from the instance of change, till the action triggered by the change is completed, the objects involved in monitoring maintain their relevant properties stable for at least Δ time with $\Delta > \delta$. We call this Δ-*monitoring*. Each pair (monitor, subject) needs to agree upon the value of Δ and considers only actions compatible with or irrelevant with this value.

A *Bond Monitoring Framework* implements this model. It consists of monitoring probe, monitoring interface, and monitoring policy. The *monitoring probe* is an object implementing the subscription/notification communication between a monitor and a subject. By attaching the monitoring probe, objects can perform that communication. The *monitoring interface* defines a list of methods which the objects with the monitoring probe should implement:

- `handleNotification()` is invoked whenever the notifications arrive.
- `handleMissingNotification()` is invoked when the expected notification is missing.
- `handleMissingProperty()` is invoked when the subject does not include the subscribed property.
- `handleError()` is invoked in the case of general errors.

The *monitoring policy* is an object containing a subject address, property names, and notification model. The monitor starts monitoring by creating the monitoring policy and handing it over to its monitoring probe.

Figure 8 shows a monitoring example. Suppose that there are two objects, A and B. A is going to monitor B. Both A and B create monitoring probes and implement monitoring interface. A constructs a monitoring policy and gives it to its monitoring probe, which in turn subscribes to B. The subscription request is delivered along the hierarchy to the monitoring probe of B, which sends back the notifications. The monitoring probe of A invokes `handleNotification()` whenever the notifications arrive.

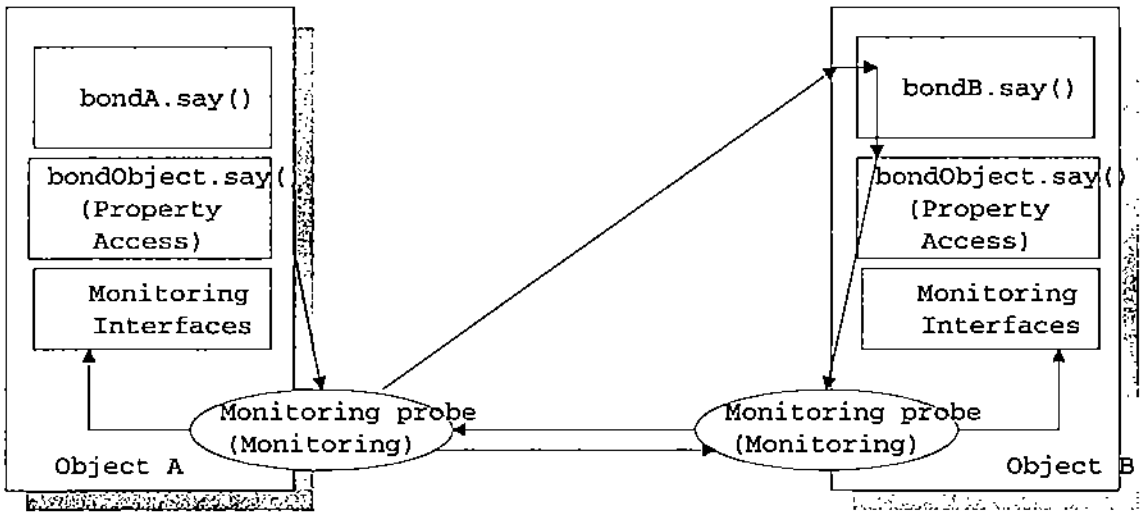

Figure 8: Two objects, A and B, involved in a monitoring relationship, A monitors B. Both objects create the monitoring probe and implement the monitoring interface. Once the monitoring is started, all the monitoring messages are exchanged between the monitoring probes.

3.6 Security Framework

Applications of network computing have vastly different security requirements and the trade-off between security and efficiency are application specific.

The first design principle of Bond security is to provide a framework for security, not implementations based upon a specific security model. By providing such a general framework, Bond leaves the decision of choosing security mechanism, such as the format of credentials, the authentication policy, the access control policy, etc. to the system developers. Such a framework is implemented through an extensible core bond object called bondSecurityContext and a set of well-defined security interfaces.

The second design principle is that security is an aspect of a complex object design. A Bond object can become a secure object by declaring or dynamically adding the property bondSecurityContext. The bondSecurityContext sets up a secure boundary for the object and intercepts all messages sent to/from the object.

The third design principle is to support multiple authentication and access control models. This goal is achieved by defining a common interface for different security functions, like credential, authentication and access control. A bondSecurityContext contains security-related objects supporting security functions. Each of these security-related object implements one of the security interfaces defined by Bond.

The bondSecurityContext sets up a secure boundary for the object. It has two methods: `incomingMessageProcess()` and `outgoingMessageProcess()`, called on the incoming and outgoing messages.

The bondSecurityContext is implemented as a preemptive probe to intercept all incoming and outgoing messages. The bondSecurityContext captures all the messages sent to its Bond object, and the `incomingMessageProcess()` method is invoked to enforce security measures without the involvement of the object.

The bondSecurityContext captures all message sent out by the object. It achieves this by intercepting the message when it was moved from the master object to the local message thread for delivering. The `outgoingMessageProcess()` method will be called to do any programmer defined security process on the outgoing message.

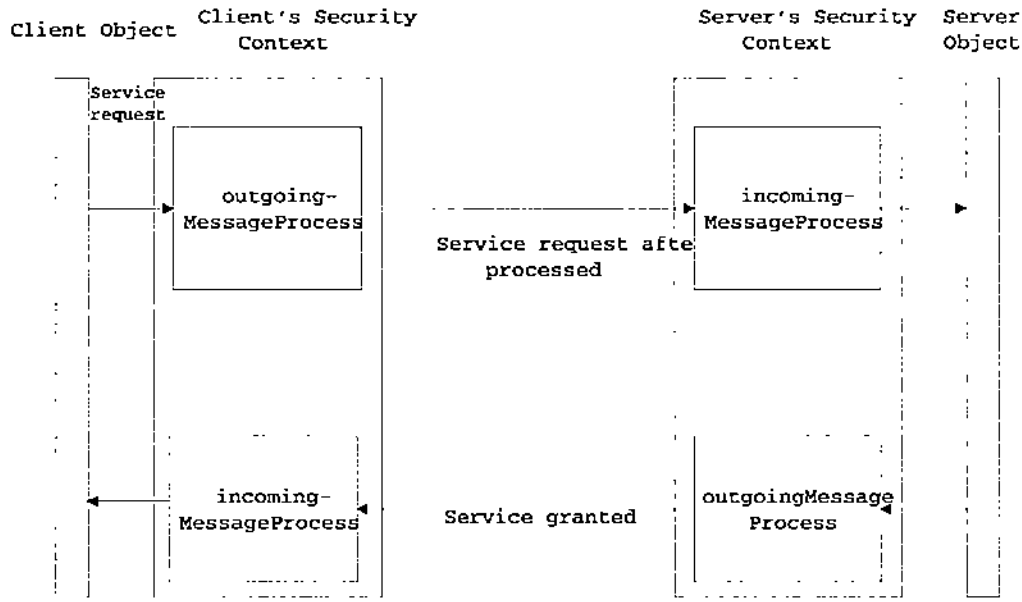

Figure 9: Application of the Bond security model to client-server operations. The client object first sends a service request to the server object. This message is intercepted by its security context, it is then forwarded to the `outgoingMessageProcess()` and sent out. All incoming are intercepted by the server's security context. The `incomingMessageProcess()` enforces authentication and access control before delivering the message to the server object. Finally, the server object grants the service.

A BondSecurityContext contains security-related objects that implement various security functions. Each security-related object implements one of the security interface defined by Bond. Existing security interfaces includes:

- `bondCredentialInterface` - This interface defines the method to access the credential possessed by the current bondSecurityContext. The implementation of a credential can be user/password, access ticket or public/private key pair.
- `bondAuthenticatorInterface` - This interface defines the method used to enforce authentication for each received message. Programmer or administrator can define different authentication mechanisms, for example, username/password based authentication, ticket based authentication, or certificate based authentication. The only restriction is adhering to this interface. The only method in this interface is `authenticateClient()`, it will return an authenticated user identifier.
- `bondAccessControlInterface` - This interface defines the method to enforce access control for each received message. Programmer or administrator can define different access control mechanisms, for example, IP address based firewall, username based access control list, or others. The methods provided by this security interface are `initACL()` and `checkRight()`. The first method is used to initialize the access control object; and the second one is used to check access rights of an authenticated user identifier.

4 Applications - Middleware for a Virtual Laboratory

We propose to create a Virtual Laboratory middleware to support experiments and computer modeling activities in a scientific laboratory. First, we present a model of the activity the middleware is expected to support. In our model a group of individuals collaborate to solve a scientific problem. They carry out experiments and collect data produced by sensors connected to the experimental setup. Often the volume of experimental data is very large due to the nature of the physical phenomena and to advances in sensor technology. The experimentalist needs to keep a detailed log of the experimental conditions and the precise setup used for each set of data. The next step is to extract the relevant information from the experimental data. Seldom this process is straightforward; often it requires a significant computational effort. Once the experimental data are distilled, they are plugged into a computational model. This model itself is typically very complex, depends upon a large number of parameters and computations have to be carried out repeatedly with different sets of parameters until the results are deemed to be acceptable. Some of the components of the model are inherited from other groups and there is little understanding of the inner working of the legacy codes written long time ago but trusted due to repeated use. At the same time, there are new components of model that may change on a daily basis.

The entire process of discovery typically takes months if not years, it involves complex interactions between humans, and between humans and computers, it requires backtracking, keeping detailed information about each step, keeping intermediate data. The members of a group are often geographically distributed and have to use a diversity of computers.

Based upon this model we concluded that the critical elements of the Virtual Laboratory are annotation of network resources, an infrastructure for communicating objects, knowledge processing, monitoring of concurrent activities, and workflow management.

Some of the functions desirable in such an environment and supported by Bond will be:

- System-level resource management,
- User-level resource management,
- Annotation of network resources,
- Management of communication channels,
- Support for collaborative work, ability to share resources,
- Access to computational servers,
- Remote start-up and execution of single programs,
- Execution of workflows,
- Persistent storage,
- Data migration,
- Scheduling,
- Monitoring,
- Access control and security related functions,
- Accounting.

5 Summary

We believe that several metaphors from structural biology are useful for designing distributed object systems. Metaobjects, persistent objects providing information about network resources, provide a metaphor for the genetic material. Agents use introspection and reflections to emulate lock and key mechanisms necessary to bind together various components. The aspect-oriented design supports aggregation of components developed separately and emulates binding of proteins. Security probes build a defense perimeter surrounding an object emulating the function of antibodies. The monitoring probes can be used to construct the equivalent of a central nervous system.

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 planned for early November 1998. Additional information about the system is available at http://bond.cs.purdue.edu.

6 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] T. Kohonen *"Self-Organization and Associative Memory"*, Springer-Verlag, Berlin 1984.

[5] 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

[6] J. R. Koza, *"Genetic Programming: On the Programming of Computers by Means of Natural Selection"*, MIT Press 1992.

[7] L. Bölöni. *"Bond Objects - A White Paper."* Department of Computer Sciences, Purdue University CSD-TR #98-002.

[8] L. Bölöni and D.C. Marinescu. *"Robust Scheduling of Metaprograms"*, September 1998, (submitted).

[9] 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).

[10] K. K. Jun, L. Bölöni, Ruibing Hao, and D.C. Marinescu. *"A Subscription-based Monitoring Model for Distributed Systems"*, October 1998, (submitted).

[11] K. Mani Chandy. *"Caltech Infospheres Project Overview: Information Infrastructures for Task Forces."*

[12] M.C. Cornea-Hasegan and D.C. Marinescu. *"SBL, The Structural Biology Language"*, Department of Computer Sciences, Purdue University CSD-TR #94-008.

[13] 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

[14] 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.

[15] T. Finin, et al. *"Specification of the KQML Agent-Communication Language"* DARPA Knowledge Sharing Initiative draft, June 1993.

[16] 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).

[17] 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.

[18] 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).

[19] R. Orfali and D. Harkley. *"Client/Server Programming with Java and Corba"* Willey, 1997.

[20] R. J. Stroud. *"Transparency and Reflection in Distributed Systems"* ACM Operating Systems vol. 22, no. 2 pp. 99-103, 1993.