1998

Reflections on Metacomputing, The Bond View

Dan C. Marinescu

Ladislau Bölöni

Report Number:
98-006

Reflections on Metacomputing, the Bond View

Dan C. Marinescu and Ladislau Bölöni
Computer Sciences Department
Purdue University
West Lafayette, IN 47907

CSD-TR 98-006

Reflections on metacomputing, the Bond view

Dan C. Marinescu and Ladislau Bölöni
Email: dcm, boloni@cs.purdue.edu
Computer Sciences Department
Purdue University
West Lafayette, In, 47907, USA

Abstract

In this paper we present the basic ideas of a metacomponent based architecture for network computing on a grid consisting of autonomous nodes. 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. Then we present an infrastructure for metacomputing built out of primitive elements able to discover each other's properties using a language created by the Knowledge Sharing Effort.

Contents:

1. Introduction

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 and allowing them to run on computing grids consisting of autonomous systems. 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. It turns out the computing landscape is considerably more diverse than most us think. While we are resigned to multiple versions of Microsoft's Office programs or Adobe's Framemaker most computer users and even software designers are unaware that there are more than 50 versions of the Pentium Pro processor, each one with its own minute differences from the others, [1], [26].

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 and a respectable source of funding for the more able computer scientists. 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 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? Who would like to live in a world populated solely by NT systems running on Pentium processors?

It is our belief that entering the information age requires more affordable and more accessible computers. It is truly inconceivable to expect a growing segment of scientists, engineers, and other people who need computers for complex tasks, face the problems of hardware heterogeneity and software diversity as is the case today. We advocate a pragmatic approach of building complex systems, one should attempt to find a glue to bond together existing hardware and legacy software the components developed independently, with components using new technologies. The same infrastructure should be used in all forms of computer life from scientific software to electronic commerce. To accommodate the inherent heterogeneity of any complex system we propose to define information objects that describe in the most minute detail the actual software, hardware, and data objects we want to compose. These objects should be classified into disjoint classes, with objects being able to inherit properties from their ancestors and acquire new properties.

There is a very long list of evolutionary attempts to design complex computing systems, including Andrews, Locus and NSF file systems, Sun XDR, the Washington HCS, Chorus, Linda, PVM, Maud, Garg. There are success stories, several large and dependable systems were built, but at very high costs. Some attempts were fundamentally flawed and could not scale due to technical fallacies, e.g. insisted on mechanisms to control resources centrally and maintain an accurate state of the system, regardless of the scope of the distributed system. Others failed because they were ahead of their time, the technology of networks, processors, storage devices, sensors, as well as the programming languages, operating systems, communication protocols, were not sufficient to support their ideas. But above all, the society was not prepared for wide spread use of computers so that such systems were targeted primarily for the elite few who had a high threshold of pain and could afford to dedicate considerable time to computing activities.

During the past few years several projects in the areas of metacomputing have achieved a level of recognition. The list includes the Legion, Globus, and the Friends projects surveyed below. They all share the vision of seamless metacomputing and propose to achieve high performance via parallelism using loosely coupled distributed systems [7], [8], [11], [13], [14], [15], [17], [25]. The Legion project started in 1995 at the University of Virginia envisions a system capable to schedule transparently applications on processors, manage data transfer and coercion and provide communication and synchronization [15]. Its design objectives are: site autonomy, an extensible core, a scalable architecture, a seamless computing environment, high performance via parallelism, a single global name space, security for users and resource providers, management and exploitation of resource heterogeneity, multi-language support and inter-operability, and fault tolerance, [17]. The actual implementation of the Legion system was expected to start in late 1997.

The Globus project is "a multi-institutional research effort that seeks to enable the construction of

computational grids providing pervasive, dependable, and consistent access to high performance computational resources, despite geographical distribution of both resources and users" [14]. In Globus a low level toolkit provides communication, authentication, and data access. The goal is to build an Wide Area Resource Environment, AWARE, an integrated set of high level services that enable applications to adapt to heterogeneous, dynamic changing environments [13]. Components of the Globus project have been demonstrated as early as 1997 and some of them are used by different research groups.

The Friends system is a metalevel architecture providing libraries of metaobjects for fault-tolerance, secure communication and group-based distributed applications [11]. Metaobjects can be used transparently and be composed according to the needs of an application. The system attempts to support composability of mechanisms dealing with different fault classes as needed by an application. Metaobject protocols give the user the ability to adjust the language implementation to suit their particular needs. Reflections expose the language implementation at a higher level of abstraction [25].

The Infospheres project at Caltech [7], [8] has the goal to develop network centric technologies and design information infrastructures that support virtual organizations. The Infospheres prototype has been distributed to a number of organizations. An early version of Bond 2.0 uses the mailboxes and the transport mechanisms provided by the Infospheres.

Now we discuss some of the considerations leading to the development of the Bond system. Bond is an infrastructure project in the area of scientific computing, it is not an attempt to design yet another distributed system or to link together a number of components into a rigid structure for a specific domain. 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 [10], [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.

Our first step to automate the complex data collection and analysis process in computational biology was to define a scripting language, SBL, the Structural Biology Language, suitable for executing iterative computations on a parallel computer [9]. SBL supports mechanisms to determine if convergence criteria have been met and if so execute a different sequence of programs, or carry out more iterations involving the same group of parallel programs. SBL includes a ckeckpointing facility as well as a log. This approach has obvious limitations that became evident once we attempted to execute parallel programs using the MPI communication library, on a cluster of workstations. Another problems we faced, were the constant need to modify the programs, and the need to ensure compatibility between different program versions and data. Different members of the group used different versions of programs and merging partial results was a difficult task. After weeks or even months of calculations, it was often impossible to determine the "genetic" information, to establish what parameters of the model were used in computations carried out in the past, unless one kept a very detailed diary of each step.

The next step was to define descriptors of all objects, programs and data and to design a remote execution shell [21]. The *Bond shell* uses these descriptors to support *user level management of resources* distributed over a computing grid and to ensure the compatibility between different program and data formats. Each user has a database of program, data, and hardware descriptors, used to start the execution of a program on a remote host. The Bond shell is written in Tcl/Tk and Expect. Though well suited for the needs of a single individual, the Bond shell does not support group activities.

Bond 1.0 (http://bond.cs.purdue.edu) inherited the descriptors as well as the mechanisms for remote execution, from the Bond shell, but made further steps towards supporting group activities. The new system is built around a *Bond server*, an HTTP server, which provides persistent storage for all descriptors, or Bond objects in our terminology. The system is accessed from a Web browser and provides a uniform and location independent execution environment. Once a user with a Bond account connects to the Bond server, a collection of applets is downloaded and the user can build a workspace, the collection of Bond objects visible during a session and execute flow graphs remotely. The *sandbox* security model used to support applets in Java is very restrictive it does not allow local file access, it only allows an applet to communicate with the host it has been loaded from. Therefore we created an *applet server* running on the same host with the Bond server and all operations required to manipulate Bond objects were carried out at the site when the two servers were running. Bond objects could be shared among the members of the group.

A number of services for *seamless remote execution* are provided in Bond 1.0, including data migration, password, and scheduler agents. The *scheduling agent* [22] implements a dataflow execution paradigm and maps computations to the platforms of a computing grid with autonomous nodes.The *data migration agent* replicates data from the producer site to the consumer site on the grid. The knowledge base of the *password agent* identifies groups of machines which share the same password for a given user. Bond is a facilitator not a distributed system, the basic philosophy of the system is to defer security and resource allocation mechanisms to the individual nodes of the grid. Bond initiates an action, say **ftp** or **rexec** in behalf of a user, on a node of the computing grid using information stored in its internal knowledge bases, but does not attempt to store confidential information or interfere in any form with the resources management of individual nodes. When the password or even the user id are necessary, the system requests the information from the user. By August 1997, the Bond 1.0 system consisted of about 80,000 lines of Java 1.0, Tcl/Expect, and Clips code.

In this paper we discuss ideas and concepts pertinent to the new version of the system, Bond 2.0. In Section 2 discusses biological analogies introduce metaobjects, in Section 3 we present an infrastructure for communicating objects, and in Section 4 we present agents and services to support a metacomputing environment and introduce models for Bond domains.

2. Biological systems, composition, introspection, and metacomponents

A number of biological analogies have found their way into computer science. Neural networks provide an alternative to von Neumann architecture, genetic algorithms are used to solve optimization problems, mutation analysis was proposed for software engineering. The question we are concerned with in this section 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 sys-

tem. First, we overview some basic properties of biological systems relevant to our discussion.

Nature uses composition to build extremely complex structures [2]. 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* [2]. 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 defence 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 cary 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. Spheric 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 relay 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* consisting of a *physical/network object* (be it a program, data or hardware) and an *information object* associated with it. We use here the term "object" rather loosely, but later on it will become clear that the metacomputing architecture we envision is intimately tied to object oriented concepts, very much like Corba is. We talk about network objects to acknowledge that we are concerned with a distributed environment where programs and data are distributed on a computing grid consisting of autonomous platforms interconnected by high speed networks.

The set of properties embedded into an information object provide a "lock and key" mechanism similar with the one used by proteins to recognize one another. In an *workspace* populated with metaobjects, one can envision mechanisms to link them together to form a *metaprogram*, a new metaobject, capable to cary out a well defined computational task. Example: to link a data metaobject to a software metacomponent we need to search our 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 metacomponent 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 information objects describing the elements discussed above is a non trivial task.

We expect the information object to contain a description of all relevant properties of a network object including "genetic information". This information must be in a form suitable for machine processing by an *intelligent agent*. For example the information object of a software metacomponent 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 intelligent 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 information objects seem daunting? We should expect it according to the biological analogy discussed above. The task of abstracting the properties of network objects is monumental, one can only succeed if the network objects, 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 metacomponent. 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, which 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 intelligent 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 information object only *stable properties* of the network object. For example the amount of main memory available 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 information objects is a difficult, but not an impossible task we have to ask ourselves if the two elements of a metacomponent, the information object and the network object need to be tightly or loosely coupled with each other. The tightly couple approach would require that the two components are kept together to ensures consistency. There are fundamental flaws with the tightly coupled metacomponent 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 Word would need to store a huge amount of information including its genetic component that Microsoft is unlikely to reveal willingly.

By virtue of the arguments discussed above a metaobject is a distributed object consisting of one or more network objects and a distributed information object possibly with a hierarchical structure. Different properties of the 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 information object 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, the components of the information object would be shared among all the users of the object. An important side effect of this approach is that software distribution and maintenance would be greatly simplified. The information object of a software package will point to a database of sites that have downloaded the software and every time a new release becomes available all of them will be notified.

The price to pay for the distributed metacomponent approach is that inconsistency between the information objects and the network objects of a metacomponent are occasionally unavoidable. We argue later on that in a network rich environment catastrophic consequences of such inconsistencies are unlikely to occur.

In summary, we propose is to create metaobjects consisting of physical objects and information objects describing their properties. 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 are primarily concerned with the task of building an infrastructure supporting the design principles presented in this section.

3. An architecture for communicating objects.

The information objects described so far form a *distributed knowledge base*. They are lifeless entities, regardless how complex their structure may be. They can only be brought to life by a system that allows them to communicate with one another, defines specific actions that occur as side effects of a message and entities capable to carry out actions and to coordinates the activities necessary to carry out a computational task. In this section we discuss the architecture, the language and the transport mechanisms of the Bond system.

A critical aspect of the Bond system is how objects communicate with one another, how they learn about each other's existence, how they discover their properties. Bond is a Message Oriented Middleware, MOM, whose main ingredients are a lightweight mechanism to establish long term relationships amongst objects and a "universal" language which enables objects to communicate attitudes about information, being indifferent to the format of the information itself. *Shadow objects* provide a high level abstraction for a unidirectional communication channel linking two objects together. When an object needs to communicate it involves the directory service and, when the object is found, a shadow of the remote object is created and the connection is established. This is a first significant difference between Bond and the popular ORB, Object Request Broker of OMG. Other differences are summarized in Table 3 at the end of this section.

The second important decision was to use KQML, Knowledge Sharing and Manipulation Language, as the native language of Bond. KQML is a product of the Knowledge Sharing Effort supported by DARPA, NSF, and AFOSR, for organization and coordination, [12], [16]. Intended as an inter-agent communication language by its designers, KQML is used in Bond as an inter-object communication language. In Bond all objects can receive messages regardless of their state. KQML is indifferent to the format of the information itself and KQML messages may contain information in the so-called "context language". The meaning of a KQML message is defined in terms of the constrains of the message sender and allows the message receiver to choose a course of action compatible with other aspects of its function. The richness of KQML is most appealing to us, it allows us to define subprotocols, sequences of messages needed to carry out a desired function.

A Bond object allows dynamic definition of new properties. Properties defined at run time are stored in hashtables attached to the objects, while the properties defined at compile time are regular Java fields. Performance considerations force us to define all known properties at compile time, reserving the dynamic definition for the unexpected cases. The remote access to the properties of an object requires a transparent access to both types of properties as well as "access by name". The problem was solved using the reflection mechanism available in Java 1.1. The local "get" and "set" methods and the remote property access subprotocol provide access to the regular Java fields and the dynamic properties in a uniform way, while the compiled code can still access the fields at the full speed of a compiled language.

3.1 Bond objects.

The entities manipulated by the Bond system are called Bond objects [4]. A Bond object has a name, a unique internal id and a set of properties. New properties can be added, properties can be deleted, or the value of a property may be changed. All Bond objects (a) are persistent, and (b) communicate using the KQML language. A Bond object can be either active, loaded into the main memory or passive, stored on persistent storage. Figure 1 shows a class hierarchy of Bond objects. Internal Bond objects are currently implemented as Java classes. Bond executables as well as the communication objects provide the glue which bonds together metacomponents.

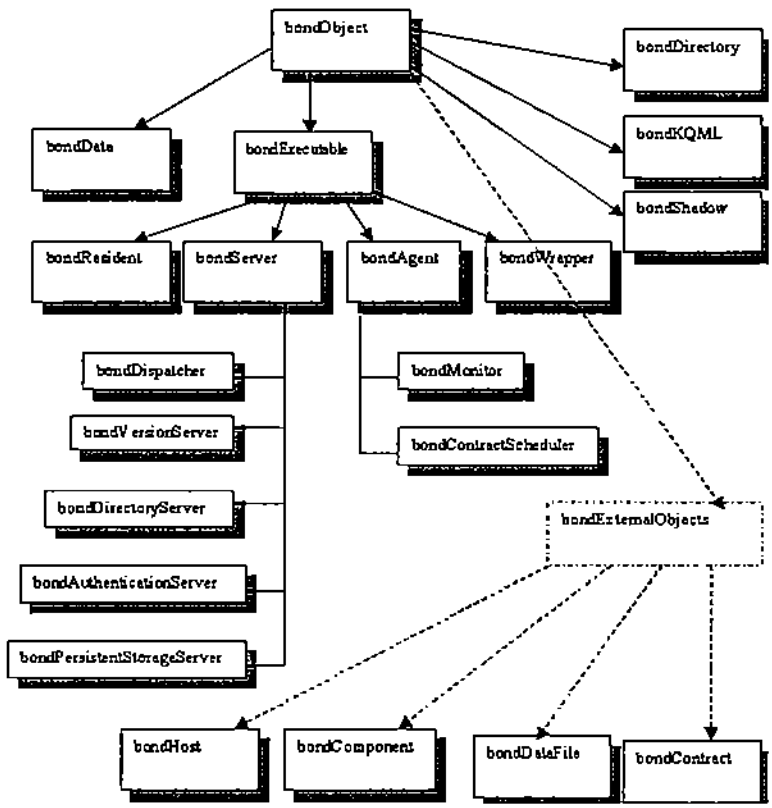

Figure 1. Bond class hierarchy

There are several classes of Bond executables, *residents*, the main thread of control of any Bond program, *agents*, autonomous programs that may have different and even conflicting agendas,

servers, programs that performs a specific function, or *wrappers*, programs used to run legacy applications. Directories, shadows and KQML messages provide communication support. Other objects encapsulate information about an external object, be it a hardware platform, a software component, a data file, or a contract and are called external Bond objects. An external Bond object contains either an URI, the Uniform Resource Identifier of a file, or the IP address of a platform.

Bond Objects are *network objects* distributed over a computing grid, they communicate with each other, can be instantiated and run remotely. The basic networking abstraction in the Bond system are the *shadows*. To access a remote object, a program creates a local shadow of the object as shown in Figure 2. The shadow of an object is created as a side effect of a directory search for the object. Once the shadow is created all messages sent to the shadow are forwarded to the original object. The message patterns for the directory search are discussed in Section 3.3.

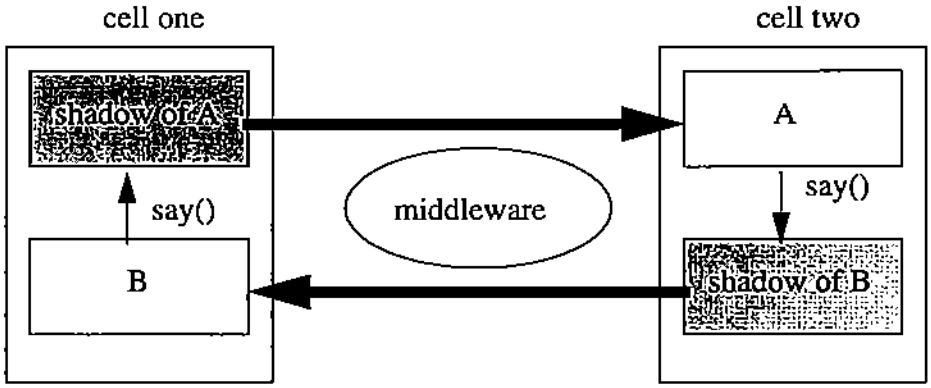

Figure 2. To communicate with object A, at cell two, object B, at cell one, creates a shadow of A. To establish a bidirectional channel, A creates a shadow of B. Then B sends messages to the shadow of A using the local method say() of cell one.

The shadow abstraction completely isolates the programmer from the transport layer. Object networking reduces to calling functions on shadows which are regular local objects. Only the implementation of the shadows and the messaging thread discussed in the next section are dependent on the underlying middleware or communication library. The Bond system can be ported to a new middleware by re-implementing the shadow mechanism on the new middleware, while the rest of the programs does not even need to be re-compiled. Systems employing more than one middleware are also possible.

The shadow abstraction in Bond is analog to the stubs used in Corba or Java RMI for remote procedure call. All three abstractions are local representations, "proxies" for remote objects. The main difference is that Bond shadows are "universal", a shadow fits any type of Bond object while Corba stubs have to match the remote object, regardless of their creation methods, static or loaded dynamically from an repository. Bond shadows only pass strings to the remote object, it is the task of the remote object to interpret such a message in a meaningful way. In this context "sorry, i don't understand' is a meaningful response of a remote object to a message while the most likely answer of a contemporary system would be a coredump. This communication is pos-

sible because every pair of Bond objects have a common ground for communication, the KQML language.

Local copies of remote objects can be created by *realizing* the shadow of the object. To create a remote object a program instantiates a shadow of the object, specifies its properties and creates it using a *factory* object at the remote site. One can observe that in this case the shadow precedes the object. The most important examples are the remote creation of database items and remote start of executables.

3.2 The Architecture

The basic element of Bond architecture is a *cell,* a collection of *atoms*, or Bond objects, **b**, coexisting on a given host. A cell consists of a *local directory*, **dir**, a *resident* **r**, the main thread of control of the cell, other threads spawn by r, including a *messaging thread*, **mt** and two *mailboxes*, **in** box and an **out** box, as shown in Figure 3.

The messaging thread pools its **in** box to determine if there are messages to be delivered, then invokes the message parser to determine the destination and contents of a message, and finally it invokes the corresponding method on the target object. Outgoing messages from all objects in the cell are placed in the **out** box by the messaging thread. Cells communicate with one another using a transport mechanism that removes messages from the **out** box of a source cell and places them in the **in** box of the destination cell.

There are different types of cells in the system. The function of a cell is determined by the configuration of the additional threads spawn by the resident. For example, the *system monitoring* cell is responsible to start *server* cells supporting the basic functions of the system, an *external* cell is created once a user connects to the system.

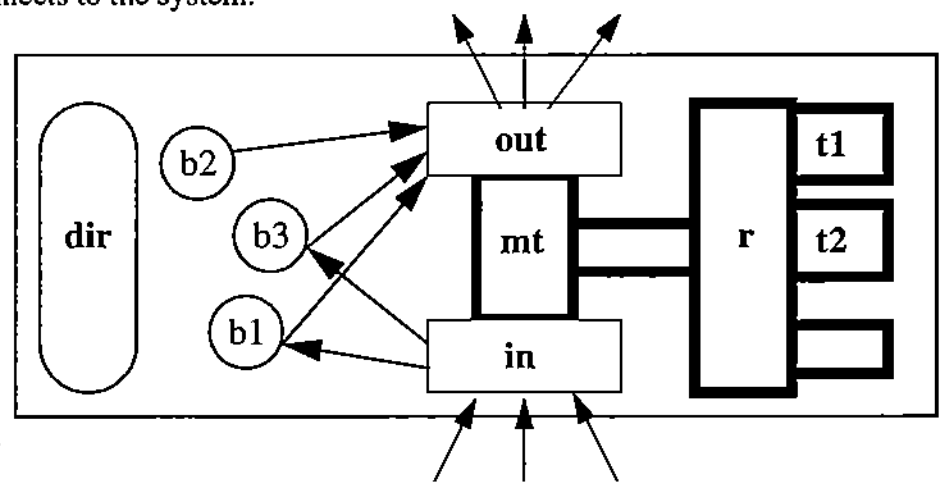

Figure 3. Bond cell

Bond cells are organized into *domains* or Intranets. The cells in a domain connect to one another according to service patterns, as shown in Figure 4. An external cell, **E**, first connects to the *dispatching service cell*, **D**, to gain access to the system, then accesses the *persistent storage and global directory service*, **P**, and other services distributed throughout the system, e.g. *brokerage*, **B**, *authentication*, **A**, *software distribution*, **S**. Control and supervisory services, are ensured by the *system monitor*, **M**, and the *quality of service monitor*, **Q**. Each service may be provided by the

one or more service cells.

New cells are created as needed. For example a contract coordinator cell, ***C***, is created in response to a request from an external cell to execute a *contract*. A contract is a metacomponent describing a complex activity that can be carried out on a computer grid, it is a metaprogram in execution, the same way a process is created to run a program. A contract consists of sub-contracts, atomic actions performed by *transient* servers, T. A transient server is a cell created by a contract coordinator consisting of a *wrapper* and a *legacy application*. The subject of contracts is discussed in depth in the next section.

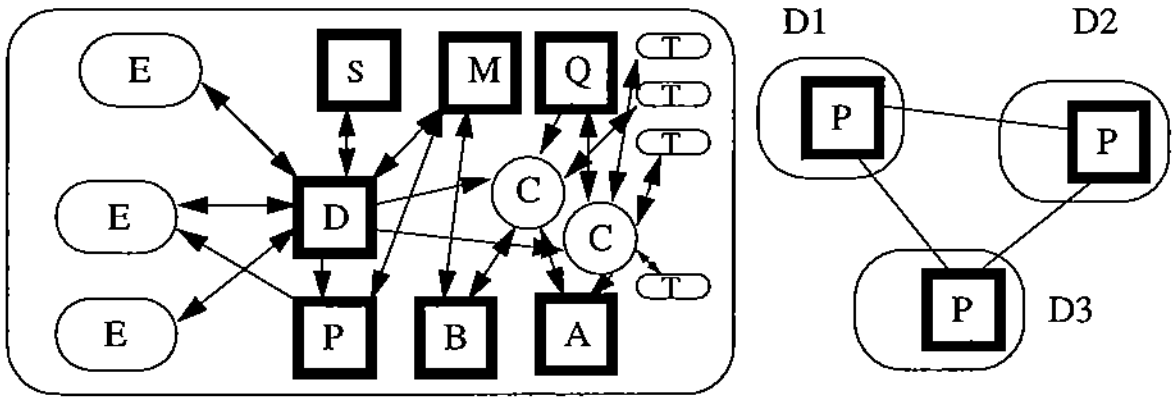

Figure 4. A Bond domain and an interconnection of several domains.

A domain is generally associated with a group of users with common interests. The cells of a domain are physically distributed over a computer grid. A domain grows and shrinks depending upon the needs of the community using it. The only cells active at any time during the lifetime of a domain are those providing Bond services, e.g. P, M, Q, D, B, A and so on. Domains may be connected with one another by establishing links between their persistent storage and global directory service cells. Our concern is to create an infrastructure suitable for inter-domain activities and to populate a domain for structural biologists.

3.3 Object communication using KQML.

Bond objects communicate with one another using KQML, an inter-agent communication language [12], [16]. KQML messages, called performatives encode basic abstractions like *asking*, *replying*, *achieving*, *subscribing* or *notifying*. There are several classes of performatives: informative like *tell* and *deny*, database performatives e.g. *insert*, and *delete*, basic query performatives as *evaluate, reply, ask-if, ask-about, ask-one, ask-all, sorry*, effector performatives like *achieve and unachive*, notification performatives *as subscribe* and *monitor*, networking performatives as *register, unregister, forward, broadcast, pipe*, facilitator performatives e.g. *broker-one, broker-all, recommend-one, recommend-all, recruit-one, recruit-all.*

KQML does not specify the contents of a message but allows agents to express an attitude relative to the contents of communication. The contents of a messages are partially encoded into the parameters, and partially derived from a common *ontology* known by all parties involved in a

communication act. In the following example a scheduler agent requires information about `program1`. A shadow of the `program1` object is created as a result of a search involving the directory of the local cell, the domain directory and possibly directories of other cells in a pattern very similar with the one for a service search, discussed next. The scheduling agent sends the following performative:

```
(stream-about : language Bond : ontology programs :
                reply with alpha : contents program1)
```

The shadow of `program1` responds with a series of performatives:

```
 (tell: language Bond : ontology programs : in-reply-to alpha
      :content (=val (source program1) (URI
      :cozia.cs.purdue.edu/home/dcm/programs/program1.c))
 (tell: language Bond : ontology programs : in-reply-to alpha
      :content (=val (executable program1) (URI
      :cozia.cs.purdue.edu/home/dcm/programs/program1))
```

Message patterns for different functions are discussed in depth in [6]. Figure 5 illustrates the message pattern to search for a service. The Bond object sends a query (`ask`) to the local directory. If the object capable to provide the service is not available locally, a query is sent to the domain directory. Cells with a copy of the object respond (`tell`) and a broker agent recommends one of the available choices (`recommend-one`).

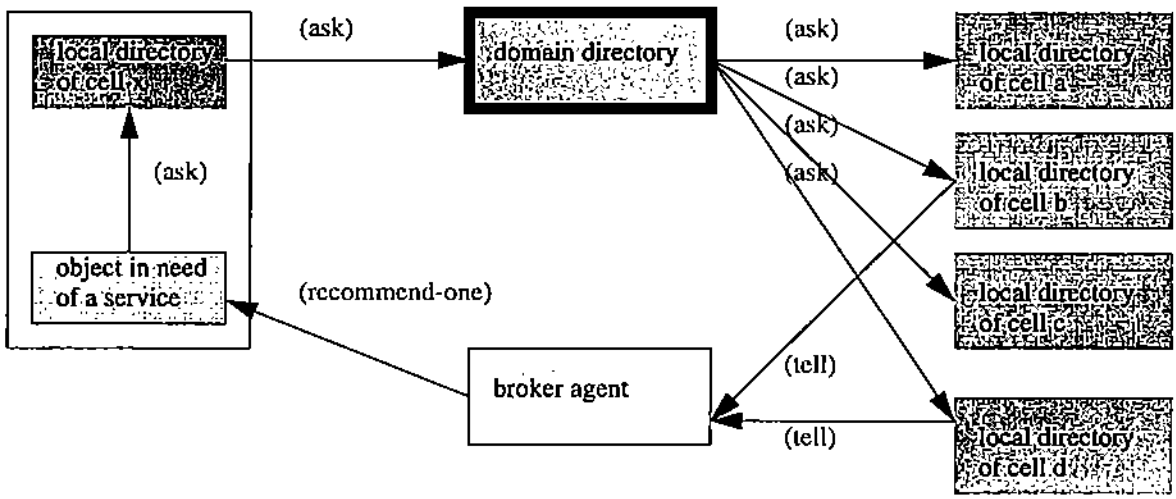

Figure 5. Message pattern to search for a service.

In Bond any object, active or passive may receive a message. The message is processed by the messaging thread of a cell (see Figure 3) which has one instance of the KQML parser. After parsing the message the messaging thread invokes the `say()` function of the destination object. If the incoming message requires a longer processing, a new thread is created for the object, otherwise the object's existing thread is instructed to carry out the required processing. Although all objects understand the syntax of KQML, their level of semantic understanding depends on their type. Different objects can carry out different conversations - a scheduling agent understands different messages than a database server.

We suspect that there is an isomophism between remote method call and message oriented systems, for a given functionality and a required level of inter-operability one can use either paradigm to build a system, subject to the same set of constraints. For a remote method call system like Corba, the basic question of inter-operability is "what methods am I able to call on the remote object and what are the parameters for these methods?". Corba solves this problems providing static and dynamic interface discovery methods. For the static case one provides the description of the objects interface in IDL, the Interface Description language and for the dynamic case Corba has interface repositories. The equivalent question for a message oriented system, like Bond, is "what messages does the remote object understand?". The Bond solution is based the definition of *subprotocols*.

A subprotocol is a closed subset of the KQML language, used to implement a specific functionality. Examples of the more important subprotocols in the Bond system are presented in Table 1.

Subprotocol	Who implements it	Function
Property access	All Bond objects	Supports read/write access to all properties of a Bond object.
Security	Some Bond objects	Used to establish trust relationships between Bond objects.
Monitoring	All Bond executables	Allows a Monitor to obtain information about the current state of an executable.
Checkpoint/restart	Some Bond executables	Support checkpoint/restart facilities.
Agent control	All Bond agents	Allows a Bond agent to start, stop, and control a remote agent.
Interface discovery	Some Bond objects	Allows an object to discover the syntax and the semantics of an unknown subprotocol implemented by another object.
Database access	All database servers	Supports creation, deletion and updating of objects from databases.
Scheduling	All scheduling agents	Supports scheduling of a subcontract.
User access	All wrappers	Allows the External agent to control the execution of a legacy program running under the control of a wrapper

Table 1: Subprotocols in Bond

Individual Bond objects implement a number of subprotocols. Every Bond object implements the property access subprotocol, shown in Table 2 and inherits all subprotocols implemented by its

ancestors in the *Bond object hierarchy* in Figure 1. One of the properties of an object is the list of the subprotocols it implements. This list can be interrogated using the property access subprotocol. When two objects need to communicate they first use the property access subprotocol, understood by all Bond objects to determine the set of common subprotocols. Then two objects communicate using the subprotocols implemented by both of them.

Performative	:content	Parameters	Description
ask-one	get	:property *name*	asks the value of the property *name*
achieve	set	:property *name* :value *new_value*	asks to set the value of the property *name* to value *new_value*
tell	value	:value *value*	reply to "get"
tell	ok		reply to "set"
sorry	-	:error *description*	an error occurred

Table 2. The property access subprotocol.

An exception to the access method described above is provided by the interface discovery subprotocol, which permits an object to learn the *syntax and the semantics* of an unknown subprotocol. Interface discovery may be used by agents to learn to communicate with other agents which provide a known service, but with different means. A good example is a user which can connect to the Internet using a PPP or an ISDN service. The service (connection to the Internet) is the same, however both service providers need a different set of commands and information in order to provide the service. In this situation the service provider can present its service access subprotocol to the user agent, together with the semantics of the messages in the subprotocol. The semantics specifies how the state of the service provider can be modified by different messages and how the changes in the state of the provider are mirrored in the messages it sends to the user. Knowing the syntax and the semantics of the subprotocol, the user agent can select the messages to be sent to bring the provider to the desired state, "Connected" in our example.

Table 3 presents a side by side comparison of Bond and Corba. In Corba the communication amongst objects is multiplexed on the ORB, while in Bond the communication is multiplexed to the **in** and **out** box of the messaging thread of a cell. Bond provides event waiting slots and subscribe/unsubscribe services [6] while Corba has an event service. Corba used an object trader service [20] to find services while Bond uses the directory service and brokers. Both Bond and Corba support static as well as dynamic access to objects.

4. Services, agents and contracts for metacomputing on a grid.

We change our focus from cells to domains and discuss services and agents necessary for meta computing on a grid of autonomous compute nodes. Our concern is the creation of an infrastructure for a virtual organization expected to use a virtual metacomputer. The servers and the agents within a domain are expected to provide services we are accustomed to, when confined to a single system, namely scheduling, monitoring, user interfaces, and so on. As pointed out earlier Bond is a facilitator, not a distributed system, its basic philosophy is to defer security and resource allocation mechanisms to the individual nodes of the grid.

There are some subtle differences between servers and agents. Servers perform a well defined function. For example the system monitor is a server; it ensures that critical domain services and agents are available at all time. When replicated, multiple system monitors have the same objective. Agents, on the other hand, are autonomous entities with their own agenda. For example the objective of a contract scheduler is to optimize the execution of a contract. Two contract schedulers may have conflicting objectives.

implementing...	Bond	Corba
communication channel	multiplexed to the inbox+outbox of the messaging thread	multiplexed on the ORB
messaging	native	messages as procedure calls
remote procedure calls	remote procedure calls as messages	native
event handling	subscribe/unsubscribe + event waiting slots	event service
finding objects	directory service	naming service
finding services	directory service + broker agent	object trader service
remote execution	resident	ORB
remote instantiation	realizing a shadow	life cycle service
static access	shadow + property access subprotocol	IDL + access functions
dynamic access	shadow + interface discovery subprotocol	interface repository service
multilanguage	preference for Java, but implementable in any OO language	yes

Table 3. Bond and Corba.

In an workspace populated with metaobjects, one can envision mechanisms to link metacomponents together and form a metaprogram, a new metaobject, capable to cary out a well defined computational task. A metaprogram is a static structure describing the data flow of a complex computation, using parallel, sequential and choice composition. To map a metaprogram onto a grid we define an abstraction called *contract* and an agent called *contract scheduler*.Contracts and

contract schedulers are metaobjects. A contract is a metaprogram in execution, the same way a process is a program in execution. A contract consists of sub-contracts, atomic actions performed by permanent or transient servers.

A domain is expected to support a set of *core services* available all members of the group associated with the domain. Core services are provided by *permanent servers* that often are replicated to ensure the desired quality of service. Yet one cannot expect to locate within a domain servers capable to provide all services required by all contracts submitted. There is also the issue of the cost associated to individual services. In our model, most services are started upon request by a contract scheduler. Such a *transient service* often runs a legacy program on behalf of the person who initiated the contract. A transient service may only be started on nodes of the computer grid where the initiator of the contract has computing accounts, the service runs with the privileges and authorizations available to the user, and terminates once it has accomplished its task. To start a transient service the contract scheduler locates an executable compatible with the target platform and a *wrapper* for the program. The wrapper is a control agent that supervises the execution of the legacy program and provides feedback to the contract scheduler regarding the progress of the computation.

A contract defines a temporary relationship amongst metaobjects expected to collaborate towards the goal of the contract. *A Virtual Object Network, VON,* is an abstraction for a set of metaobjects involved in a contract. The shadow mechanism described earlier allows Bond to establish a network of cooperating objects that need to inform each other about the progress of the individual sub-contracts.

Figure 6 illustrates Virtual Object Networks created by a scheduling agent assigned to execute a contract. The components of the contract are linked into a Virtual Object Network. The scheduler does not have copies of the data or programs involved in the execution of the contract, it contains only their shadows. In Figure 6, `Program 1` and `Program 2` are legacy applications, executed under the supervision of a wrapper. The virtual network is a dynamic object: as new data files are generated they are added to the virtual network. Another virtual network consists of the group of hosts available for the scheduler for executing the contract. The scheduler also creates a virtual network of the control agents needed to support the execution of the contract, in this case the authentication server and the monitor agent.

We turn our attention to the core services and agents of a Bond domain, [3], [16] and describe briefly the functions of each. Some of the services and agents are generic others are domain specific. The generic servers are the monitor, the dispatcher, the persistent storage and directory server, the software distribution server, brokers, authentication, and contract schedulers. Matchmaker agents and some arbitration agents are domain specific.

The monitor is used to start up core servers, continuously monitor their behavior, and replicate them when the load placed upon them exceeds a high water mark. To ensure fault tolerance a standby monitor runs continuously along with the monitor.The dispatcher runs at a well known network address, and provides initial user access to the system. Once a user is authorized to use the system, the dispatcher contacts the software distribution server which determines if the software running at the user's site is current and if not a JAR file including the current version is

downloaded.

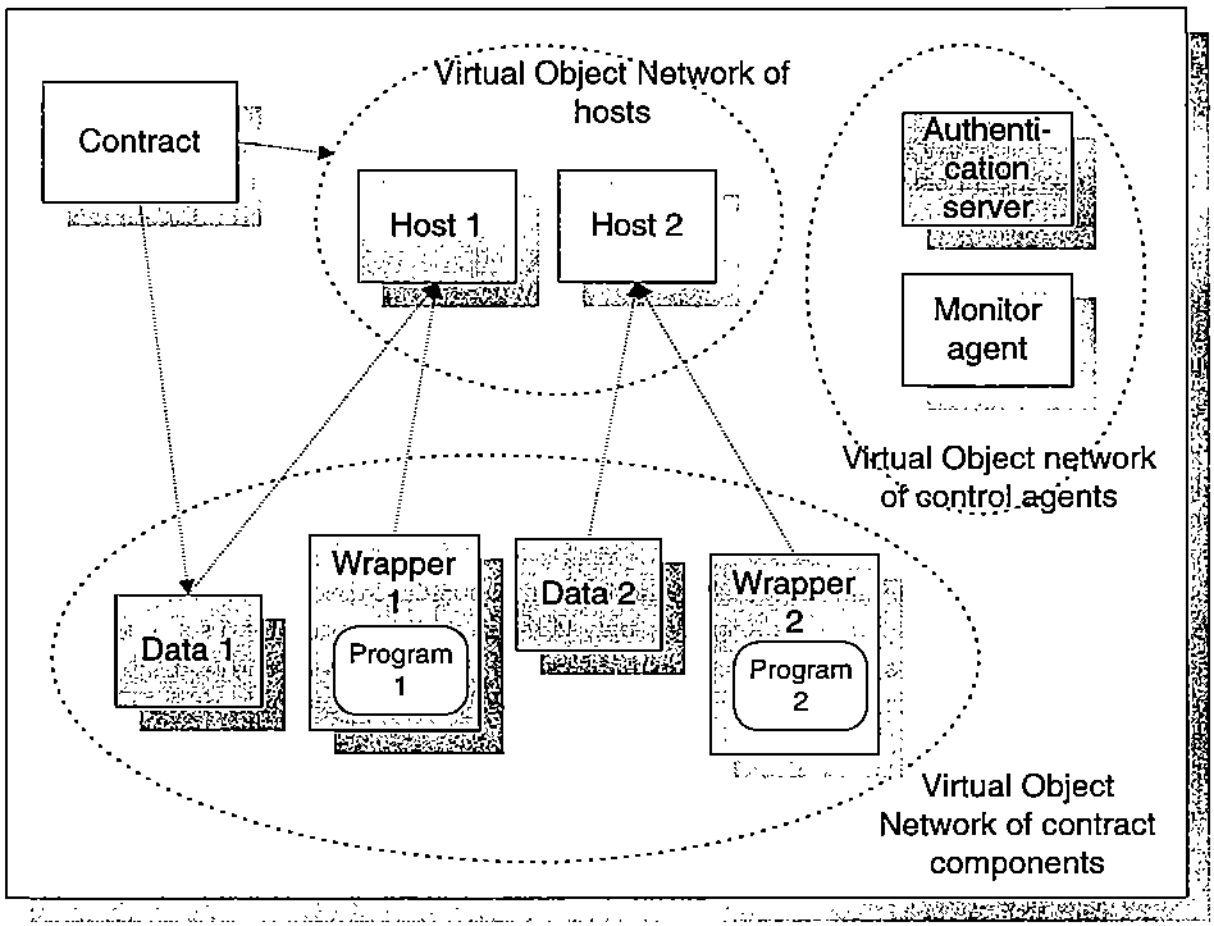

Figure 6. A virtual Object Network associated with a contract.

The persistent storage and directory server is responsible to create a local directory and a work-space whenever a new cell becomes active, and to maintain a list of active cells. At a later stage the persistent storage and directory server will ensure connectivity among different domains. The broker is involved in activities which require a selection process for example the service search shown in Figure 5.

The contract scheduler is expected to perform several functions: (a) map atomic activities required by a contract into services available within the domain, (b) locate servers capable to carry out individual services, (c) ensure the proper workflow, (d) implement a scheduling algorithm, and (d) support interactions with the user. A contract scheduler is a transient agent, started when an external agent submits a contract for execution. A user may submit several contracts at the same time. The contract scheduler acts as a proxy for the external agent, even if the user terminates the Bond session, the contract scheduler continues its activity until either the contract is successful or a user action is needed and the contract is suspended. When a new session is started the dispatcher provides a list of pending contracts and actions expected for each.

The authentication server facilitates creation of trust relationships using the protocol described below. An agent, **a** occasionally need to ensure the identity of an another agent, **b**. To do so **a** generates a random message and asks **b** to encrypt the message using its own key and send it back to **a**. Then **a** sends the encrypted message to the authentication server who has all the keys and is capable to decrypt the message and send the clear text back to **a** where a comparison with the original random string is performed. The protocol outlined above may be used by a trusted resident to start a new thread at the request of a trusted agent [6].

To support "user deadlines" a domain needs quality of service monitors to gather performance data regarding the load of nodes of the grid that may be used by a contract. We are primarily concerned here with soft deadlines and we assume that some knowledge about the expected execution time of individual subcontracts on some of the nodes of the grid exists [5]. We also assume that the individual schedulers accept some form of reservations. If some of these assumptions are not true than the only solution is to provide a "best effort" scheduling. In other words, once a subcontract is enabled for execution the contract scheduler uses its internal scheduler and the server mapper to locate the optimal target system available at that time to carry out the computations of the sub-contract.

	D	M	P	B	C	S	A	Q	Mm	Ar
IB	yes	yes	yes	yes	yes	yes	no	no	likely	likely
SB	yes	yes	yes	yes	yes	yes	yes	no	likely	likely
IQ	yes	yes	yes	yes	yes	yes	no	yes	likely	likely
SQ	yes	yes	yes	yes	yes	yes	yes	yes	likely	likely

Table 4. Services and agents for different domain models. The services and the agents involved are Dispatcher (D), Monitor (M), Persistent storage and directory server (P), Broker (B), Contract scheduler (C), Software distribution server (S), Authentication server (A), Quality of service monitor (Q), Matchmaker (Mm), Arbitration (Ar). The first eight are generic servers/agents, the last two are domain specific. The domains are classified based upon security (Secure/Insecure) and support for user deadlines (Qos/Best effort). IB=Insecure and Best effort, SB=Secure and Best effort, IQ=Insecure and Quality of service guarantees, SQ=Secure and Quality of service guarantees.

An arbitration agent is an agent capable to implement a certain semantics associated with conflicting requests from either external or even internal Bond agents. Arbitration agents are necessary for collaborative environments where different external agents may request conflicting alterations of the persistent storage. We expect arbitration agents to mediate actions requested by different contract schedulers who have conflicting objectives. Last but not least, match-making agents are ultimately responsible for implementing the "lock and key" mechanisms discussed earlier.

The actual configuration of services and agents depends very much upon the requirements of a specific domain. Table 4 presents a summary of core servers and agents and classifies domains along two dimensions, security and support for user deadlines. An Insecure & Best effort, IB, sys-

tem does not support establishment of trust relationships among objects and provides best effort scheduling. At the other extreme a Secure-Quality of service guarantee, SQ, system includes an authentication server and a quality of service monitor.

5. Conclusions

In early 90's at a conference on Massively Parallel Systems, Gordon Bell gave a keynote address entitled "Massively Parallel Computing or Computing for the Masses". His talk preceded by a few years the Twilight of the Gods - Gotterdammerung, the rapid extinction of expensive, custom built systems used by the elite few to solve the most difficult computational problems. Since then we have witnessed the fast retreat of large computer manufacturers from the supercomputing business and their migration to the lucrative business of enterprise and personal computing for the "masses".

Since Gordon Bell's prophetic talk substantial changes in computer science and engineering have already occurred or are imminent. Gigabit networks will be available in the immediate future. The Web is widely used by a growing segment of the population. Java is becoming the language of choice for network applications because it supports code mobility. Corba is gaining popularity and its clean design simulates a new wave of thinking in distributed systems. The time seems ripe for network computing, a new computing paradigm which emphasis network versus local resources. In this framework a Petaflops system can be a "virtual" computer consisting of thousands of autonomous nodes interconnected by very fast networks.

The infrastructure to ensure usability of such a metacomputer does not exist. By its very nature a computer grid is heterogeneous and accommodating heterogeneity at the scale of a national computing grid and ensuring usability and dependability of such a metacomputing environment is a daunting task, unless we can find the means to transfer this task to the computers themselves.

In this paper we present our philosophy of building complex systems out of components capable to recognize each other by a lock and key mechanism similar to the one used by proteins. We advocate a pragmatic approach of building complex systems, one should attempt to find a glue to bond together existing hardware and legacy software the components developed independently, with components using new technologies. The same infrastructure should be used in all forms of computer life from scientific software to electronic commerce. To accommodate the inherent heterogeneity of any complex system we propose to define information objects that describe in the most minute detail the actual software, hardware, and data objects we want to compose. These objects should be classified into disjoint classes, with objects being able to inherit properties from their ancestors and acquire new properties.

Then we define an architecture for communicating objects using KQML. Bond is a Message Oriented Middleware, whose main ingredients are a lightweight mechanism to establish long term relationships amongst objects and a "universal" language which enables objects to communicate attitudes about information, being indifferent to the format of the information itself. *Shadow objects* provide a high level abstraction for a unidirectional communication channel linking two objects together. When an object needs to communicate it involves the directory service and, when the object is found, a shadow of the remote object is created and the connection is established.

There are several significant differences between Bond and other metacomputing architectures. The first is the approach in accommodating heterogeneity. We advocate the creation of a distributed knowledge bases and the creation of intelligent agents capable to reason about objects and work towards fulfilling their agendas. Another one is the systemic approach. We believe in creating an infrastructure that is quite general and applicable to a variety of applications from scientific computing to electronic commerce and then attempt to populate "domains" with application specific agents and knowledge. For example we are confident that once the Resource Specification Language, RSL, [13], [14], gains popularity we'll be able to incorporate it and have the objects involved in scheduling carry out dialogs in RSL.

Additional information about the Bond project is available at the following Web address: `http://www.cs.purdue.edu/homes/sb/`. Access and documentation for the Bond 1.0 server at: `http://bond.cs.purdue.edu`.

6. Acknowledgments

A server of Institute for Scientific Information (http://isi1.med.iacnet.com/ISI/CIW.cgi) provided us with the humbling experience of network access to a citation index and the sobering reality that many papers published by the best computer science journals are never referenced. At that time it occurred to us that the solution might be to write multimedia papers. This paper's hyperlinks to Richard Strauss music are as follows: Don Juan (Section 1), Tod und Verklarung (Section 2), Don Quixote (Section 3), Salome: Tantz der sieben Schleier (Section 4), Also sprach Zaratustra (Section 5).

Kyung Koo Jun is developing code for Bond 2.0 and K.C. VanZandt contributed with helpful comments. M. Sirbu wrote most of the code for the Bond Shell and Bond 1.0.

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 the Intel Corporation, and by the Computational Science Alliance and the NCSA at the University of Illinois.

7. References

1. A. Avizienis. *Infrastructure-Based Design of Fault Tolerant Systems*, Proc of the 1998 IFIP Workshop on Dependable Computing and its Applications, DCIA'98, pp 51-69.
2. C. Branden and J. Toose. *Introduction to Protein Structure*. Garland Publishing 1991.
3. L. Bölöni, K.K. Jun, and D.C. Marinescu. *QoS and Reliability Models for Network Computing*. Department of Computer Sciences, Purdue University CSD-TR #97-051.
4. L. Bölöni. *Bond Objects -- A White Paper.* Department of Computer Sciences, Purdue University CSD-TR #98-002.
5. L. Bölöni and D.C. Marinescu. *Robust Scheduling of Metaprograms in a Nondeterministic Environment*, Department of Computer Sciences, Purdue University CSD-TR #98-003.
6. L. Bölöni, K.K. Jun, T. Daniels, and D.C. Marinescu. *Message patterns in the Bond Distributed Object System*, Department of Computer Sciences, Purdue University CSD-TR #98-004.
7. K. Mani Chandy, A Rifkin, Paolo A.G. Sivilotti, J. Mandelson, M. Richardson, W. Tanaka,

and L. Weissman. *A World-Wide Distributed System Using Java and the Internet, IEEE International Symposium on High Performance Distributed Computing*, August 1996.
8. K. Mani Chandy. *Caltech Infospheres Project Overview: Information Infrastructures for Task Forces.*
9. M.C. Cornea-Hasegan and D.C. Marinescu. *SBL, The Structural Biology Language*, Department of Computer Sciences, Purdue University CSD-TR #94-008.
10. 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
11. 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.
12. T. Finin, et al. *Specification of the KQML Agent-Communication Language.* DARPA Knowledge Sharing Initiative draft, June 1993.
13. I. Foster and C. Kesselman. *Globus: A Metacomputing Infrastructure Toolkit.* International Journal of Supercomputing (to appear), 1998.
14. I. Foster and C. Kesselman. *The Globus Project: A Status Report.* Proc. Heterogeneous Computing Workshop (to appear), 1998.
15. A. Grimshaw and W. Wulf. Legion - a view from 50,000 feet. Proc. 5-th IEEE Symp. on High Performance Distributed Computing, pp. 89-99, IEEE Press, 19996.
16. Y. Labrou and T. Finin. *A Proposal for a new KQML Specification.* UMBC TR-CS-97-03.
17. M. Lewis and A. Grimshaw. *The Core Legion Object Model.* Proc. t-th IEEE Symp. on High Performance Distributed Computing, IEEE Press, 1996.
18. 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.
19. D.C. Marinescu. *Software Development for Intranet Applications.* Proc of the 1998 IFIP Workshop on Dependable Computing and its Applications, DCIA'98, pp. 31-50, 1998.
20. R. Orfali and D. Harkley. *Client/Server Programming with Java and Corba.* Willey, 1997.
21. M.G. Sirbu and D.C. Marinescu. *A Parallel Virtual Environment.* Proceedings of HPCN Europe'96, pp. 722--728, Lecture Notes in Computer Science, Volume 1067, Springer Verlag, 1996.
22. M.G. Sirbu and D.C. Marinescu. *A Scheduling Expert Advisor for Heterogeneous Environments.* Proceedings of the HCW'97 Heterogeneous Computing Workshop, pp. 74-82, IEEE Press, 1997.
23. L. Rodrigues and P. Versissimo. xAMp: *A Protocol Suite for Group Communication.* Proc SRDS-11, pp. 112-121, 1992.
24. M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillermont, F. Hermann, C. Keiser, S. Langlois, P. Leonard, and W. Neuhauser. *Overview of the Chorus Distributed Operating System.* Chorus Systems Technical Report CS-TR-90-25, 1990.
25. R. J. Stroud. *Transparency and Reflection in Distributed Systems.* ACM Operating Systems vol. 22, no. 2 pp. 99-103, 1993.
26. *Pentium Pro Family Developer's Manual*, Vol 1,2,3; Intel Corporation: Mt. Prospect, IL, 1996.