

# Wrapping and Interoperating Bioinformatics

## Resources Using CORBA

RobertStevens<sup>1,2</sup> and Crispin Miller<sup>2</sup>

Department of Computer Science<sup>1</sup> and School of Biological Sciences<sup>2</sup>

University of Manchester

Oxford Road

Manchester

M13 9PL

robert.stevens@cs.man.ac.uk

November 19, 1999

### **Abstract**

Bioinformaticians seeking to provide services to working biologists are faced with the twin problems of distribution and diversity of resources. Bioinformatics databases are distributed around the world and exist in many kinds of storage forms, platforms, and access paradigms. To provide adequate services to biologists, these distributed and diverse resources have to interoperate seamlessly within single applications. The Common Object Request Broker Architecture (CORBA) offers one technical solution to these problems. The key component of CORBA is its

use of object orientation as an intermediate form to translate between different representations. This article concentrates on an explanation of object orientation and how it can be used to overcome the problems of distribution and diversity by describing the *interfaces* between objects.

**Keywords: CORBA, Object Orientation, Interface, Distributed programming, Heterogeneity, interoperation.**

## **1 Introduction**

biology is a data rich discipline. There are now hundreds of databanks and tools that work over those resources [1]. It is this piecemeal growth that has presented the bioinformatics community with a large and difficult problem. This is a technical problem of enabling many, very different resources to work together (interoperate) to make more complex, sophisticated applications, than is possible with one resource alone. The Common Object Request Broker Architecture (CORBA) is one solution to such problems that is an industry standard. This paper describes both the problem and introduces CORBA, which is likely to form part of the solution.

Instead of only wanting to ask simple questions of one resource, biologists now wish to ask more complex questions, over many of these resources. When a programmer tries to use any of these resources in another program, he or she is presented with two major obstacles: First, the resources are

*distributed* around the networks of computers; second, there is great diversity or *heterogeneity* in these resources. The distribution of these resources means that complex communications have to be set up between the client program and the remote resource. Heterogeneity is a larger problem. These resources run on platforms with many different operating systems and are implemented in many different programming languages, such as C++, Perl or Java (TM). The databanks are stored using many different paradigms — Relational, object and object-relational databases as well as numerous different flat-file formats. The data stored in flat-files tend to be accessed by utility programs that retrieve records by one parameter alone, whereas most databases have some form of query language. The tools that work over these data also present different means of operation to users.

To the programmer trying to use these resources within one program, the fact that all the resources 'look different' means a considerable amount of effort has to be expended. When including external resources into a program, the programmer has to translate between the representation used externally into that of the host program. For example, a Java program needs to access a remote database written in C++, and another program written in Perl. In order to do this, two things must happen. Firstly, the functions in the remote objects need to be made accessible to the Java program. Secondly, the data itself needs to be translated from something the remote objects understand into something the Java program is familiar with. If this is done to the point where the external programs look like a real Java object, it can be said that the external resources have been *wrapped*. Wrapping is an appealing solution because, once a resource has been wrapped, it can be re-used by different

programs, if it is in a form understood by the language used. This would mean that the hard work expended in performing the integration does not have to be repeated.

There are many ways of achieving this wrapping; most programming languages have the constructs for establishing connections across networks and performing subsequent transformations. Java, for example, has remote method invocation (RMI), to access distributed objects and Java Database connectivity (JDBC) for accessing databases. These, however, only work in a Java environment. In a more heterogeneous environment, wrapping has to take place for *each* programming language used and repeated in *each* institution that requires usage of the external resources.

This highly heterogeneous world needs a generic solution. Instead of having as many views of the world as there are resources, wrapping would be much easier if there was one view of the world. This view would form a common intermediary between client programs and the distributed and heterogeneous targets they wish to use. CORBA presents such a common view of the world by seeing it from an *object* modelling perspective. The concepts underlying an object orientated view are introduced in Section 2. Target resources can be described in this common language and the code generated automatically, in the programming language of choice, to make those target resources appear as if they are part of the local host application.

In the ideal world, the target resource's owners provide this description of their resource and potential clients download this description and generate the code that establishes all communications to the target and makes it appear

in their application. a core feature of the CORBA specification is the Interface Definition Language (IDL). This language is used to describe what target resources perform and it can be used by CORBA compliant tools to generate code for both providing access to the services and the means for the services to be accessed. What CORBA does is introduced in Section 3 and how the IDL works is described in Section 3.1.

That individual resources make this access available is not enough. For multi-resource applications to be developed, many resources have to work together seamlessly. This working together, whilst remaining separate, is called *interoperation* (integration is the forming of one large united resource). For interoperation to be available, the providers of the common views described above have to describe their resources in such a manner that part of one resource can be passed to another without the intervention of the host developer. For example, the providers of a sequence similarity search tool have to make available services that accept protein sequences in a common form that can be adopted by providers of protein databanks. The alignments returned would also comply to a common description.

Obviously, such a unified, generic approach needs much effort, co-operation and planning. This presents considerable design problems. Section 2 introduces some example object orientated views of biological sequences and analysis tools. These examples are taken through this article and are shown in Section 3.1 in their IDL form.

CORBA is the central component of the Object management Architecture (OMA) proposed by the Object Management Group (OMG). Section 4

describes the OMG and its role in promoting an object view of the world. Within the OMG, there are task forces that deal with issues for particular domains. The Life Sciences Research Group (LSR) is the task force that attempts to provide standard CORBA based solutions to wrapping and interoperating within the life sciences community. The OMG mechanisms and the work of the LSR are discussed in Section 4. Finally, Section 5 draws together the potential role of CORBA in the bioinformatics domain. The section also discusses some of the issues that have to be resolved when deciding whether to take a CORBA based approach in a project.

## **2 Object Orientation**

The CORBA specification uses an object orientated view of the world to allow generic, intermediary descriptions of resources to be made. An understanding of the object modelling view is vital to an understanding of the principles of CORBA. This section gives a brief introduction to the concepts, but many texts are available that give more depth to these ideas (see, for example, [3]).

Object orientation considers the world to be made of ‘things’ that interact with one another. One of the first tasks a computer scientist tackles when building a new database is to decide what ‘things’ in the world are important, and how they interact with each other. Over the last 30 years this process has been refined and developed to the point where a large number of tools and techniques, such as Extended Entity Relationship (EER) diagrams [8] and the Unified Modelling Language (UML) [2] exist to aid in the design process. These tools and techniques not only facilitate analysis and design, they also

allow different computer scientists to communicate in a common language and to share solutions to common problems. The prime example of this is the text 'Design Patterns' [6]. A more gentle introduction, aimed at Java programmers can be found in 'Design Patterns in Java volume I' [7].

Object oriented programming (OOP) languages apply the same ideas to computer programs as are applied to database modelling – splitting domains into logical components which interact with each other. For example, each of the buttons on a web browser is an object which, when pressed, sends a *message* to the browser (another object) instructing it to do something, such as reload the current page.

Each button is a self contained unit, which knows how to draw itself on the screen, what text it should be displaying ('reload', for example), and what message it should send when it's pressed. This is known as *encapsulation*. An object knows about its data (the button's 'text'), and what actions it can perform. Both data and actions are encapsulated inside the same object.

From the outside, how an object's behaviour is implemented is invisible.

Other objects interact with it through its *interface*. An object's interface shows what behaviour that object has to offer to the world, but not how that behaviour is implemented. CORBA works through the description of object interfaces and then mapping these descriptions to programming code that connects objects over networks and different kinds of underlying implementation.

The OOP paradigm offers a number of advantages. Firstly, in a big program, it can be hard to locate all the possible places where a piece of the data is

being changed. This is one reason why the millennium bug is so hard to fix – thousands of lines of code need to be checked to see if they are handling dates. In a well designed object orientated program, the only code that deals with dates should be in the 'Date' object. Secondly, if a better object becomes available, a millennium compliant one, it is possible to simply substitute the new object for the old. Thirdly, by breaking a program into smaller subunits with well defined boundaries or *interfaces*, each an object, it is possible to build and test each piece in isolation.

Once the interface to an object is defined, a programmer can start implementing its behaviour without needing to consider what other programmers are doing to the insides of their own objects. As an object's behaviour can only be accessed via its interface, users of an object are shielded from exactly how that behaviour is implemented. An interface is somewhat like a contract, offering what objects of that type can do, but without saying how those services will be performed. How those services may be performed may change, but the contract, i.e. the interface, stays the same.

As well as encapsulation, OOP has the notion of *inheritance*. For example, in a graphical user interface, every object – be they buttons, scroll bars, places in which to type text, need to know where they are on the screen, and to whom they should be sending messages. In an OOP this can be implemented by defining an object called for example, 'GUIComponent', which knows where it should be on the screen and who owns it. Other objects such as buttons become subtypes of GUIComponent – its children. These objects inherit the data and methods from their parent, but add some extra functionality themselves. For example an IconButton might be a subtype of TextButton,

with the extra lines of code necessary to draw a picture on the button's surface.

An important distinction to make is between the definition of a *class* of objects, and actual *instances* of that class. For example, a web browser's toolbar is populated by a set of objects, which are all members of the class 'Button'. The class definition says that a button has some text, and an icon, etc., but it says nothing about the actual text or icon on an individual button. In order to do this, it is necessary to create an instance of the class 'Button', and tell it what text it should be displaying. A class can be thought of as a recipe that tells a program how to make a particular type of object.

The potential that OOP has for defining clean interfaces to objects naturally leads to the idea of standards. Standard interfaces have helped the world wide web achieve its current status after only a decade. these standards range from HTML – the common language that webservers and browsers use to communicate with each other, to the programming libraries that allow developers to implement web browsers which are all capable of running programs written in the language Java.

## **2.1 Some Example Bioinformatics Object Classes**

In order to turn these notions into something more concrete, we will explore the way objects can be used in bioinformatics to address some of the issues described in the introduction. Note that there are a number of different ways to achieve the same ends, and that there are many possible solutions to the problem, each with their own merits. The example here is greatly simplified

in order to make it small enough to fit in this article and illustrate the essential concepts.

Biological sequences, from a similarity search perspective, are a list of letters. They have length, and they have a unique identifier – an accession number. The fragment of pseudocode in Figure 1 defines the class `BioSeq`. The `private` keyword in the pseudocode tells the programming language that `sequence` can only be seen by `BioSeq` objects - nothing else can access the data stored in `sequence` directly. Instead, it must be accessed via one of the public methods of the class. This is how the *interface* is defined – it keeps the inner workings of `BioSeq` hidden. This is advantageous because the inner representation can be changed, for example, by compressing the sequence so that it takes up less storage space, without affecting the way it interacts with other classes of object via its interface.

[Figure 1 about here.]

The interface to `BioSeq` offers a few of the standard manipulations needed for biological sequences – finding by accession number, finding length, etc. Nucleic acid sequences are a kind of biosequence which can be translated into a Protein sequence – an addition to the interface. Note that although this is incorrect from a strictly Biochemical perspective (mRNA can be translated, DNA cannot), we are modelling *bioinformatics* and it is sensible to allow any Nucleic Acid sequence to be translated. This can be shown in the following pseudocode:

```
class NASEq extends BioSeq {  
    ProteinSeq translate(int frame) { ...; } throws NoSuchElementException;
```

```
}
```

Note that `NASeq` is a *subtype* of `BioSeq`. It inherits all of the methods from its parent, such as `getLength()`, but adds a method for performing translation.

The translation method returns objects defined in another kind of class (`Protein`), which is not shown in this definition. thus, the `NaSeq` class 'extends' the `BioSeq` class, keeping the same behaviour and interface, but adding some additional functionality to the class, and methods to the class interface.

A DNA sequence is a type of `NASeq`, that has a forward and reverse strand. It can also be translated in six frames. This behaviour can be inherited from the `NASeq` class, with only the reverse complement method added to the child class. This is shown as follows:

```
class DNASeq extends NASeq {  
    DNASeq getReverseComplement();  
    ...  
}
```

`DNASeq` further specialises `BioSeq`; it is still a kind of `BioSeq`, but has added services to its interface, so that the general form of sequence can be used as DNA.

Sequence analysis has similarity search tools that take sequences and return a similarity score. Some tools return kinds of alignments, others return different objects, such as dot plots or motif sets. A good object model will unite these kind of tools in an inheritance hierarchy, with classes representing actual tools

such as BLAST at the bottom of the tree and common behaviour inherited from base classes. For example, the pseudocode in Figure 2(a) shows a BLAST class definition. The superclass `AlignmentTool` supplies methods such as `setQuerySeq(BioSeq b);` and `setDatabase(Database db);` that are common to all alignment tools. The BLAST class itself adds methods specific to that alignment tool, such as setting scoring matrix, expected and probability thresholds. The class, `BLAST`, also implements its own way of performing sequence alignments in the method `doQuery()`, that returns a collection of BLAST-results in the collection type `ResultsSet`. This is a trivial example; a real solution would be very much more sophisticated.

So, a program might run a BLAST search by first creating an instance of the class `BLAST`, together with an instance of a database and assuming there is already a sequence called `aSequence` defined we can get a short code fragment as seen in Figure 2(b). As the implementation is encapsulated inside the objects, the code fragment only shows the essentials of performing a BLAST search.

[Figure 2 about here.]

The example in Figure 2(b) shows how BLAST searches might be performed in an object orientated program, where all the resources are available in the host language. In reality, however, Bioinformatics tools and resources are distributed and heterogeneous. They are not offered to their users in a form that easily falls into the example code fragment. Different search tools exist on different computers, in different parts of the world, with interfaces that do not easily interoperate.

In the next two sections of This paper the CORBA standard is introduced as a possible solution to these problems. From a programmer's perspective, there is the need to create an instance of BLAST and other objects, that appear to reside on the local computer, but actually exist somewhere else. If this can be achieved, the code fragment above does not need to change significantly.

### **3 What Does CORBA Do?**

CORBA takes this object orientated view of the world. CORBA offers a mechanism by which resources that are not represented in the object orientated paradigm can be made to appear as objects. What is more, CORBA includes in this mechanism the ability to make resources running on different types of computer, using different languages, work together. CORBA provides a mechanism by which it is possible to make objects on remote systems appear as if they are present within a program running on a local system. Not only does it make remote objects appear local, it makes remote resources that are not objects appear as if they are so. CORBA enables programmers to make client-server applications that work across different platforms and networks (a server 'serves' information to a client program).

The host program has an object that 'appears' to be there in the program, but it is really an empty shell. The real object exists on the remote system. A request made for some behaviour on this empty shell has to be sent to the real object and the information elicited by that behaviour returned to the host program.

This is a complex procedure, best performed by an expert. Just as a financial broker may be used to control complex financial transactions, requests

between objects can also be brokered. An Object Request Broker (ORB) manages the requests between objects in a CORBA based system — it brokers requests between objects.

the CORBA bus is the communication system between objects used by an ORB. Different vendors supply their own ORBs, so one person's client may use a different ORB to the one being used by the server with which they wish to communicate. This is the importance of the *common* part of the Common Object Request Broker Architecture – it is a common architecture for ORBs. One of the specifications in the CORBA standard is that ORBs will interoperate. This means that one vendor's ORB will work seamlessly with another vendor's ORB. Figure 3 shows the relationship between client and server objects, their ORBs and the CORBA bus along which they communicate.

[Figure 3 about here.]

As well as providing this mechanism for communicating between client and server objects, other standard specifications for services are offered to support the use of distributed objects. When objects are distributed they have to be found by programs that wish to use them. Thus, the two fundamental services offered are the naming service, that finds objects by name, and the trading service, that finds objects by the requests they answer. Other services include object life cycle services, event services and security services. Many other services are specified within the CORBA specifications, but they all support the use of distributed objects and make use of the CORBA bus to communicate. The CORBA standard specify standard interfaces for these

services, which may be implemented by CORBA compliant implementations.

### 3.1 How Does CORBA Work?

As described in Section 2, objects communicate through their interfaces. The key component of the CORBA system is the Interface Definition Language (IDL). As the name suggests, IDL is a language that describes an object's interface — It does not, however, describe how the behaviour offered by that interface is implemented. Figure 4 shows some IDL descriptions of the `BiOSeq` and related classes from Section 2.1. This IDL description is declarative, it only defines what there is in an object's interface and not how it is implemented. Thus, it is independent of any individual programming language, though it looks a little like C++.

[Figure 4 about here.]

This IDL description is a definition of the interfaces of the classes required to appear in the local system. It has a C++ like syntax and describes the types and operations available in the classes' interfaces. Space does not permit a detailed explanation of these IDL descriptions. It will be apparent that the IDL descriptions readily map to the interfaces of the classes from Section 2.1. Some features are, however, worth pointing out.

In the IDL descriptions of the classes in Section 2, only the *interfaces* to the classes have been described. In the IDL descriptions, the keyword **interface** denotes a class and differs from the class definitions in Section 2.1 in two ways. Firstly, the private data seen in the `BiOSeq` class does not exist in the

IDL description. The IDL description only describes the interface; how the data is implemented is a job for the target resource. Secondly, for the same reason, the implementation details seen in the class definitions (between '{' and '}' in method definitions), are also removed. The IDL descriptions only contain information about the interface to the target implementation – how the target is actually implemented is hidden.

Otherwise, these IDL descriptions have essentially the same constructs as many of the target languages to which it is mapped. Elements of the IDL have to be mapped to a lot of different types of programming languages, not all of which are object oriented. As a result, the IDL is very precisely defined and may sometimes seem inflexible. As CORBA is a generic solution, some of the mappings from IDL to target languages may seem ungainly.

How information is returned from targets via the interface is also a significant consideration. By way of illustration, consider a BLAST server that creates an object containing the results of a search. Either, the whole object can be returned, across the network, to the program that made the request, or it can be kept at the server side and only an object reference returned. In the latter case, the program asks for each individual bit of data as and when it needs it. Deciding which method to use can have a major effect on a program's performance, because networks tend to be slow, ORB's performance vary, and the overheads involved with sending packets of data wrapped up in the CORBA protocol can be quite high.

It is important to note that the IDL description of an interface describes the syntax of that interface. That is, it describes the names of an object's methods,

and what data it sends and receives. It does not describe *what the methods actually do*. A small amount of semantics are described, since the IDL gives the types of arguments to, and return types of, the methods or operations available via that interface, but an IDL is basically a structural definition. The deeper semantics of what behaviour the target object actually possesses are only implied by the natural language names of the types and operations described. A user has to rely on a combination of these names, and documentation, to divine the behaviour of an object.

An IDL compiler takes the IDL description of an interface and turns it into code, in a particular programming language, for making client and server representations of the object. Each vendor's ORB may have many IDL compilers, each producing code for a particular language. The CORBA specification describes how each language is *bound* to the IDL syntax. Bindings exist for many languages including: C++, C, Java, Perl and many more. ORBs are produced for specific platforms and the IDL compiler produces code that runs on that particular platform. An IDL description of an interface can be taken, compiled on any platform to produce code for either clients or servers. As the IDL is platform and language independent and the IDL compilers are platform dependent a general mechanism exists that can connect hosts to targets running on different platforms using different languages.

On the client side, this automatically generated code is simply included into the program and instances of the class can be made and used almost normally. So, the code fragment for sequence analysis in Figure 2(b) looks the same, except for some standard calls to invoke the ORB and find the target object.

On the server side, similar code is generated, but this time the system to be wrapped is fitted into the skeleton code generated from the IDL description. If the target system does not already exist, it is written according to the outline or skeleton dictated by the interface description.

Stubs and skeletons are the glue by which client objects are connected to the target object via an ORB. Stubs are the methods that the programmer appears to invoke on an object in the host program. Of course, the behaviour invoked by that request is not implemented in the host, but remotely, on the server side by the target object implementation. The IDL compiler for a particular language generates skeleton code for the target object, which are the counterparts of the stubs. These skeletons provide a frame-work in the target language for a particular request. The programmer on the server side has to insert the appropriate code in this skeleton to answer the service requested upon use of the corresponding stub.

For example, there is an operation in the `BioSeq` class called `attribute string accession_umber ;`. In the server side code generated by the IDL compiler, within the code for the `BioSeq` class, will be the outline or skeleton for this method. The server side programmer then has to write the code that implements the accessor function that fetches and sets accession numbers for a biosequence.

## 4 The Object Management Group and the Life Sciences Research Group

The CORBA specification is a central component of the technology specifications supported by the Object Management Group (OMG). The OMG is a consortium of Industry, vendors and academics who seek to promote the use of object technology in distributed systems. The OMG does not implement systems, but adopts and supports standard specifications for systems. CORBA is a specification for a standard ORB architecture and commercial developers implement the specification. The OMG has a mechanism by which the need for a standard solution can be proposed; information on requirements gathered; solutions proposed; example implementations published and specifications adopted. The overall aim is to arrive at technically excellent, commercially viable and vendor independent specifications for distributed object systems. All these stages are open to public scrutiny and specifications are adopted by majority decision of the consortium members. In this way, standards are more likely to be adopted across the community.

The OMG promotes a complete architecture for supporting distributed object systems. This is the Object Management Architecture (OMA). It describes an **object model** that defines how distributed objects can be described. The **reference model** characterises interactions between those objects. The **services** described in Section 3 are also part of the OMA. The ORB is a component of this architecture, that facilitates communication between objects. Technologies adopted by the OMG have to conform to the OMA – in

order to facilitate interoperation.

The OMG home page (<http://www.omg.org/>) is a rich resource for information and materials about all aspects of the OMG. For instance, a beginner's guide to CORBA may be found at <http://www.omg.org/corba/beginners.html>. Specifications, with useful descriptions of how they work, including the CORBA specification can also be down loaded from this site. The 'object store' is an OMG web-resource describing most of the text-books available on CORBA and distributed systems.

As well as the general promotion of the OMA, the OMG has domain specific groups that seek solutions to problems in their own domain (sometimes these become widely used as many problems cross domains). One of these Domain Task Forces (DTF) is the Life Sciences Research Group (LSR). The LSR home page may be found at <http://www.omg.org/homepages/lsr>. Some of the current activities of the LSR can be seen in Table 1.

When a DTF proposes the need for a standard solution to a problem, it may first issue a Request for Information (RFI). This gathers information on the requirements and scope of the problem and its solutions. This information is collated and a specification of what is needed in a solution drawn up. This is issued as a Request for Proposals (RFP). Members of the consortium then make responses to this RFP, giving details, down to the IDL description level, on how they would match the specification. When a particular proposal is accepted, an implementation has to be offered within one calendar year, otherwise the proposal is dropped. This open mechanism is designed to

spread investment of effort and risk across consortium members and aims to make it more likely that standards will be adopted and followed.

[Table 1 about here.]

## 5 Summary

This article has introduced the basic idea behind the CORBA specification. It has attempted to describe what CORBA is used for; the problem which it addresses and the basics of how it works. This is neither the forum for a detailed discussion about using CORBA within applications nor should a programmer be able to implement CORBA based solutions after reading this article.

CORBA may be used as a technical solution to the problem of distribution of resources and the heterogeneity in the implementations of those resources. Both factors make it difficult to successfully develop applications that interoperate between bioinformatics resources. CORBA IDL offers a mechanism by which a common intermediary view can be made, that brings all the heterogeneous implementations within the same paradigm. CORBA IDL is used to describe interfaces of objects that represent the target resource. This declarative description can then be used both to generate stub code for request invocation on a client, and skeleton code to wrap up a target resource in the object view.

An IDL compiler generates both client and server code automatically from the IDL description. Separate IDL compilers exist for each language for which

IDL bindings exist. ORBs exist for many platforms and operating systems and the IDL compilers generate code that can interoperate with ORBs on other platforms. The IDL allows a common object view to be used to wrap up the diverse sources. The communication protocols built into the CORBA specifications allow all the code to be generated for allowing remote objects to be included into local systems.

The LSR DTF of the OMG seeks to promote standard interfaces for the Life Sciences community. Like all standards, this common view is only good for the community if the common view and its implementation fulfil the requirements of the community. CORBA is still a developing technology: it is a standard that is still being completely consolidated, though its use in industry is widespread. As a result programming to use some of the most recent developments in CORBA gives the sensation of being on slightly shifting sands. Secondly, the task of completely wrapping an object so that it is a generic resource that can be incorporated in any program can be rather large. Hence, the long standing and still unfulfilled promises to provide CORBA IDL based servers to some common bioinformatics resources.

Other technical issues arise – most notably that of performance.

Bioinformatics often involves moving a large number of relatively small objects between resources. The protocol employed by CORBA may have a fairly high overhead, so that, for this task, it can be rather inefficient. Writing CORBA based systems that perform acceptably with a lot of fine grained data transfer can be quite difficult. Some developers have used CORBA to establish communications between resources, then employed other, more efficient, data transfer protocols.

Another issue arises when dealing with firewalls. Computers that are used to separate a company's network from the rest of the world. Typically firewalls block things that look like programs – such as packets of CORBA data. This is really a problem which relates to a company's network security policy, rather than a CORBA specific issue.

It is also true that the alternative solutions to CORBA, such as Java Remote Method Invocation and Perl scripts are well understood, have been used for a long time and form a major part of many organisation's code base. In addition, CGI scripts allow the serving of data from web-sites, and, for simple tasks, offer an alternative solution to CORBA, but without some of its more elegant aspects. The CORBA standards can be used to smooth out problems of distribution, platform and implementation heterogeneity. Resulting resources can interoperate at a syntactic level, but many bioinformatics resources remain semantically heterogeneous [4, 5]. In essence, this means that concepts can be used in different ways in different resources – Gene can mean one thing in one resource and have a slightly different meaning in another. CORBA IDL will not automatically remove this heterogeneity.

Another protocol which deserves a mention is the Extensible Markup Language (XML).<sup>1</sup> XML is a flexible, but standard way of describing the structure of data which is produced (or consumed by a program). An XML file uses special *tags* to define data elements. So that, for example, an accession number might be specified by the line `<AccNo>P123456</AccNo>`. It is not an accident that this looks rather like HTML, the language used to describe

---

<sup>1</sup>See [www.w3c.org/xml](http://www.w3c.org/xml) for XML standards, technical reports, discussions and publications on XML.

web pages: both have evolved from the same ancestor.

Associated with an XML file is a definition of what tags are allowed, and the relationships between them (so that, for example, a Protein must have an Accession-number and a Sequence, and it might, or might not, have a Bibliographic-Reference). This definition file is known as the Document Type Definition, or DTD. The fact that XML is a standard means that a number of excellent off-the-shelf toolkits exist for reading and writing XML files, and for validating their correctness against a DTD. Another perceived advantage of XML is that their files are written in ascii, making them accessible to human beings as well as computers (although anybody who has tried to read the raw HTML source code of a web page may argue as to how accessible that is).

So, although XML offers a powerful set of tools and standards for exchanging data, it does not provide mechanisms for handling the communication between remote objects, or invoking their methods. This means that it has potential for being part of a wrapping solution, but not a wrapping solution in its own right. It is likely that a combination of XML with other mechanisms for establishing and managing communication between objects (perhaps using CORBA) might provide an interesting alternative to a pure CORBA solution.

Despite some of these reservations, the CORBA standard remains an interesting potential solution to a pervasive problem in bioinformatics. Over the past few years many people have promoted the CORBA standard as a universal panacea in bioinformatics, but like all such claims they should be viewed with some caution. It can provide an 'off the shelf' solution to

problems of distribution. With the possibility of resource owners providing CORBA IDL descriptions and servers for their material, CORBA remains an attractive prospect for the community.

**Acknowledgements:** Robert Stevens is funded by AstraZeneca Pharmaceuticals and the BBSRC/EPSRC Bioinformatics programme. Crispin Miller is funded by the BBSRC and Pfizer through a CASE studentship. We are pleased to acknowledge their support.

## References

- [1] T.K. Attwood and D.J. Parry-Smith. *Introduction to bioinformatics*. Addison Wesley Longman, 1999.
- [2] G. Booch, J. Rumbaugh, and I Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.
- [3] T. Budd. *Understanding Object-Oriented Programming with Java*. Addison Wesley, 1998.
- [4] I.A. Chen and V.M. Markowitz. An Overview of the Object-Protocol Model (OPM) and the OPM Data Management Tools. *Information Systems*, 20(5):393–418, 1995.
- [5] S.B. Davidson, C. Overton, and P. Buneman. Challenges in Integrating Biological Data Sources. *Journal of Computational Biology*, 2(4), 1995.
- [6] E. Gamma, R. Helm, R. Johnson, J. Vlissides, and G Booch. *Design Patterns Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

- [7] M. Grand. *Patterns in Java Volume I*. Wiley, NY, 1998.
- [8] R. Hull and R. King. Semantic Database Modelling: Survey, Applications, and Research Issues. *ACM Computer Surveys*, 19(3):201–260, Sept 1987.

```
class BioSeq {  
  
    // private data and methods  
  
    private char[] sequence;  
    ...  
  
    // public methods  
  
    void setSequence(char s[]) { sequence = s; }  
    void setAccessionNumber(String acno) { ...; }  
    String getAccessionNumber() { ...; }  
    char getResidue(int pos) { ...; } throws IndexOutOfRangeException;  
    int getLength() { return sequence.length; }  
}
```

Figure 1: The Java pseudocode for the class BioSeq.

```

class BLAST extends AlignmentTool {
void setScoringMatrix(ScoringMatrix
sm);
void setEThreshold(float e);
void setPThreshold(float p);
    ResultsSet
doQuery() {...}
    throws operationNotPossibleException;
    ...
}

```

(a) A pseudocode definition of a BLAST class.

```

BLAST searchTool = new BLAST();
Database db = new
Database("Swissprot");

searchTool.setDatabase(db);
searchTool.setQuerySequence(aSequence);
searchTool.setScoringMatrix(blossom62);
ResultsSet answer =
searchTool.doQuery();

```

(b) Pseudocode fragment creating and using a BLAST search tool.

Figure 2: A class definition for a BLAST alignment tool (Figure 2(a)) and a code fragment that creates instances of BLAST and Database and uses them to perform a simple similarity search on the sequence instance aSequence (Figure 2(b)).

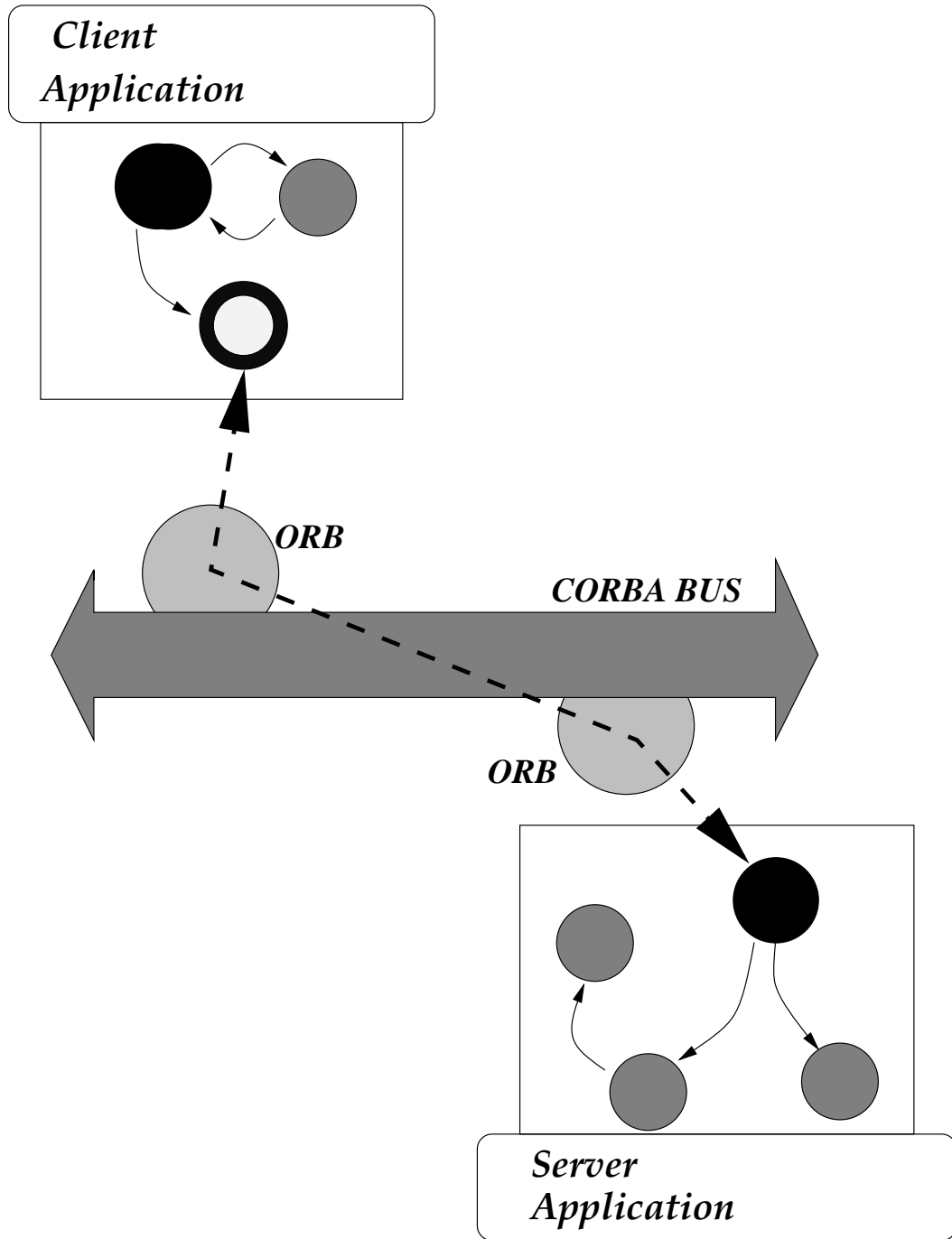


Figure 3: interoperating ORBs embedded in the CORBA bus, allowing requests from client objects to be transmitted to server objects. Note that objects can be both clients and servers.

```

module
Bioinformatics {
interface BioSeq{
    attribute string sequence ;

attribute string accession_number ;
    char get_residue(in int pos)
raises (IndexOutOfRangeException);
    int get_length() ;
};

exception
IndexOutOfRangeException {
...
};

interface NASEq: BioSeq{
    ProteinSeq
translate(in int frame)
    raises (NoSuchFrameException);
};

interface DNASEq : NASEq {
    DNASEq
get_reverse_complement();
};

interface BLAST : AlignmentTool {
attribute
ScoringMatrix set_scoring_matrix ;
attribute float set_p_threshold;
attribute float set_e_threshold;
    ResultsSet do_query()
    raises (OperationNotPossibleException);

...
};
};

```

Figure 4: Some IDL definitions of the interfaces to the bioinformatics classes described in Section 2.1.

Architecture & Roadmap	top level partitioning of the domain into sub units and their relationships.
Bibliographic Services	To promote development and adoption of interfaces and services supporting the use of bibliographic citation information.
Cheminformatics	To drive the development and adoption of flexible and robust CORBA interface specifications for information system components in support of Chemical entities in the context of drug discovery research.
Clinical Trials	To promote the development of standard software interfaces and services supporting Clinical Trials development, execution and data analysis.
Entity Identification	To discuss, collect and analyse requirements for entity identification services in the LSR domain (e.g. a gene identification service).
Gene Expression	To promote development and adoption of interfaces and services supporting microarray gene expression data collection, management, retrieval, and analysis.
Macromolecular Structure	To promote development and adoption of interfaces for the processing and distribution of 3-D structure data of biological macromolecules.
Maps	To define, plan, and develop LSR activities in the sub-domain of biological maps, for example, genetic maps, physical maps, contig maps, transcript maps, radiation hybrid maps and cytogenetic maps.
Object & Data Models	To develop a consistent description of the LSR domain by collecting and analysing LSR object and data models and related information.
Sequence Analysis	To define, plan, and develop LSR activities in the sub-domain of biological sequence analysis.

Table 1: Some of the LSR's activities in the life sciences research domain.