Architecture for Biological Model and Database Networking
by
Orion Aubrey Reblitz-Richardson
Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of Bachelor of Science in Computer Science and Engineering and Master of Engineering in Computer Science at the MASSACHUSETTS INSTITUTE OF TECHNOLOGY
May 2000
© Orion Aubrey Reblitz-Richardson, MM. All rights reserved.
The author hereby grants to MIT permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part.

Author ............
Department of Electrical Engineering and Computer Science May 21, 2000
Certified by ...... Forbes Dewey, Jr.
Professor of Mechanical Engineering and Bioengineering Thesis Supervisor
Accepted by ...... Arthur C. Smith
Chairman, Department Committee on Graduate Students
Architecture for Biological Model and Database Networking

by

Orion Aubrey Reblitz-Richardson

Submitted to the Department of Electrical Engineering and Computer Science
on May 21, 2000, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science in Computer Science and Engineering
and
Master of Engineering in Computer Science

Abstract

Computational biological models are complex in nature and highly specific. The time invested in developing these complex models has been invaluable, but the goal of code reuse between projects and laboratories has been difficult to achieve. By designing an architecture for biological model and database networking we hope to remedy this problem. Through the use of lightweight CORBA interfaces and well-defined XML namespaces, models can be easily added to the system to expose their interfaces to the group and use data generated by other models. Two simplified versions of models from the heart physiome project are used as examples.

Title: Professor of Mechanical Engineering and Bioengineering
Acknowledgments

First and foremost I would like to thank all of the people, past, present and future, who work in the ICMIT group at MIT. Without you I wouldn’t have research to build from or a legacy to leave. Special thanks go to Prof. C. Forbes Dewey for giving me support and advice when needed and trusting in my abilities when I had to think over things myself. To Ngon Dao, thank you for letting me share the vision of your PhD project and move it maybe one step closer to its final goal. To William Chuang, thanks for being there as another Masters student. There are many times when I appreciated having someone else there to push me along and make sure I met the deadlines that had to be made. To Yuan Cheng, thank you for trips to the food trucks and afternoons discussing the stock market.

A special thanks and good luck goes to all of the new students joining our group. This includes Ben Fu, Jennifer Huang, and Jeannette Stephens. You will be the ones carrying on our work. I hope you take it into uncharted territory.

Last, but certainly not least, I would like to thank my family for all of their support while I was at MIT. Although some would say graduation is the culmination of a goal, I think this is just the beginning. This is a stepping stone, a segment of the path, on which I hope to impact the world in a positive way. Who knows where it will lead, but it warms me to know that your support is there whatever I do. I feel like it is important that I call you by name, so although you know who you are, Beko, Gaia, and Jes thank you for being a part of my life and this experience.
## Contents

1 Introduction .................................................. 9

1.1 General communication protocols ............................ 10
1.1.1 TCP/IP .................................................. 10
1.1.2 COM ...................................................... 12
1.1.3 CORBA ................................................. 12
1.2 Vocabularies for biomedical domains ......................... 14
1.2.1 MeSH ................................................... 14
1.2.2 ICD ...................................................... 15
1.3 Standardization on vocabularies and protocols ............... 15
1.3.1 HL7 ....................................................... 16
1.3.2 DICOM ................................................... 16
1.4 Current research .............................................. 17

2 Motivations .................................................. 19

2.1 The target audience ........................................... 19
2.2 Design requirements .......................................... 20
  2.2.1 Use requirements ....................................... 20
  2.2.2 Technical requirements ................................ 21

3 Designs to Meet Requirements .................................. 25

3.1 Discussion of design alternatives ............................ 25
3.1.1 Data sharing paradigms .................................. 26
3.1.2 Language and communication technology ................. 28
3.1.3 Data representation
3.1.4 Model adapter
3.1.5 Requirements that were postponed
3.2 Resultant architecture
3.2.1 Technologies being used
3.2.2 XML Domain Definitions
3.2.3 CORBA Communication Layer
3.2.4 Model Adapter
3.2.5 Model Core
3.2.6 XML Tools
3.2.7 Dependency Tree Navigation
3.2.8 Name Server

4 Prototype
4.0.9 Heart 2D Model
4.0.10 Pulse Model
4.0.11 Data Store
4.1 How to create your own model core
4.1.1 Storing all your state in the state XML string
4.1.2 Implementing the start, step and stop methods

5 Future work
5.1 Publishing of models to the network
5.2 Creation of a toolkit
5.3 Replicate work done in Java in C and C++
5.4 Security
5.5 Multiple connections to a single model
5.6 Improvements to name server

6 Conclusion

A Glossary
List of Figures

1-1 The CORBA communication protocol .................................. 13
3-1 Relinking the model versus just observing value .................. 27
3-2 Values passed directly between models ............................. 28
3-3 Key/value pair formatting of data .................................. 30
3-4 XML formatting of data .............................................. 31
3-5 First interface option ................................................. 34
3-6 Second interface option ............................................... 34
3-7 Model adapter observing the state string ............................. 36
3-8 Components of our architecture .................................... 38
3-9 Example bucket XML string ......................................... 41
3-10 Example filled bucket XML string .................................. 41
3-11 Second example bucket XML string ................................ 41
3-12 Second example filled bucket XML string .......................... 42
3-13 Set value XML and internal state XML unioned to get new state ... 42
3-14 Push and pull techniques for passing values ....................... 44
3-15 Internal differences between push and pull request fulfillment ... 45
3-16 Simplified model core interface .................................... 46
3-17 The dependency tree and user setting values ....................... 48
4-1 Communication flow between the models ............................ 52
4-2 The heart model’s user interface, in expanded and contracted state 53
4-3 How the Heart2DCore, UI, and HeartPanel interrelate ............. 55
4-4 State stored within the pulse model .................................. 56
4-5  State stored within the data store model  . . . . . . . . . . . . . 57

5-1  Referencing the same model in two different networks . . . . . . . . 64
Chapter 1

Introduction

Many biological processes can be modeled through computer simulation. These models are invaluable in understanding many complex phenomenon, including the workings of the human body. These models, though, are complex and highly specialized. A group at the University of California, San Diego headed by Andrew D. McCulloch has developed an extremely advanced finite element model of the heart. The Center for Computational Medicine and Biology at John Hopkins University has observed and modeled the capillary networks embedded in muscle fibers. Another group, this one at The Cleveland Clinic, has analyzed and recorded large amounts of data on normal heart function, which will eventually be entered into a database and made available to future research. All of these projects have invested large amounts of time in tackling different facets of the same problem - that of modeling and analyzing the vasculature and structure of the heart.

Given the complex and complementary nature of these projects, the following questions naturally arise: Can we somehow reuse or combine these models to improve or extend their results? Can we link these models in a meaningful way, so that they become more than the sum of their parts? By integrating them can we create a more accurate simulation and achieve a more in depth understanding of heart function? Through code reuse and sharing of resources, the time to develop complex models would be reduced.

The goal of my thesis was to develop an abstract architecture for inter-model
and inter-database communication. A developer should be able to take an existing model, create an interface to it, and have this model be immediately available to all other users of the system. Other models could then use this model and the results it provides to extend their own functionality. Communication between models would occur over a network such as local Ethernet or the Internet and the interfaces models published would be well defined through an interface definition language such as that provided by CORBA. These interfaces should be standardized, so that two models of the same physical phenomenon, but with different internal workings would have the same interface and be interchangeable for running a given simulation.

1.1 General communication protocols

There are a number of general-purpose communication protocols, which are used by a wide range of software architectures. These are strong candidates for our own architecture, since they have been tested and tuned through extensive use. We will explore three popular protocols, TCP/IP, COM, and CORBA.

1.1.1 TCP/IP

As a communication layer that lies very close to the wire, TCP/IP is one of the layers on which the Internet was built. Most Intranets, networks that are internal to a specific company or organization, are based on TCP/IP as well.

This communication layer sends data from one computer to another by breaking the information up into a set of packets, each sent independently over the network, and recombining them at the destination. Routing from one computer to another is done through switches, routers, and hubs. These mechanical devices connect computers and guarantee that a packet will eventually reach its final destination. In some cases a packet will get lost. If this happens the receiving client can request a re-sending of the original packet.

Computers on this network are uniquely identified by their IP address. These addresses consist of a 32-bit unsigned binary value, usually represented in a dotted
notation such as 18.205.0.1. The maximal address value is 255.255.255.255 and the minimal address value is 0.0.0.0. Some addresses in these ranges are reserved for functions such as virtual private networking and assigning IP addresses to disconnected computers [26]. The IP address reserved for a computer referencing itself when it is disconnected from the network is called the loopback address. The newest version of TCP/IP is IPv6. This extension to the current specification increases the address space available, but leaves the other details unchanged. [26]

Computers on a TCP/IP network are also clustered according to their IP addresses. A set of addresses is assigned to a local area network (LAN), which is a subset of the wide area network, or WAN. For example, a company may be allocated the 18.205.XXX.XXX set of IP addresses. A department within this company is then assigned all of the 18.205.0.XXX addresses to be used for their own lab. This hierarchical allocation of IP addresses simplifies the routing algorithms needed, since one can find a specific IP address by going through the gateways for smaller and smaller sets of addresses that still contain the destination address. Once a packet has gone through all of the gateways it will be picked up by the destination machine over the local network. [26]

TCP/IP is a minimal communication protocol. It provides connectivity between machines, but very little beyond that. But there are advantages to this. A program that communicates directly through TCP/IP can make the data it sends extremely compact, sacrificing very little bandwidth to routing information and other packet overhead. This is why many game developers write their network gaming modules to communicate at the TCP/IP level. For a gamer dialed up over a 28.8k modem, a smart use of bandwidth is critical. A disadvantage to TCP/IP is that a lot of maintenance has to be taken care of on top of the TCP/IP communication layer to provide fault tolerance, object-oriented interfaces for server components, and registration of server availability. All of these things and more are taken care of by a higher-level communication architectures, but with TCP/IP one has to implement them his or herself.
1.1.2 COM

COM, or the Component Object Model, takes the idea of communication and applies an object-oriented style to it. Each object in COM has an interface, which defines the set of values and functions exposed to other COM components. The creation of these interfaces is done through an interface definition language or IDL. Although it is most widely known for its use by Microsoft in Windows and Windows NT, COM was designed in collaboration with Digital Equipment Corporation, DEC, to extend Microsoft’s OLE architecture and have it interoperate with DEC’s ObjectBroker, which was based on CORBA. DEC eventually dropped the project, but Microsoft has been supporting it ever since. [10]

DCOM or Distributed COM is an extension to COM that lets components communicate between different process spaces and machines. This architecture uses an intermediary to broker communications between the two components. The goal is to hide the communication protocols and details of dealing with faults in the network. All of this is transparent. [13]

COM/DCOM has some strong advantages in that it provides extremely fast local communication between components, provides language-independent component reuse, and a level of abstraction that eliminates the actual details of sending information over a faulty network. Unfortunately, though, it also has some disadvantages. Probably the most glaring one is that the architecture has only been fully embraced on Windows and Windows NT platforms. Although it is available for Macintosh and different flavors of Unix, the founder and major supporter of COM/DCOM is Microsoft. Their support of these other platforms is limited and, as we shall discuss, there are other protocols more popular on Unix.

1.1.3 CORBA

CORBA, or Component Object Request Broker Architecture, is an open standard set forth by the Object Management Group, OMG, for component-level language-independent communication. In many ways its functionality resembles DCOM, but
its primary adopters have been the Unix community and back-end systems. Although the specs for CORBA have been around since 1991, it has only recently started to take off. The CORBA architecture lets distributed objects communicate by use of object request brokers, or ORBs. A client object gains access to a server object by using an interface stub located on the client's machine, which in turn uses the client's ORB to contact the server's ORB. The server's ORB then forwards the request onto the server object, where it is processed. See figure 1-1 for details. [12, 11]

![Figure 1-1: The CORBA communication protocol](image)

From our standpoint, one of CORBA's most valuable traits is that it is designed to be language and platform independent. CORBA is a standard for which there are many ORB implementations available. Each of these provides bindings for one or more languages so that CORBA's functionality can be accessed from that language. For example, Visigenic's Visibroker ORB provides bindings for C++, Java, and COBOL. IBM's, on the other hand, has bindings for C, C++, Ada, Java, COM, and COBOL. As one can see with IBM's ORB, these bindings can even be to other component architectures like COM. [18]

The cross-platform and cross-language nature of CORBA is definitely its most
powerful feature. A drawback to CORBA, when compared to models such as COM is its speed. Because CORBA attempts to make the language and location of objects transparent, all communication between components, even on the same machine, must flow through the ORB. With COM these objects communicate directly, thereby eliminating a bottleneck. For our purposes, though, where models are naturally distributed and are implemented with a variety of different languages, CORBA, we'll show, is the most appropriate.

1.2 Vocabularies for biomedical domains

To communicate concepts accurately, one either needs an extremely accurate translation system to convert terms as they are passed from one system to the other or a well-defined vocabulary that is universally accepted. Of these two approaches, the second is by far the most reliable and accurate. Given the daunting number of terms used in medical and biological research and the importance of accurate descriptions, researchers have extensive experience using well-defined grammars and vocabularies to encode meaning.

Two popular examples of medical vocabularies are MeSH and ICD. Both have been around since before the advent of computers, but have become even more useful given the large amounts of data now available electronically.

1.2.1 MeSH

The indexing and searching of articles in a MeSH-indexed database relies on a set of terms or descriptors to categorize an article’s content. MeSH, an acronym for Medical Subject Headings, defines a set of terms for indexing medical books, journals, and other references. The MeSH terminology provides a consistent way to retrieve related information even though articles may use different terminology for the same concept in their text.

MeSH is used primarily as an indexing system for medical databases. One of the more prominent uses of this is in the National Library of Medicines MEDLINE
database. The MEDLINE database contains over 11 million references and abstracts and 400 full text journals. A free interface to the system is provided via the web. [6]

1.2.2 ICD

The ICD, also known as the International Statistical Classification of Diseases and Related Health Problems, provides a standard nomenclature for describing disease and health problems seen throughout the world. This classification system lets medical practitioners correlate instances of disease across different countries, languages, and locations.

The early medical statisticians William Farr (1807-1983) and Jasques Bertillon (1851-1922) did much of the original work on ICD. The acronym ICD stands for one of the original names used for this system, International Classification of Causes of Death. The classification system was formalized in 1893, its function at that time being the description of causes of death in a structured manner. Over the years the system has been expanded to include non-fatal diseases and other health related issues. [8]

In this age of computerization and automation the ICD is being used more than ever. The current revision of the ICD, ICD-10, is available in book and electronic form. The electronic version is extremely popular, letting clinicians correlate their own data with the ICD automatically. [4, 5]

1.3 Standardization on vocabularies and protocols

In some situations it is prudent to adopt a standard for both vocabulary and protocol. Standardizing the communication medium and media is important when one wants to have many components talking with each other. In this case one can add a new component that uses the standard vocabulary and protocol and be assured that it will communicate with the group.

Some examples of this approach being widely used in the medical field include HL7 and DICOM. HL7 is used for sharing patient and hospital data, while DICOM
is a standard for transmitting medical images. We will discuss both of these below.

1.3.1 HL7

Health Level 7, or HL7, has been around since 1987 and is used by medical institutions such as hospitals to transmit clinical, financial, and administrative information internally and between independent corporations. Insurance companies, hospitals and research labs are only some of the places that transmit information in HL7 form.

HL7 defines the grammar used to transmit data and the communication protocol on which it is transmitted. A standards group agrees upon revisions to these standards and then publishes them to the medical community. Implementations of the standard come from a number of vendors and can implement just the communication protocol or both the protocol and grammar. [3]

There are task forces currently working on integrating different transmission protocols such as CORBA and DCOM into HL7. This would eliminate the communication standard, but let the grammar be even more widely used. HL7 version 3, in the early stages of adoption, has added an object-oriented approach to transmitting and storing patient data. Both of these improvements have the goal of adding power and extensibility to the HL7 architecture.

1.3.2 DICOM

The American College of Radiology, ACR, and National Electrical Manufacturers Association, NEMA, developed DICOM as a standard for the distribution of radiological images and other forms of medical image data. [7] It is widely used to transfer images from test equipment such as MRI machines, data storage, and image analysis programs.

The DICOM standard specifies the kinds of services that can send and receive DICOM data, what form messages sent between these entities should have, and what an acceptable lower-level network protocol is to pass these messages. This definition specifies a paradigm for client-server communication between medical imaging hard-
ware and software from multiple vendors. It also defines a large dictionary of tags that one can apply to images as meta-data. Therefore it specifies the communication layer and the data being communicated. [2, 1]

1.4 Current research

The creation and use of computational models and storage of experimental data have seen a steady increase in the biological sciences. Through the heart physiome project, our group has strong ties to institutions working on this kind of storage and modeling.

The International Consortium for Medical Imaging Technology (ICMIT) has been working on the Heart Physiome project for less than two years now. Researchers from UCSD, John Hopkins, Cleveland Clinic, University of Washington, and MIT are working to make this integration of projects a reality. Their goal, which is a test-bed for these theories, is to develop a physiome of the human heart. A physiome is a "quantitative description of the physiological dynamics or functions of the intact organism". This comes from the roots "physio-" (life) and "-ome" (as a whole). Therefore, the challenge is to create a model that takes into account all of the aspects of the whole and living heart. The ultimate goal is to develop a complete physiome of the human body, but projects such as the heart physiome are needed before this larger problem can be tackled.
Chapter 2

Motivations

2.1 The target audience

Motivation for this project stems from issues encountered in the heart physiome project and while observing other computational biology work. In particular, Andrew McCulloch's finite element model and network model integration sparked our interest. [16] Also, work being done in our group on database design for storing medical data prompted a need for some standard way to reference the data. [17] The model and database architecture we've developed is ambitious in that it tries to solve both of these problems by creating a standard way to publish data from a model or a database.

Our target audience is researchers that have developed simulations of biological systems and want to share these simulations with colleagues or integrate them with other models in their own lab. Although our current focus has been limited to simulations of the human body, the same architecture could be used for connecting any form of computational or mathematical modeling. Population simulations, weather forecasting, and protein interactions are just some of the fields that use complex computational models and would benefit greatly from sharing research across institutions.
2.2 Design requirements

Given the target audience, there are a number of design requirements that must be satisfied. These design requirements fall into two rough categories: Those requirements that reflect the way in which the architecture is going to be used and those requirements that are needed from a technical standpoint for the sake of maintenance, extensibility and reasonable implementation time.

2.2.1 Use requirements

Those requirements that are specified by the model builders and users of the architecture fall under this category. Researchers in the field of biology and medicine, our target audience, generally do programming as a means to an end and have only limited familiarity with sophisticated computer science technologies such as CORBA and XML. Their goal is simply to get their simulations and results as quick as possible. We need to take this into account when creating the architecture.

After talking to a number of researchers and noting their common requests, we narrowed them down to the following list of use requirements:

1. Publishing a pre-existing model through the network must be as painless as possible. It should hopefully require only a minimum of additional coding. People using it are not going to be programmers by profession.

2. Not only must it be easy to publish a model to the network, but it must also be easy to incorporate results from other models in one’s own calculations.

3. Many programming languages are used, so we have to accommodate them. Popular languages include Matlab, Fortran, C, C++, and Java. Proprietary languages associated with specific modeling packages are sometimes used.

4. The emphasis is on connecting models. We need some architecture by which a model done in Matlab can use a model developed in Fortran with a minimum of effort. This is the motivation for our architecture and also the ultimate use case.
Countless man years have been put into developing computational models of biological systems. These systems, however, are disjoint and only accessible by a limited few. Our goal is to make these models available to a much wider audience of researchers and pave the way for models being combined to create simulations of even more complex systems. The heart physiome project and the researchers in this group are our primary focus for the time being, but our hope is to create an architecture general enough that it will contribute to a much wider range of research.

2.2.2 Technical requirements

Technical requirements are those things that must be implemented in order to meet the above use requirements or that are needed from a realistic programming or future-improvements standpoint. Some of the points found to be important include:

1. The exact domain in which this architecture will be used will vary. The architecture needs to be general enough to support a variety of grammars and information being passed between models.

2. To limit the invasiveness of our architecture, we want to minimize the amount of old code the user must modify and new code needed to be written to publish the model through our network. This can be done by providing mostly pre-built adapters and requiring that they only write the glue code which attaches this adapter to their pre-existing code base.

3. We know that things are going to evolve. It's possible that in the future CORBA will become obsolete or XML will be supplanted by a better standard. Changes in the technology used and minor improvements or the architecture are bound to happen, but if we prepare for it these changes can be graceful.

4. Other general requirements that are relevant to any programming architecture including speed, reliability, fault tolerance, and security.

As requirement one states, we don't know what grammars and vocabularies models will be using. These things are completely domain specific and can vary greatly even
for models in a very tightly focused domain such as the human heart and circulatory system. For example, there's no way to know what exact data a researcher would need to pass back and forth between a surface model of the heart and a model of electrical impulses along that surface, but we do know that those values will most likely be different from those shared between the surface model and a fluid dynamics model of blood flow through the heart. These models all exist in the same domain and yet the data they provide and transact was completely different. Therefore the values published by a model should not be restricted in any way by our base architecture. It should only be the way in which these values are published that is enforced.

Our second requirement is to make it easy for users to publish current models. Only requiring a small amount of modifications to existing code and the addition of a minimal amount of glue code are imperative if researchers are to embrace the architecture. This is a reasonable requirement given that most of the adapter code can be standardized and provided in a library. A user would implement some “core” functionality and leave the rest of the code, which would be common across many models, in an auxiliary class.

The third requirement is adaptability. Improvements and changes to the architecture will inevitably occur. The ease of extension and modification are critical. To address this one needs to look at how the architecture might evolve over time and make the architecture extensible in these directions. It is impossible to make an architecture that is extensible in all ways, but anticipating what kinds of functionality will be needed in the future makes it more feasible. In our architecture we know that people will need to use a variety of vocabularies and grammars and that the interfaces we’ve defined for models might need to be improved. Also, we know that technologies such as CORBA, XML, and Java are not permanent and will most likely be replaced by better options over time. At the same time as we allow for extensibility, we need to make sure the core implementation that a researcher writes works even if the architecture is modified and improved. The API that they write to must be standard or additions to it must be backwards compatible. All of these considerations need to be taken into account when designing the system.
Lastly, we have the general requirements of speed, reliability, fault tolerance, and security. The time taken for a two models to communicate and the running time of a model when using the network versus not using it are two places where speed is a factor. Communication between models must be fast enough to not adversely affect the running of any given model that relies on other models via our network. Real-time communication is desirable, but not necessary, given that most models in the biological domain cannot compute their results in real-time. Given the rate at which computational power increases, though, following Moore's Law, this will be an issue in the near future. Regardless of computational speed, communications should never be the bottleneck of a model’s operation. Reliability is needed when a model needs to find another model on the network. Fault tolerance comes in the form of gracefully handling error situations and making sure the system doesn’t get into an unplanned state. For our system in particular, there are a number of areas that should have special attention when it comes to fault tolerance. A model’s use of other models must be robust enough that it can tolerate bad data or no data at all. If it expects to receive certain values and does not get them, it should gracefully fail and alert the user to which model is failing and what the value requested was. Also, the finding of models on the network and, more importantly, not finding them, must be fault tolerant. If a model depends on one that is not available then the user should again be alerted with relevant details. Security, our last general requirement, is also of prime importance. The data sent between models can be of a very sensitive nature. For example, if a researcher is working with clinical data from a patient then leaking that data would put the patient’s medical privacy at risk. Since data has to travel over insecure networks such as the Internet, some form of encryption will be required. To prevent the spoofing of services and collection of illegal data that way, one also needs an authentication system.
Chapter 3

Designs to Meet Requirements

Given all of the design requirements listed in the previous chapter and the current tools available to us from computer science, we decided to develop a distributed architecture by which models could be “published” for use by other models and whereby pre-existing models could be augmented to take new data into account. To reiterate that separation of goals, we decided to:

1. Create a simple way by which models could publish themselves over the network for use by other models

2. Make it relatively straightforward for model builders to then incorporate these new data feeds into their next version of their models

There were a number of sub-requirements that arose from these primary goals and the unique user base for which we are designing. By looking at our design requirements in more detail and weighing all of these priorities according to their importance, we arrived at a solution that fit within these bounds.

3.1 Discussion of design alternatives

Most of my research time has focused on the first of those two goals, that of publishing a model to other researchers. Our focus on this goal was a matter of necessity more
than anything, since it is a prerequisite for the second goal. The project scope is large enough that focus on just one of these goals can define a whole thesis, but some explorative work has also been done on the second point.

These requirements solidified into a number of design decisions between approaches to achieving the same goal. We found that tradeoffs often had to be made between satisfying one requirement over another and that the choice was never simple.

3.1.1 Data sharing paradigms

One of the first decisions we had to make was how information should be shared between models. There were two fundamentally different ways to go about it:

1. Have a central repository from which models got their values and to which they would write back their changes. This meant that there would be a shared memory space in which all of the services in a model network would operate.

2. Connect models directly, having a model pass values to other models interested in the values they produce. There would be no shared memory space, only a distributed flow of data.

Having a centralized database and name-space has many advantages. First and foremost is that a model can retrieve values from the database without worrying about what model generated or modified the values. Compare this to the model having to know exactly which model is generating the data and querying that model directly. In this case the model must know where the value comes from and also request the value itself. Changing which model is used to calculate a given value would require that every model in your network be unlinked from the old model and linked up to the new one. See figure 3-1 for a graphical explanation.

The biggest disadvantage with this system, though, is that the database or state-sharing server becomes a bottleneck to scalability. As models are added, each one of them will have to query the database service to get its inputs and write its outputs.
All communication between the models is brokered by the database server. Each model sets values in the memory, which will later be queried by other models. To eliminate this bottleneck we must consider the second option, which is to have models communicate directly in a distributed network.

The advantage to this second approach is elimination of the middle man. See figure 3-2 for details. Unfortunately, with this you also lose some things. As was described before, one cannot simply swap out models and replace them with another model that provides the same function. All of the references from dependent models have to be redirected to the new one. Also, there is no longer one place where all of the current state of the system can be found. Instead it is distributed throughout the whole network. In order to observe these values one needs to inspect the output of all the models in the network. Generally, though, we'll only be interested in the results of one model, the root model, as we call it, which will provide output to the user. This might be shown in a graphical form, such as a surface rendering, or data might be written to a file. The important thing is that the user will only see the output of this model and will not even necessarily be aware that there are models below that root model helping to generate the output. See figure 3-2 for details. Our architecture uses direct communication between models to pass values.
3.1.2 Language and communication technology

Once we'd decided what general paradigm the architecture should take, questions that needed to be addressed included “What communication protocol should be used?”, “What standards should be set for data being sent across that network?”, and “What should the interfaces of the models be?”. Should we specify our own communication protocol and packet structure like HL7 or should we use a standard layer like TCP/IP or CORBA? Should we enforce a specific data packet format or should models just somehow define between themselves some format that they will be using? What interfaces would models need to implement to provide all this functionality? These are the questions that had to be answered.

We started by exploring communication protocols, putting off the decisions about data representation and interface specification until later. Possible candidates for the communication level included CORBA, COM/DCOM, and Java RMI. Even a CGI-based interface and raw TCP/IP packeting solution were explored. The CGI-based and raw TCP/IP communication schemes were eliminated early on, since they were not as full featured as the other protocols. Although for some applications this
can be a benefit (faster, more streamlined connections if one doesn’t need the extra functionality) in our case almost all of the features provided by protocols such as CORBA and COM/DCOM were needed. The creation of standard interfaces and implementing these in a wide variety of languages would be needed. Connections between models would have to be monitored and any packet loss resent. Communications over a local area network or over the Internet would have to be handled transparently. All of these features are provided by the higher level protocols, but would have to be implemented from scratch if we used CGI or raw TCP/IP. Therefore, these options were out.

One factor that made a strong impact on our final choice of communication protocols was the wide range of programming languages that would need to be supported for coding of biological models. In a survey of current models we had already seen C, C++, Java, and FORTRAN being used extensively. Proprietary languages such as Matlab and those used for 3D modeling packages where also popular. Whatever architecture we decided on, there would have to be access to it from all of these languages.

The communication architecture that turned out to be the most suitable for our needs was CORBA. Java RMI only supports Java connectivity, so this option was eliminated. COM/DCOM, although a standard across multiple platforms, is primarily supported and endorsed by Microsoft and therefore only widely used on Windows machines. Also, supporting a wide number of languages was not one of the primary goals of COM. Java, C++, C, and Visual Basic all have hooks into this architecture, but we couldn’t find support for more academic languages like Ada and Fortran. CORBA, on the other hand, was designed to be language independent and is popular on and off Windows platforms. Since most of our own servers run some flavor of UNIX (Compaq True64, Solaris, HP/UX, and Linux, among others) this was an important factor in my testing the architecture. Other academic researchers use a variety of platforms, often using Windows machines for simple computations, but moving to higher end hardware such as that provided by SGI and Sun for complex simulations. We wanted to make support for all of these platforms possible in the future, even if
not immediately available. The OMG’s commitment to making CORBA available for multiple platforms and languages coincides with our goals.

3.1.3 Data representation

To address the question “What standards should be set for data being sent across the network?”, we needed to decide how much extra information would have to be passed between models above and beyond the actual values being shared. Then we needed to determine how the data would be represented.

Some descriptive information needs to be sent either before or along with any data communicated between two models. This descriptive “meta-data” gives the receiving model a context in which to understand the data themselves. Just like two people have to agree on a language before communicating, models need to agree on the kinds of data that can be communicated and understood before communication can take place.

One way to represent data and meta-data together is to use key/value pairs to represent each value. In this case the meta-data, or type of information being sent, is the key and on the left hand side of the equals sign below, and the data, or values of that information, are on the right. Using this encoding scheme, the description of an apple could be represented as shown in figure 3-3.

```plaintext
fruit_type=apple
fruit_color=green
fruit_size=medium
fruit_hasseeds=true
```

Figure 3-3: Key/value pair formatting of data

This style of associating values with variables has a long tradition in the UNIX community as a way to store configuration information. Each program has a file associated with it that contains key/value pairs that define preferences set by the user. The keys are special in that each program has a different set that it understands. Although this style has been popular in the past, a new format for storing
and transmitting meta-data has the potential to start a revolution. XML, or eXtensible Markup Language, is a scaled down version of SGML, a complex markup language used in academic circles, but rarely seen anywhere else. The almost instantaneous popularity of the web and use of HTML has laid the ground work for XML. As with HTML, meta-data in XML is stored in the form of tags, which wrap around data and store semantic meaning. As an example, an XML encoding of the information represented in key/value form above could look something like that shown in figure 3-4.

```xml
<?xml version='1.0' ?>
<fruit type='apple'>
  <color>green</color>
  <size>medium</size>
  <has_seeds>true</has_seeds>
</fruit>
```

Figure 3-4: XML formatting of data

XML has a number of advantages over other meta-data encoding schemes. These all stem from it being a well-defined standard with many tools available for its manipulation. Issues that one commonly has to deal with when creating unique markup systems are already taken care of in XML. For example, a common problem when creating a grammar is how to have a set of control characters, but at the same time be able to transmit these characters as data? For example, in the key/value example we showed above, the = sign was a special operator that separated the variable name from the value. What would we do if the value contained equal signs? If we just inserted them without any pre-processing, a parser reading that information would misinterpret those equal signs as separators between new keys and values. To get around this problem, one can escape the \= character with another special character, most commonly the backslash character, \. Therefore, when one wants to have an = show up in the value, you would instead insert \=. This adds another level of complexity in that you also have to escape the escape character itself if it is ever needed as a value. So, to represent a backslash in the value section we would have to use \\

31
As one can see this gets complex very quickly. The problem with building your own markup language is that you have to define these conversions and then write a parser that will understand your markup language and convert it into a form a programmer can use. XML, as a widely used standard, already has a number of commercial and open-source parsers available. These programs eliminate the hassle of parsing the files by hand and can provide a number of different outputs. DOM parsers, which use the Document Object Model components, create a tree representation of an XML file, where each node represents a tag or text element in the original document. A SAX parser uses the Simple API for XML standard to generate events when it encounters certain tags in a document, which a program can then listen for. If you want to check if a certain value is present in an XML document, a SAX parser can alert you when it occurs. [31, 32]

Another feature that XML provides is that of XML Domain Type Definitions or DTDs. A DTD defines a shared grammar that constrains the contents of an XML file to adhere to a specific domain. These DTDs can be generated and published as needed by anybody who wants a language to describe their domain of descriptions. Definitions are published over the Internet or other medium and referenced in XML files to state that data contained within is specific to that domain. This supports our goals perfectly, because researchers in a given field could collaborate on specifying a DTD and then reference it when transmitting data between models, thereby standardizing the language by which they all communicate.

As a last argument for sending meta-data with the data instead of separating these two entities, it is worth it, since generally bandwidth is not our problem. Data will typically consist of small packets being passed back and forth between models and if it is otherwise, the XML meta-data wrapping will consist of a vanishingly small percentage of the total information being transmitted across the wire. On the other hand, XML is useful for the exact reason that the context of data is sent with the data itself. This lets the receiving model know exactly what it is getting and lets it focus on the data it needs, while ignoring information that it can’t understand.

With support from industry and strong standards set by organizations such as the
World Wide Web Consortium, XML is positioned to make a huge impact on the way programs store and transmit meta-data. Our goal is to leverage the power of XML to simplify the process of publishing models.

3.1.4 Model adapter

One also needs to determine what the interface for a model should be and what code should be written by us to support that interface. All of this is contained within the adapter, which wraps around the core implementation created by the researcher. The portion implemented by the researcher will be called the model core.

Two different interface styles were considered. One puts most of the complexity into the interface itself, while the other pushes most of the complexity into a complex set of parameters passed into a simpler interface. These two alternatives will be described in more detail.

Using a complex interface would mean that all of the values accessible from a model will have methods for getting and setting these specific values. This is how the implementation for most object-oriented classes is done. For example, let's say we use the apple example from before and now create an interface to an object which represents a fruit. A class representing this would have a functional definition as shown in figure 3-5.

For the sake of simplicity we've made it so all the “get” methods return values in a string form and all methods which “set” values take in a string. Although in most implementations these methods would return different data types, such as a boolean for the getHasSeeds() method, we want to standardize on only string values being returned, so that a comparison can be made between this interface and the one described below, where all of the values are get and set through the same set of methods.

Our second option is to have a fewer number of methods in the interface, but to pass more parameters to these methods. For example, you can see in the above code that each value contained within the Fruit class has two methods associated with it - a get and a set method. One way to simplify this interface is to consolidate these
class Fruit {
    String getType();
    setType(String type);

    String getColor();
    setColor(String color);

    String getSize();
    setSize(String size);

    String getHasSeeds();
    setHasSeeds(String hasSeeds);
}

Figure 3-5: First interface option

methods so we only have one get method and one set method, but each takes in a parameter which says what value you want to get or set. The definition for this class would look like the code shown in figure 3-6.

class Fruit2 {
    String getValue(String valueName);
    setValue(String valueName, String value);
}

Figure 3-6: Second interface option

Now, the question is, why would one want to use the second interface over the first one? The answer is, with the second interface, the interface doesn’t change, regardless of what values are stored in the object itself. Therefore, the same interface can be used for all kinds of objects, not just fruits. This is exactly what we need with our model interfaces, because, with CORBA you need to have a copy of the interface of any CORBA server you ever want to talk to on the client machine. If model interfaces changed for each model published to our network we would need to have some way to distribute these interfaces to the clients that wanted to use them. With this second method, though, there is a single interface used by all models and a client can simply
use this to talk to any of them. Therefore our architecture uses interfaces very similar to that listed in interface 3-6.

Keep in mind, though, that this does not mean that a model a client talks to will necessarily have the values it wants. When you pass in a valueName to the getValue or setValue methods that specific value might not be published by the model. In that case it will need to return an error message or some other indicator that the value was not found.

This section gave a general overview of two approaches to building a model interface. We used scaled down versions of the problem we worked with for the sake of simplicity, but in the explanation of our resultant architecture we will discuss the real model interface and methods of that interface in great detail.

Now that we’ve determined what kind of interface should be implemented, we need to look at the code behind this interface. What portions should be implemented generally and how much should the model architect implement? Is there some API we should give the researcher that would provide enough functionality, but hide most of the complexity involved in using CORBA and our architecture? We have tried a tentative solution to these questions, but have started to see that it might need to be expanded in some ways. The current architecture provides a model adapter, which consists of three components: The Model, ModelCoreWrapper, and ModelCore. These takes care of registering a model on the network, answering requests for values, and setting values in the model itself. An XML string is stored in the ModelCore as the current state of the model. This is then monitored by the ModelCoreWrapper, which answers requests for values. See figure 3-7 for details.

This approach is really quit similar to the shared memory space approach discussed previously. Instead of having the values shared universally, though, a model has its own small set of variables that it manipulates. These values then are passed to other models when they change or are explicitly requested.

Standardizing the adapter for all models is a very difficult problem, because we don’t know what kinds of models will eventually arise. Our tentative solution could make it a lot easier for researchers to publish their models, but even if a model
cannot be tied into this approach, one could still publish it by implementing the model interface from scratch. Our hope is to provide some tools that will simplify this process whatever the model being published might be.

### 3.1.5 Requirements that were postponed

Given that this was a first revision of the architecture and a proof of concept, we made the choice of dispensing with some of the design requirements we had originally hoped to fulfill. These deferred requirements included:

1. Reliability and fault tolerance concerns

2. Security issues

Reliability and fault tolerance, as was discussed in Chapter 2, are important at a number of points. Fault tolerance as far as getting incorrect data returned from a model has been partially achieved by using XML. Since XML encodes meta-data that describes the data being passed, a receiving model can ignore extraneous information which it did not request. It would still have to throw an error, though, if the value it wanted was not returned. Also, XML DTDs can be used to constrain the values a given bit of data can take on. If data violates the DTD the user can also be alerted.
Reliable registration and lookup of models has also been partially implemented by the name server. The name server lets one have a unique name for a service that does not change between stops and restarts of that server. The standard way of connecting to CORBA objects is to use an “IOR reference”. A new IOR reference is given out between stops and restarts of a model. By querying the name server a client model can (a) determine if the model it is requesting is available and (b) determine the IOR through which it should communicate. If the client cannot find a reference to the model it alerts the user. Unfortunately, though, we have had some difficulties with unregistering models from the name server, which then prohibits the model from re-registering itself when it is restarted. This is why this requirement has been listed under postponed requirements. A more thorough solution should be done in the future.

We decided to not do any work on security for this version of the code, but we’ve designed the architecture so that security is something that can be easily added at a later date. It would consist of developing an authentication scheme for verifying and restricting connections made between models and finding some form of encryption by which all data being transmitted could be only readable by the client and server. The way in which these two goals are meet, though, are varied. It is possible that CORBA-level encryption could be used, or it might be necessary for us to do our own. Shared public keys or public/private keys could be used to do authentication. The amount of time needed to research and implement these features, though, are beyond the scope of this thesis.

3.2 Resultant architecture

Our goal has been to provide a way in which researchers can publish their models with a minimum of effort and then incorporate values from other models in their own analysis. To do this, a standard interface for models is defined and partially implemented by the adapter. The joining of this adapter to the researcher’s model is the only step needing to be done by the model builder. We’ve provided a potential
abstraction layer with the ModelWrapper and ModelCore classes, which does most of the common tasks and lets the researcher focus on creating a core implementation.

Although the system works together as a whole, the architecture can be broken down into a number of key sections. Each one will be discussed in detail. See figure 3-8 for a picture of the complete system.

![Components of our architecture](image)

**Figure 3-8: Components of our architecture**

### 3.2.1 Technologies being used

**CORBA**

CORBA is used as the distributed object model for our architecture. After much research, we decided to use Object Oriented Concept’s Object Request Broker, or ORB, implementation to provide CORBA connectivity. OOC’s ORBacus 3.2.1 provides hooks for Java, C, and C++, which was the bare minimum of the languages we wanted to support. It provides helper classes in these languages for implementing
a given CORBA IDL interface and registering a component with the ORB. An important reason for our selecting ORBacus was that it is free for non-commercial use and very reasonably priced if needed commercially. Initially our architecture would only be used in academic circles, so it would let us distribute the ORB to other users free of charge. OOC even lets users download the source code from their website and compile it themselves. We've had absolutely no problems running it on Windows NT, Linux, Solaris, and Compaq True64 UNIX. Any other platforms that people might be interested would probably also work. [27, 29]

**XML**

The XML parsers we are using are released as open-source by the Apache XML Project. Xerces is an open source package and contains implementations of the DOM and SAX parser definitions set out by the W3C. [31] The parsers are available in Java, C++, or Perl form. They were originally written by IBM and Lotus, but eventually donated to the Apache Foundation and the open source community. The Xerces parsers are some of the most complete free XML parsers available and their multiple language support is appealing, since we will eventually be extending our architecture to these languages as well. Being able to use the same XML and CORBA tools in all of these languages is a big plus.

We're also using the Xalan package from Apache's XML Project for doing XPath searches through XML documents. [30] XPath is a standard defined by the W3C that lets one use a regular-expression-like language to look for specific tag sequences in a file. Our XML utilities file wraps around Xalan to provide a very simple way to extract information from a given XML file. Both the Xerces and Xalan packages have been some of the best XML tools I have ever used. Since they are open source, they are free to distribute and are constantly improving. They have both been invaluable for XML manipulation.
Java

Java has been picked as our language of choice because of its portability and rapid development time. Of course, as stated before, the goal is to replicate the adapter code in other languages such as C and C++. Java, though, is a good choice for the first version, because one can produce precise, easy to understand code that is cross platform. The integration between CORBA and Java is also more highly developed than that between C++ and CORBA. A version written in Java can easily be understood and ported to other languages.

3.2.2 XML Domain Definitions

*Code for this section includes XML.java and XMLBucketFiller.java*

All the values passed to and from models take the form of XML. This packaging ensures that clients know what information they are receiving and lets them extract only those values that they require.

XML domain definitions can be written for any data that a model wants to transmit. Work is currently being done on creating a DTD for the heart physiome. In the process of creating it we've realized that a number of orthogonal domains might need to be used in order to describe all the information that needs to be represented for the heart physiome. For example, data describing the surface of the heart is very different from that describing fluid flow through the heart. One domain could consist of a set of 3D coordinates, while the other might contain pressure readings and fluid speeds. Although they could be combined under the same DTD shell and segmented into two different subtrees, it doesn't seem as clean as we had originally hoped. Future work will show whether higher specialization of the DTDs will be needed. If so, each would serve a small subset of the models in a network simulating the heart.

The models use a “bucket XML string” to query other models for values. This is an XML string that uses the same DTD and tag structure as the result, but that is only partially filled with values. It is a query syntax that can be viewed as a bucket
that is then filled by the server and returned. An example is shown in figure 3-9.

```xml
<?xml version='1.0'?>
<fruit type='">
</fruit>
```

Figure 3-9: Example bucket XML string

This bucket says that the client wants information on all types of fruit, including the type of fruit and all of the information below each fruit tag. If a server was publishing the fruit data as described in Section 3.1.3, a query sent to it with that bucket would return the values shown in figure 3-10.

```xml
<?xml version='1.0'?>
<fruit type='apple'>
  <color>green</color>
  <size>medium</size>
  <has_seeds>true</has_seeds>
</fruit>
```

Figure 3-10: Example filled bucket XML string

If the client wanted the color of the fruit and nothing else they would use the following bucket to ask for only a limited subset of the information as shown in figure 3-11.

```xml
<?xml version='1.0'?>
<fruit>
  <color/>
</fruit>
```

Figure 3-11: Second example bucket XML string

The server would then return the information in figure 3-12.

For setting values a similar technique is used so only a subset of a model’s state is altered. All of the values passed in that overlap with the model’s current state are
updated with the new values. All values that do not overlap are left unchanged. A pictorial representation of the process can be seen in figure 3-13.

As one can see this process is identical to retrieving values in that the bucket and state are overlapped and the subset that coincides is modified. In fact, this is even how it is implemented. In the get method the bucket is filled with values from the internal state, while in the set method the internal state string is filled with values from the bucket. This is an elegant approach to providing both functions.

Values can be set by other models, the user, or from text files. Model values are set at initialization of the model network, when the client model requests an update of a value, and sometimes when a value in the server model has changed. The user enters values through a user interface displayed to him at initialization. The values entered are propagated down and set in each of the models in the dependency network. All of the values set take the form of a filled bucket, each modifying the model’s internal state.

This technique of using buckets to retrieve and set subsets of the data is tightly integrated with the idea of using XML domain definitions, because one cannot only
specify the values requested, but also what DTD these values come from. Therefore
the request and response syntax are both directly defined by the DTD set up by a
given set of researchers to describe their domain. This architecture should be general
enough that it will have applications in many different contexts.

3.2.3 CORBA Communication Layer

*Code for this section includes all of those needed for ORBacus and ModelConnector.java*

CORBA was used for the ease with which programmers can develop objects that
communicate over the network and because of the wealth of added features that come
integrated into the system. CORBA has the ability to make a component accessed
over the Internet look like one on your private network, and even look like one on
your local machine. This network transparency is a corner stone of how we want to
let models communicate regardless of their physical location.

CORBA provides only a minimal communication layer through which complex
XML is sent. For the most part, the user shouldn’t even know that CORBA is being
used. Since most of the code needed to communicate via CORBA is taken care of by
the model adapter, a communication layer other than CORBA could be used instead.

3.2.4 Model Adapter

*Code for this section includes automatically generated files created by ORBacus’ IDL
to Java converter, ModelII.java, Model.java, ModelWrapper.java, and ModelCore.java*

The model adapter is the code that publishes a model to the network. It takes
care of most of the communication details for CORBA and forwards requests for data
to the core model implementation. All models use the same CORBA interface, but
can provide different values through it. The interface provides methods for getting a
list of values the model provides, getting a list of the inputs to a model, and getting
and setting values of the model. In addition, it provides methods for starting, pausing and stopping the model's execution.

Getting and setting values can be done in one of two ways: pulled, where the value is requested from a model when it is needed; or pushed, where a value is set in the requesting model when it has changed. Refer to 3-14 for an illustration of the differences between these two techniques.

![Diagram of push and pull techniques for passing values](image)

**Pull Method**

Request → Immediate, One-time Response

Model One

Model Two

**VS**

**Push Method**

Request → Delayed, Periodic Response

Model One

Model Two

Figure 3-14: Push and pull techniques for passing values

The pull method of value propagation is definitely the most common. When a model needs a value it simply queries another model for it and it is returned. The push method, though, is needed when a model needs to be driven by some input. An example in our heart physiome project was a model of heart muscles being driven by a model of electrical pulses along the surface. Pulses are sent to the muscle model, which in turn contracts. This way the lower model drives the upper model instead of the upper model forcing the lower model to compute a value.

As can be seen in figure 3-15, the CORBA interface is only the surface of the adapter. This interface is implemented by Model, which contains an instance of ModelCoreWrapper. ModelCoreWrapper spawns a thread in which the ModelCore itself is run. The ModelCore's start method is called to initialize the model, then the
step method is called repeatedly to have it do its calculations. When the model is shutting down the stop method will then be called. The state internal to the model core is observed by the Model and ModelCoreWrapper. Whenever a value is requested it is pulled from this state string. If a push request, or asynchronous request as it is sometimes called, has been made, then the model making the request is listed in the asynchronous request lookup table and the state is checked at every step to see if it has changed and, if so, whether the registered models should be alerted. In the case of the pulse generator this technique is used to alert the heart model at periodic intervals that the 2D surface should be contracted.

3.2.5 Model Core

*Code for this section includes ModelCore.java and TestMCore.java, an example model core*

Any implementation of a model core must implement the ModelCore interface. A simplified version of the interface is shown in figure 3-16.

The real ModelCore interface has a few extra methods, but the general idea of this interface is for the model core to be started with the `start()` method, run
public interface ModelCore {
    public String getState();
    public void setState(String s);
    public void start();
    public void step();
    public void stop();
}

Figure 3-16: Simplified model core interface

...with repeated calls to the `step()` method, stopped with the `stop()` method, and interacted with by getting and setting values from the model's XML state. All other chores, such as dealing with push and pull requests for values and setting of values from other models is taken care of by the model adapter.

TestMCore.java is provided as a simple reference for how a model core could be implemented. Like all model cores, this class extends `edu.mit.physiome.architecture.ModelCore`. It simply ticks off cycles and displays them on standard out and answers requests for a subset of the state string, which doesn't change. It is simply an example of how to implement the ModelCore interface and provides very minimal functionality.

These classes provide a tool for simplifying the task of publishing to our architecture. It is possible, though, that some people might have to implement the Model class themselves. It is possible that the start, step, stop paradigm will not work for some models. Anything that needs to fork off a thread or run a command-line process might have problems with this. We haven't used it enough to be sure if this is the case or not, but more testing should reveal any weaknesses.

In a future revision of the system we hope to improve the interfaces and classes so that they simplify the task of publishing a model to the network and support a wider range of models than our current start, step, stop paradigm.
3.2.6 XML Tools

*Code for this section includes XMLBucketFiller.java and XML.java*

We’ve provided a set of Java classes for manipulating XML data. They are used by the model adapter classes and by the model cores to access and modify their state strings.

XMLBucketFiller.java, as its name implies, provides methods for manipulating and filling XML buckets. A bucket and source XML string or document are passed into its method and the bucket is modified to contain values from the source string. Using the DOMWriter class, one can then generate a string that is the filled version of the XML bucket.

XML.java provides methods for searching XML documents. These methods wrap around the Xalan XPath methods and are designed to mimic the methods provided by Microsoft’s Visual Basic MSXML objects. A search is done by using the XPath search language. The methods return a set of nodes that match the given search.

These files are used by model cores to manipulate and parse their state strings. XMLBucketFiller is mostly used by the adapter class, but the XML should be used often in a model core to make sure the state string is up to date. Refer to the code for TestMCore.java to see how one should go about doing this.

3.2.7 Dependency Tree Navigation

*Code for this section includes TreeWalker.java, TreeIterator.java, TWModelNode.java, TWUserNode.java, and TWFileNode.java*

Values for a model can be set in three different ways: by another model; by the user; or from a file. For those values that are set by a user, we wanted to provide a single interface through which all of the values for dependent models could be set. This interface would be shown to the user and the values would be propagated to the appropriate models. See figure 3-17 for more details. The circles Mi represent
models and the user inputs come from the Model Settings interface. Arrows indicate dependency.

![Dependency Tree](image)

Figure 3-17: The dependency tree and user setting values

The user only sees the interface or output from the root model, but all of the other models are used in the calculation of its results. To visit all of the models in the dependency tree, we use the TreeIterator or TreeWalker. The TreeIterator steps through all of the nodes in breadth-first order. We do not guarantee this with the iterator, though. The only guarantee is that it will iterate through all the nodes. The TreeWalker, on the other hand, is much more flexible. With it one can step through the nodes in a number of different ways. It lets one get the child of the current node, the sibling of a current node and the parent of the current node. It also has methods for doing breadth-first and depth-first traversal of the tree.

Each node seen in the traversal is one of three types:

1. **TWModelNode** - A node which represents a model. It has an output and can also have required inputs.

2. **TWUserNode** - A node representing input by the user. It only has an output.

3. **TWFileNode** - Represents input from a file. It also only has an output.
The only nodes that can be internal to a dependency graph are model nodes. Model nodes, user nodes, and file nodes can be leaves on the tree. All of these node types inherit from TreeModelNode and have methods for accessing their specific information. An example of this is that model nodes can have inputs, where no other node type can.

At initialization the TreeIterator is used to traverse all of the nodes and determine what values the user needs to set for each of them. The classes, though, are provided as a library and can be used to do other manipulations as well.

3.2.8 Name Server

*Code for this section includes NameServer.java and ModelConnector.java*

The name server provides a registry of models that are running, their unique names, and the IOR where they can be reached. It is essentially a lookup table in which every model registers itself when it starts and from which it unregisters when it stops. When a model wants to find the IOR for another model it uses the unique name of the requested model to look up its IOR on the name server. Using a registry like the name server lets us get around a deficiency in CORBA whereby a model that is stopped and restarted will be forced to have a different IOR address every time. Some CORBA ORBs, such as ORBacus, provide name server implementations of their own, but they are implementation specific and not reachable if one uses a different ORB implementation. Our name server, on the other hand, since it is a true CORBA service, is accessible through any ORB implementation. In the future we might even what to move away from CORBA. Having our own name server implementation will let us do this if we want to.

One should note that all of the sending requests to the name server and registering and unregistering models is taken care of by the model adapter. A researcher will not have to worry about this. Also, if they want to make connections to another model the ModelConnector class hides all of the requests sent to the name server. One simply needs to know the unique name of the model and it will find a reference
to it for you.
Chapter 4

Prototype

To test the architecture, we wanted to build a set of models that would partially simulate a real-world problem. The goal was to provide a proof-of-concept and not necessarily a rigorous stress test, since this could only be provided by a real application. The goal was instead to make any holes in the architecture apparent and to provide a prototype for people to copy when publishing their own models.

The real-world problem we decided to tackle, but in a simplified manner, was the integration of two models being developed at UCSD. One of them is a finite-element model of the surface of the heart and the other is a resistor network model of the propagation of electrical signals along its surface. The integration of these two has been attempted by the two research groups, but progress has been slow. The models were not designed to be integrated. A good amount of work will be need to be done in order to have them talk to each other. They will both have to expose values to the other model and incorporate this data into their own calculations. This problem is compounded by the fact that they are written in different languages and reside on different machines. These are all problems that our architecture tries to overcome.

We decided to simplify the above models, because the publishing of each of the original models could be a Masters thesis in and of itself. Instead, we decided to focus on analyzing and testing the communication architecture itself.

To simplify the system, we decided to model a two-dimensional cross section of the heart, which contracts in response to stimuli and then expands gradually to its original
state. This mimicked the function of the finite-element model developed at UCSD, but removed a dimension and used a much simpler calculation for the surface deformation. To simulate the resistor model of electrical signal propagation, we decided to develop a pulse model that would periodically stimulate the two-dimensional heart model. The frequency and intensity with which the pulse model activated the heart could be set by the user. This, again, greatly simplified the computations of the reference model, but still had most of the same characteristics as far as interaction with the heart model was concerned. Lastly, we developed a simple model that would act as a data store, pulling values out of a file and publishing them to the network. In our case, a file was used to store the 2D geometry of the heart cross section, which is published by the data store. The heart model uses these values to determine the geometry of its polygon on startup.

The flow of values between the three models is shown in figure 4-1.

![Diagram](image-url)

Figure 4-1: Communication flow between the models
Figure 4-2: The heart model's user interface, in expanded and contracted state
4.0.9 Heart 2D Model

The heart model displays a 2D representation of a cross section of the heart. Our goal was to see how a model could be published through our architecture, so we had two stages to the development. First, a stand-alone model was created that would represent a rough approximation of what a research group would start with. Then we modified this pre-existing code base to make it use other models on the network and publish its own state.

The original model starts with a polygon which represents the cross section of the heart. This polygon is then deformed by activations on the nodes of the polygon, thereby contracting the shape. The heart then expands back to fit its original polygon shape until it is activated again. The graphical rendering of the polygon is taken care of by the HeartPanel class, which subclasses the standard Java Panel class. An instance of this panel, initialized with polygon information from a file, is contained within the UI class, which displays the complete user interface, including buttons and menu, to the user. The activation and relaxation of the polygon is carried out by an independent thread, which calls the tick() method on the HeartPanel repeatedly. This in turn calls the deactivate() and activate() methods of the HeartSurface2D object, which calculates and actually deforms the polygon.

We wanted to then publish the model through our architecture. We needed to have it reference the two other models described above. In order to do this, a number of things needed to be changed:

1. Pulses, or commands to contract originally came from an independent thread that would activate the heart at periodic intervals. We wanted this to be taken care of by the pulse model instead.

2. Originally the 2D geometry of the heart was read out of an XML file. We now needed it to be read from the data store.

3. It needed to display its UI to the user when the model was attached to the network and start running when the start command was called on the model core.
To solve these design issues we built a model core called Heart2DCore, a new UI, and a new heart panel class. Both the new UI and panel classes are very similar to the originals, but with minor modifications.

![Diagram](image)

**Figure 4-3: How the Heart2DCore, UI, and HeartPanel interrelate**

The UI class was changed so that the geometry of the heart cross section would come from values passed down from the Heart2DCore and ultimately from the data store. The UI class was also modified so that the activation values would be passed down from the model core. We also made it so the heart panel could be fetched, since the `activate()` method needed to be called by the model core whenever the pulse model triggered a contraction.

The thread spawned within the HeartPanel class to incrementally relax the model over time after an activation also was originally used to activate and contract the polygon at specific intervals. In the new version we didn’t want this process to be automatic. Instead the activation of the polygon would be done by the Heart2DCore class whenever the pulse model triggered it. This was the only change made to the HeartPanel class.

### 4.0.10 Pulse Model

The pulse model is designed to activate the heart 2D model periodically, thereby contracting the polygon cross section. In order to do this the heart model registers
with the pulse model for an asynchronous request. A frequency and magnitude for the pulse model is set by the user or from a file and the pulse model sleeps for a designated amount of time so that it is active and changes its state at a rate that is equal to “frequency” number of times per second. Since the heart surface model is registered for asynchronous output from the pulse model, it then pushes data to the heart model each time it is activated.

The simple state stored within the pulse model is shown in figure 4-4.

```xml
<?xml version='1.0'?>
<heart>
  <pulse-info updated='true'>
    <step>0</step>
    <frequency>0.5</frequency>
    <magnitude>0.4</magnitude>
  </pulse-info>
</heart>
```

Figure 4-4: State stored within the pulse model

The count increments from zero over the life of the pulse model. The other values are set by the user or from a file and can be changed on the fly if so desired. Generally, though, these values stay constant over time.

4.0.11 Data Store

The data store, implemented by the class DBCore, can be used to publish any static information or data that is changed by external sources, such as a user or other models. It simply loads its state from a file and then publishes these values to the network. In this case we use it to specify the geometry of the heart model.

The state string stored and published for the heart model is as shown in figure 4-5.

DBCore was designed to show how a database might be represented through our system. One can query and modify its contents by using the bucket syntax. Granted, it doesn’t provide complex functionality like transactions or SQL queries,
<?xml version="1.0"?>
<heart name="Orion's Test 2D Heart">
  <polygon>
    <node name="n1" x="130" y="40"></node>
    <node name="n2" x="90" y="50"></node>
    <node name="n3" x="70" y="70"></node>
    <node name="n4" x="50" y="100"></node>
    <node name="n5" x="40" y="140"></node>
    <node name="n6" x="40" y="170"></node>
    <node name="n7" x="50" y="210"></node>
    <node name="n8" x="70" y="240"></node>
    <node name="n9" x="90" y="260"></node>
    <node name="n10" x="130" y="270"></node>
    <node name="n11" x="170" y="270"></node>
    <node name="n12" x="210" y="260"></node>
    <node name="n13" x="230" y="240"></node>
    <node name="n14" x="250" y="210"></node>
    <node name="n15" x="260" y="170"></node>
    <node name="n16" x="260" y="140"></node>
    <node name="n17" x="250" y="100"></node>
    <node name="n18" x="230" y="70"></node>
    <node name="n19" x="210" y="50"></node>
    <node name="n20" x="170" y="40"></node>
  </polygon>
</heart>

Figure 4-5: State stored within the data store model
but this is not a limitation of the communication architecture, only of the DBCore implementation itself. More work will definitely be done in the future on integrating database access into our architecture.

4.1 How to create your own model core

To publish a model, one needs to create a Java class that implements \texttt{edu.mit.physiome.adapter.ModelCore}. Refer to the sample models for more details on how to do this. An important thing to remember is, make sure the XML state string contains the values you want to publish at all times. A lot of problems can happen if the state string is not updated correctly. We suggest that you store all of your state in this string and force yourself not to have any other state variables.

Second, create an \texttt{info.xml} file that describes the inputs and outputs for your model. Specifying the outputs are not so important, but specifying the inputs is critical, especially if you want to have values set by the user. An example of how these two steps should be done is given in \texttt{edu.mit.physiome.heart2d.Heart2DCore} and is explained below.

Lastly, run the server class \texttt{edu.mit.physiome.adapter.Server} with your new core class name as a command line argument. The call would be something like this:

\begin{verbatim}
java edu.mit.physiome.adapter.Server <your core class name>
\end{verbatim}

Make sure that the classes you’ve just created and all of the utility classes which come with the architecture can be found through your Java class path.

4.1.1 Storing all your state in the state XML string

Variable get methods should make use of the XML utility class to parse out specific values from the XML state string. The utility class is modeled after Microsoft’s MSXML classes for Visual Basic and are very convenient for grabbing specific values out of the state string. Note that the search path passed into the \texttt{findNodes()} method can use any of the XPath grammar as defined by the W3C. Go to
http://www.w3.org/TR/xpath for more information about XPath and the search grammars that can be used with it. Also, for an example of how we've used the XML.java and XMLBucketFiller.java classes in a model core, check out the code for edu.mit.physiome.heart2d.Heart2DCore.

4.1.2 Implementing the start, step and stop methods

The **start()** method is simply what you want to be done when the model starts. This might be calling a command line method that actually starts an external program or it might be creating a new thread that will run the model itself. It might also just do initializations, so that all of the calculations will be taken care of by the model core itself and no outside models will be called. Given the complexity of the biological models being developed today, though, this last scenario is unlikely.

The **step()** method is called repeatedly as the model is running. This is ideal for doing incremental calculations or monitoring the progress of an external model’s calculations. The frequency of calls to the step method can be varied by sleeping the core’s parent thread. An example of how this is done can be seen in any of the model cores in the heart model test case described earlier in this chapter.

The **stop()** method is called when a model is about to be shut down. It should contain any cleanup code, such as the saving of files, terminating of threads, or closing of network connections that need to be made. Unfortunately, in the current implementation, there are times when the model is terminated without a call to the stop method. This happens when a call to System.exit() is done and the Java VM itself is explicitly shutdown. There are ways to get around this, but given the time constraints, this was beyond the scope of this thesis. Future iterations of the architecture should address this problem.
Chapter 5

Future work

This architecture is meant to be a foundation on which future applications can be built. There are many ways in which this architecture can be extended. We will discuss a few potentially interesting areas here.

5.1 Publishing of models to the network

One of the most important extensions to our current work is to use the architecture to publish pre-existing models. The architecture is meant to facilitate and simplify this process and the more people use it the better. The physiome group at MIT is looking into some real-world test cases where research labs want to share their simulations with each other. The surface model of the heart and resistor network model of electrical propagation being developed by Andrew McCulloch and his group at U. C. San Diego are candidates for this first test case.

Although our group will be dedicating time to one test case, any interest in using the architecture for other domains would also be welcome. Only through extensive testing will we be able to see if the architecture is general enough to provide all of the functionality needed by current researchers.
5.2 Creation of a toolkit

All of the code and documentation developed should eventually be combined into a toolkit to be used by model builders to publish their models to our network. This is reinforced by our core goal of trying to make it as simple as possible for people to publish their current models through our architecture.

To take the current code base and create a toolkit out of it, one would need to clean up the documentation, create an installer, and provide some apps that would set up files so that the model builder could simply implement the portion of the interface connected directly to their model.

A simple way in which to use other models on the network should also be provided. A rudimentary version of this is provided by the ModelConnector class, which returns a model’s interface given the name it is registered with on the name server or it’s IOR. A more complete implementation might let the user pick inputs needed for his/her model and then have these values be magically available through a set of generated files.

There are a number of ways in which installation and use of our code can be simplified. They should all be explored and incorporated into a toolkit that can be released to biological researchers to publish their own legacy simulations.

5.3 Replicate work done in Java in C and C++

I’ve built helper classes in Java for parsing XML, implementing the ModelI interface and plugging in an arbitrary model core, but these classes also need to be made available in C and C++. The main reason for this is that many of the models biological scientists create are either implemented in C/C++ or have libraries available that let one access them through these languages. For example, any model created with Matlab can be easily accessed and manipulated through C, since Matlab provides C header files that expose their API. They do not yet provide these for Java.

Although I think reimplementing the classes in C/C++ would be the cleanest
solution, another approach would be to somehow wrap our current classes so they are accessible through C/C++. This would be a good solution if the code base was to change, since then those changes would only have to be made once and would be reflected in both languages. The drawback to this is that without a component model such as COM, wrapping objects and making them available in another language is not very clean. Also, if we were to access Java classes from C/C++ there would be a real performance hit, since Java is inherently slower than either of the C-flavored languages. My rational for using Java for our first implementation was its extreme portability and my own familiarity with the language, but we wouldn’t want to penalize C/C++ users by forcing them to use slower Java classes.

5.4 Security

In the current implementation I decided to postpone the security issues, since, in its current revision, our architecture was supposed to simply provide a means by which models could be published to everyone. It’s likely that people will want to limit access to models they’ve published and encrypt communication of sensitive data. This is where adding security features such as authentication and encryption would come in handy.

It might be possible to leverage CORBA and use their security mechanisms instead of recreating them for our specific architecture. If it is not, though, some possible approaches might be to create secure versions of the methods on the ModelI interface or to create a completely new interface, SecureModelI, which would be virtually identical to ModelI, but with security requirements. I would suggest that one look into the CORBA-level security first and then explore the creation of a separate SecureModelI interface. If neither of these work I would then look at extending ModelI and adding secure methods to it.
5.5 Multiple connections to a single model

Up until now we have implicitly assumed that each model is only used in one tree of sub-models, eventually giving a single aggregate result to the user. It is possible, though, to have situations where one model will be used by a number of different model networks and be required to be in different states at the same time. Refer to figure 5-1 for a graphical depiction.

![Overlapping Model Dependency Trees](image)

Figure 5-1: Referencing the same model in two different networks

There are two solutions I can see to this problem: One, simply say that a separate model needs to be created for each network of dependencies. This would require very few changes to the current implementation. Security might help with this problem, since you could make it so that every model in a given dependency network used the same authorization string and that any requests coming into a given model have to have the correct authorization. If they don’t, the request is coming from outside the dependency tree and can be rejected. A second solution is to always pass some form
of unique identifier to the model which indicates which specific version of the model one is working with. Essentially you would have multiple copies of the model running inside the model server, each with a different state. The unique identifier would let you query one of those models specifically.

5.6 Improvements to name server

The name server was one of the last things implemented. It is needed, since IOR’s, the general referencing mechanism used in CORBA, change every time a model is stopped and restarted. A way to reference models uniquely but across stops and restarting of the models was needed.

One of the biggest issues that will need to be addressed in a future version of the name server is that unregistration is currently not guaranteed. A model could exit but still be registered with the name server. This is possible, because models can be terminated by calls to `System.exit()`. In order to guarantee that a model is unregistered we have to guarantee that it is unregistered before `System.exit()` can be called. Some solutions include having the name server periodically poll the model to see if it is still up, finding some way to guarantee that the model always unregisters before exiting, or just having models fail gracefully if they get an IOR from the name server and the model is no longer accessible.

The name server might also become a bottle neck to system performance. Since the name server needs to be accessed every time a connection is made between two models, it is a limiting component of the system. Two potential ways to minimize the time needed for name server lookups include multithreading the server’s code and caching these lookup values locally. With the multithreading of the server one would spawn off a thread for each request. Locking issues would be minimized, since most of the time would be spent reading out values. Writing values to the server would only happen when a model shut down or restarted. In those cases we would have to use some form of locking, such as that provided by mutual exclusion semaphores. For caching of the values, we would also have some minor pitfalls that would need to
be negotiated. The most important one would be that values should be wiped from
the cache or replaced with the correct values whenever the server’s values change.
A possible approach would be to signal all of the caches to reset whenever a new
IOR-name pair was written to the server. This should work okay, since the number
of writes is small.
Chapter 6

Conclusion

In designing and implementing this architecture for inter-model communication, I realized that the problem I was tackling was more challenging than I had originally thought, but the final implementation seems to provide a thorough and robust solution. The architecture is also expected to change and improve over time.

Like any systems architect, I look forward to the time when I can see my work being used in real-world situations. Although initially it will be used in the heart physiome project, hopefully it will work its way into other fields of research as well. The goal was to make it general and extensible enough that the networking of models in domains ranging from biological simulation to fluid dynamics could provide added value for researchers that have invested countless man years in their proprietary models. By sharing and integrating these models, one can gain added insight and a more complete view of how these phenomenon work.
Appendix A

Glossary

COM  Component Object Model. A communication model developed by Microsoft and Digital Equipment Corp. for providing reusable components on a single machine. This was extended by DCOM, which provides the same functionality, but across multiple machines.

CORBA  Component Object Request Broker Architecture. A communication protocol that tries to be language and locationally independent.

DOM  Document Object Model. A reference to one way of manipulating XML files. This model uses a tree structure of nodes to represent a given XML document. Each node represents a tag, attribute of a tag, or the text contained within tags. These trees of nodes can be manipulated and created by use of a XML DOM Parser.

DTD  Domain Type Definition. A grammar that defines the kinds of information that can be specified in an XML file that uses the given DTD. Defined in a text file and published through the web or other means.
GIOP  General Inter-ORB Protocol. A standard protocol and messaging structure by which ORBs communicate. Both the structure of messages sent and assumptions about the transfer layer on which they are sent are specified.

IDL  Interface Definition Language. The language in which CORBA interfaces are defined.

IIOP  Internet Inter-ORB Protocol. Describes the translation of GIOP messages to be transferred over TCP/IP. This is the protocol used by ORBs for communication over the Internet.

IOR  The value by which objects in CORBA can be referenced uniquely over all ORBs communicating via IIOP. A CORBA object gets a new IOR each time it is stopped and restarted. The IOR value includes information such as the host machine of the ORB being used, the port on which it is listening, and the unique name of the given object in the ORB’s directory. All of these details, though, are encoded in a hex string.

OOC  Object Oriented Concepts, the developers of ORBacus.

ORB  Object Request Broker. An ORB is the software component that takes care of CORBA communications. During a request, two ORBs communicate with each other, one on the client end and one on the server end.
ORBacus  The ORB implementation we are currently using. It has source available and is free for educational use. The version we’ve used is 3.2.1.

SAX  Simple API for XML. An event-model method of manipulating XML files. A class is created that implements the SAX parser interface. The XML file is parsed by this class and specific node structures can be caught. The parser can then do an action whenever a specific node structure is encountered.

W3C  The World Wide Web Consortium. A standards organization that specifies web standards. Their homepage can be found at http://w3c.org/

XML  eXtensible Markup Language. A tag-based markup language similar to HTML and SGML used to encode information from any domain. Tags are user-defined and specified in a DTD.

XPath  A language specification made by the W3C for searching XML documents. XPath implementations return a set of nodes that match the query in a given document.
Bibliography

    http://www2.ios.com/dclunie/dicom-status/status.html.


[3] Health Level 7 FAQ.
    http://www.mcis.duke.edu/standards/HL7/faq/HL7FAQ10.HTM.


[8] WHO Library - Historical Resources.
    http://www.who.int/hlt/historical/English/nomenclatures.html.

    Modeling of Integrated Metabolic Processes. In *German Conference on*


