NAT'L INST. OF STAND & TECH R.I.C.
A11104 936253

NIST
PUBLICATIONS

NISTIR 5820

Distributed Communication Methods and Role-Based Access Control for Use in Health Care Applications

Joseph Poole
John Barkley
Kevin Brady
Anthony Cincotta
Wayne Salamon

U.S. DEPARTMENT OF COMMERCE
Technology Administration
National Institute of Standards and Technology
Gaithersburg, MD 20899

QC
100
.U56
NO.5820
1996

NIST

NISTIR 5820

Distributed Communication Methods and Role-Based Access Control for Use in Health Care Applications

Joseph Poole
John Barkley
Kevin Brady
Anthony Cincotta
Wayne Salamon

U.S. DEPARTMENT OF COMMERCE
Technology Administration
National Institute of Standards and Technology
Gaithersburg, MD 20899

April 1996

U.S. DEPARTMENT OF COMMERCE
Michael Kantor, Secretary

TECHNOLOGY ADMINISTRATION
Mary L. Good, Under Secretary for Technology

NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY
Arati Prabhakar, Director

Abstract

The use of software in the health care industry is becoming of increasing importance. One of the major roadblocks to efficient health care is the fact that important information is distributed across many sites. These sites can be located across a significant area. The problem is to provide a uniform mechanism to integrate this information. This paper documents the results of an investigation into the suitability of several different distributed access mechanisms. Five methods were examined: the Common Object Request Broker (CORBA), Object Linking and Embedding (OLE), remote procedure call (RPC), remote database access (SQL/RDA) and Protocol Independent Interfaces (PII, we specifically examined sockets). These mechanisms were compared with regard for use in health care applications. In particular, the following capabilities were compared:

- Ease of use by the developer
- Class of applications for which the technology is particularly effective in developing
- Security capabilities
- Protocols utilized
- Performance of the transport mechanism.

A second goal was to explore the use of role-based access control (RBAC). RBAC is a security mechanism that is more flexible than Mandatory Access Control, but easier to use than just plain access control lists. Every user is assigned to one or more roles. Each role can perform some operations but not others.

A demonstration application was constructed that used the distributed communication methods to implement a patient record database. This report discusses how these mechanisms were used in the demonstration project and the results found. Not unsurprisingly, we discovered that each of the mechanisms were effective for different purposes. These findings are discussed in detail in this report. One component of the demonstration project also implemented role-based access control and is detailed in this report.

Keywords : access control, CORBA, distributed, health care, OLE, PII, RBAC, role-based, RPC, security, SQL/RDA, transport

Trademarks

Ada is a registered trademark of the U.S. Government (Ada Joint Program Office). Microsoft and Windows are registered trademarks of Microsoft Corporation. Microsoft Visual Basic is a trademark of Microsoft Corporation. Borland is a registered trademark of Borland International, Inc. Unix is a registered trademark of Novell, Inc.

Certain commercial products are identified in this report. Such identification does not imply recommendation or endorsement by the National Institute of Standards and Technology, nor does it imply that the product, publication or service identified is necessarily the best available for the purpose.

Acronyms

API - Application Program Interface
BOA - Basic Object Adapter
CDR - Common Data Representation
COM - Component/Common Object Model
CORBA - Common Object Request Broker
DAC - Discretionary Access Control
DCE - Distributed Computing Environment
GUID - Globally Unique Identifier
HTML - Hyper-text Markup Language
IDL - Interface Definition Language
IIOP - Internet Inter-ORB Bridges
MAC - Mandatory Access Control
NCS - Network Computing System
NDR - Network Data Representation
NIS - Network Information Service
OA - Object Adapter
ODBC - Open Database Connectivity
OLE - Object Linking and Embedding
ONC - Open Network Computing
ORB - Object Request Broker
OSF - Open Software Foundation
PII - Portable Independent Interfaces
POSIX - Portable Operating System Interface for Computer Environments
RBAC - Role-based Access Control
RDA - Remote Database Access
RPC - Remote Procedure Call
RPCL - Remote Procedure Call Language
TCP/IP - Transmission and Control Protocol / Internet Protocol
XDR - External Data Representation

Table of Contents

1. INTRODUCTION 7

2. TECHNICAL OVERVIEW OF TRANSPORT MECHANISMS 8

   2.1 CORBA 8
   2.2 OLE 12
   2.3 SQL/RDA 15
   2.4 SOCKETS 19
   2.5 RPC 21

3. TECHNICAL OVERVIEW OF ROLE BASED ACCESS CONTROL 24

   3.1 IMPLEMENTING ROLE BASED ACCESS CONTROL USING OBJECT TECHNOLOGY 24

4. DEMONSTRATION APPLICATIONS 30

   4.1 POSIX DEMO 33
      4.1.1 Operation of the server object methods 35
      4.1.2 Role-Based Access Control in the Server 35
   4.2 THE PC DEMO 37
      4.2.1 OLE Objects Used in the Viewer 37
   4.3 OTHER DISTRIBUTED COMMUNICATION METHODS 40

5. CONCLUSIONS 41

6. APPENDIX A - CODE FOR ROLE-BASED ACCESS CONTROL USING OBJECT TECHNOLOGY 43

7. APPENDIX B - IDL DESCRIPTION OF PATIENT RECORD OBJECT 49

8. GLOSSARY 56

9. REFERENCES 58

Table of Figures

Figure 1. CORBA Block Diagram 9
Figure 2. Components of OLE and COM 13
Figure 3. Illustration of Multi-vendor Database Environment 17
Figure 4. SQL/RDA using an API 18
Figure 5. Source Code for Writing to a Socket 20
Figure 6. RPC Function Call over a Network 22
Figure 7. Implementing RBAC with Layered Objects 25
Figure 8. Source Code for RBAC Example 28
Figure 9. Patient Record Database Object Entity Relationships 32
Figure 10. Block Diagram of Distributed Health Care Project 32
Figure 11. PC Demonstration Application Block Diagram 38
Figure 12. Source Code to link OLE COM Object 39

1. Introduction

Software is becoming of increasing importance in many industries, including the health care industry. An important characteristic of health care applications is that the data can be distributed across a wide area. Creating a comprehensive patient record can involve collecting information from many widely dispersed host machines. There are concerns about how fast large medical images can be transferred. There are also issues of how secure the data is when being transferred over a private network.

The purpose of this report is to examine different transport mechanisms and to evaluate their effectiveness in performing health care related tasks. We selected five different transport mechanisms to examine: CORBA (Common Object Request Broker), OLE (Object Linking and Embedding), PII (Portable Independent Interfaces, specifically sockets), SQL/RDA (Remote Database Access) and RPC (Remote Procedure Call). Each of these were evaluated using five criteria:

- How easy is it to use the product to develop applications?
- What is the class of applications that the product is best suited to develop?
- What are the security capabilities of the product?
- What network transport protocols can be utilized?
- What is the performance of the transport mechanism?

A demonstration project was created in conjunction with this report. The goal of the demonstration project was to give an example of how the various communication technologies can be used in an actual application. We built two viewers of patient record data. One viewer was a POSIX (Portable Operating System Interface for Computer Environments, see [IEEE1003.1c] and others) client that used a Hyper Text Markup Language (HTML) browser to examine patient data. The HTML browser found its data through a CORBA interface. The CORBA interface determined where the data of interest was located and queried the remote database using SQL/RDA.

The second viewer used OLE. Since OLE is not yet a distributed mechanism, this viewer had a different function. The viewer was used to contain patient data that has already been collected from other sites into a patient report. Additional documents created by other applications could also be added to the patient report. For example, an article concerning new treatment taken from a medical journal could be incorporated. The viewer also had limited database querying ability. Local databases which support the ODBC (Open Database Connectivity) protocol could be queried. This viewer was more limited than the previous viewer in the sense that it had to know the exact host on which the data resided.

In this section of the report we have presented a general project overview. The next section will review each of the distributed communication methods and apply the evaluation criteria to each. After that is the section presenting a technical overview of role-based access control. The following section covers the demonstration project. Finally, we will present our conclusions.

2. Technical Overview of Transport Mechanisms

2.1 CORBA

CORBA, the Common Object Request Broker, provides integration of object systems within a client/server framework. Clients issue requests for services on objects and the server performs the requested service. The CORBA server is termed the "object implementation". The clients are isolated from the object implementation through commonly defined interfaces which specify the makeup and operations associated with each object and how the operations on the object are provided. Clients seamlessly access the object implementation wherever that object implementation may reside, i.e., in a library on the same host as the client, in another process on the same host as the client, or on a different host from the client where the two hosts are connected by a network.

IDL is the language used for defining interfaces. IDL is programming language independent. When IDL is compiled, the output is programming language source code. Using IDL, the programmer describes the interfaces to the objects. The IDL compiler then takes the descriptions and outputs language in the target language. This code is a skeleton which the programmer then fills in. CORBA provides the Dynamic Invocation Interface which allows applications to generate requests at run-time. Target Language bindings are provided for C, C++, Ada and SmallTalk. The interfaces and methods of the IDL are mapped into programming constructs of the target language. For example, when IDL is mapped into C++, interfaces are mapped into classes and methods are mapped into functions, which perform the services of the interface. IDL does not specify how an interface is implemented. IDL does provide detailed information about the operations permitted on each object, the arguments expected, what is returned, and what happens when errors occur.

Running an IDL script containing an interface definition through an IDL compiler generates the code for the client stubs and the implementation skeleton. Every possible operation on an object defined in the IDL generates a client stub. The client stubs bind the client to the object, i.e. translate between local and standard data representations, and marshal the method call parameters to and from the object. The implementation skeleton marshals the method call parameters to and from the client. The CORBA components and flow of information is shown in figure 1.

CORBA includes

1. an object model;
2. the Object Request Broker (ORB);
3. an object-oriented Interface Definition Language (IDL);
4. language mappings to the IDL (e.g., C , C++, Smalltalk, Ada);
5. the mechanisms necessary for interoperability between clients and object implementations; and
6. standard services whose interfaces are defined in CORBA IDL.

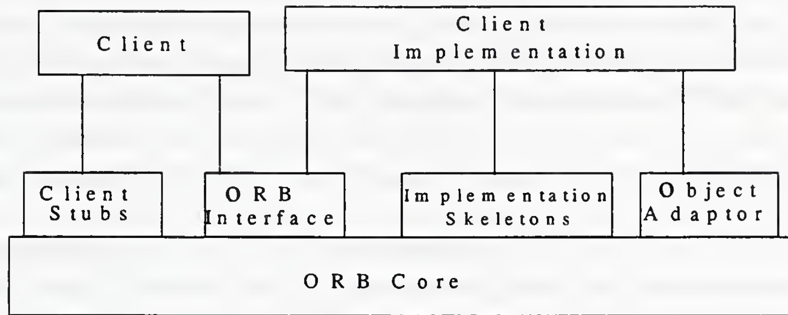

Figure 1. CORBA Block Diagram

These standard services are used by clients and object implementations when interacting with the CORBA environment. Examples of standard services include the Interface Repository which permits applications to get information about objects and their methods and Dynamic Invocation which permits applications to make requests to object implementations without having to be recompiled.

CORBA is specified in three documents available from the Object Management Group (OMG): The Common Object Request Broker: Architecture and Specification (CORBA) [CORBASPEC], CORBAservices, and CORBAfacilities. The CORBA specification includes the object model, the semantics of the ORB, the IDL, the language mappings, interoperability, and basic object services. The CORBAservices specification includes additional services useful for realizing and maintaining objects within a distributed computing environment. These services include naming, events, lifecycle, persistence, transactions, concurrency, externalization, and relationships. The CORBAfacilities specification includes both higher level services applicable across all application domains and higher level services targeted to specific application domains, such as manufacturing or healthcare.

As new capabilities are developed, they are added to these documents. Currently under development in the OMG are the following specifications:

- CORBA Security: a security model and architecture; object services for CORBA object security designed for application developers, administrators, and CORBA Security implementers; and interoperability.
- COM/CORBA Interworking: a capability for CORBA to interwork with Microsoft's COM (Component Object Model) (see section on OLE) including a mapping between COM and CORBA IDLs, the ability for CORBA clients to use OLE automation servers, and a protocol bridge.
- Asynchronous Messaging: an object service which would permit CORBA object requests and responses to take advantage of messaging technology featuring connectionless communication, message store and forward, message prioritization, and application triggering based on message events.

At a minimum, a CORBA implementation includes the ORB, the IDL, the standard services in the CORBA specification, and one language mapping. Optionally, a CORBA implementation may include additional language mappings, Interoperability, and additional services from the CORBAservices specification. Note that all of the components of a CORBA Implementation need not come from a single producer. In particular, services from the CORBAservices specification may come from a producer different from the one that provides the ORB.

CORBA can be viewed as the technological successor to Remote Procedure Calls (RPC). While CORBA provides a means for implementing applications using the client/server paradigm, CORBA has the following capabilities not found in RPCs:

- IDL is object oriented.
- Interoperability supports the concept of bridging (i.e., gateways) between different protocols.
- The Interface Repository Service which in addition to providing a directory service for interface names, also stores parameter names so that with the Dynamic Invocation Service, an application can dynamically discover objects and obtain all the information it needs to invoke those objects.
- An object implementation may be implemented as a library. Thus, for those applications which require high performance, existing objects and methods can be recompiled in a non-networked environment and achieve high performance.
- It supports multiple server policies for an object implementation, i.e., shared (all clients connect to a single server process which implements the methods of the object; server process exist only when there are clients), persistent (same as shared except server exists all the time), unshared (each client has its own server process), or server-per-method (each request creates a server process which terminates upon servicing the request). RPCs typically support only persistent and unshared server policies.

The purpose of the ORB is to direct object service requests from the client to the server and return server output values back to the client. The client and object implementation can exist in the same process, on different processes on the same or different hosts on the same or different networks. The ORB, based on the availability of the object implementation to the client, facilitates the transfer of object service requests from the client to the server and the results back to the client. These ORB facilities: transparency of object location, activation, and communication, are completely transparent to the client.

Because CORBA supports various types and styles of object implementations, the details for supporting specialized object implementations must be handled by the host of the object implementation requiring these objects. The OA (Object Adapter) provides an interface for this purpose. It assists the ORB with providing services such as activation, deactivation, object creation, and object reference management, such as generation and interpretation of object references and marshaling. A CORBA implementation may have many OAs. Each object implementation determines the OA it will use. CORBA expects each ORB to provide a general OA, the BOA (Basic Object Adapter). The BOA is intended to support objects implemented as separate programs. It is expected that most object implementations can work with the BOA.

The client operates on an object by issuing requests to the object. From the client's viewpoint, issuing a request is similar to a method invocation in a conventional C++ program. Most of the work is done by the client stubs and implementation skeleton generated by the IDL compiler. The steps associated with processing a client request, assuming the object implementation has registered its services with the ORB, are:

1. The object reference is obtained from an API provided by the ORB Interface.
2. The client generates an operation request using the object reference, explicit parameters, and an implicit invocation context.
3. The client request for an object service is marshaled by the client stub and sent to the ORB who delivers it to the object adapter.
4. The object implementation operates on the client request.
5. The results from the object implementation are passed through the skeleton which marshal the results to the ORB. The ORB returns them back to the client.

The object reference, which is opaque, identifies and locates a particular object. A client of an object has access to the object reference for that object. An object implementation may also be a client of other objects. To facilitate an efficient means for obtaining the object reference, the ORB services can make the opaque object reference persistent by converting it to a string. This string can then be stored and later retrieved and changed back to its object reference.

CORBA provides for interoperability between client and object implementations when the client and object implementations are located on different hosts connected by a network. This capability has two dimensions. The ORB provides the means for a client application to locate the object implementation even if the object implementation migrates to another host. The ORB also provides the means by which the client application can transmit a request to an object implementation and receive the response from that object implementation.

There are two protocols which provide interoperability in CORBA: the General Inter-ORB Protocol (GIOP) and the DCE Common Inter-ORB Protocol (DCE-CIOP). Support for the GIOP is mandatory for all implementations. The GIOP specifies protocols that are supported by the TCP/IP protocol suite. The DCE-CIOP is optional and is designed to work within an OSF DCE environment. In that environment, applications may make use of DCE security and management services. A CORBA implementation which provides the optional DCE-CIOP must also provide either the GIOP in addition to the DCE-CIOP or a bridge (CORBA terminology for gateway) to implementations that provide the GIOP. It is by means of either direct GIOP support by the DCE-CIOP implementation or this bridge that object systems within a DCE-CIOP environment can interoperate with object systems in a GIOP environment.

2.2 *OLE*

OLE 2.0 is a set of operating system extensions to Microsoft Windows used to facilitate application integration. It provides mechanisms that allow various application to exchange data without the applications having to understand the internals of the other application. The first version of OLE was much more limited than OLE 2.0; therefore when the report refers to OLE, OLE 2.0 is implied unless otherwise specified. OLE is intended to support the concept of the document-centered environment instead of a file-centered one. The user will work with documents which will invoke the applications required to edit themselves rather than the user first starting the application, then loading the file to be edited. This will allow the user to group work together by overall project organization rather than by application.

OLE is based on an object oriented model, so several definitions have to be covered first. There are many different definitions for object, for example "An encapsulation of data and services that manipulate that data"[IEEE610], and Grady Booch's "something you can do things to"[BOOCH94]. Methods are the services that manipulate the data or provide access to the object's data. The signature of a method is the data type of the method's return value and its parameters.

OLE is built on the Component Object Model (COM), which provides the basic infrastructure for OLE. There are two basic constructs in COM, COM Objects and interfaces. An COM Object is an object that can only be accessed through its interfaces. An interface is a set of methods which are used to access an object. An interface is usually implemented as a pointer to an array of function pointers. Interfaces can inherit the method signatures, but not the implementation of the methods. Each COM Object must support the IUnknown interface or an interface inherited from IUnknown.

The IUnknown interface provides three methods: QueryInterface, AddRef and Release. Objects use another object's QueryInterface method to inquire if the object supports a particular interface. If the interface is supported, QueryInterface returns a pointer to the supported interface, otherwise it returns a NULL pointer. AddRef and Release modify the objects reference count. Objects use the reference count to keep track of how many external objects are using them. AddRef increments the count, while Release decrements the count. When the reference count goes to 0, the object destroys itself. Interfaces are distinguished from each other by a unique GUID (Globally Unique IDentifier). This number is a 128 bit integer which can be assigned in blocks to vendors.

There are three basic services supplied by COM: persistent storage, intelligent names and Uniform Data Transfer. Persistent Storage is the ability to store the state data of COM Objects so that they can be deleted and later restored. Intelligent names not only name a COM Object, but also contain information on how to reference the contents of the object. Uniform data transfer allows two applications to exchange data without the applications understanding the internals of each other.

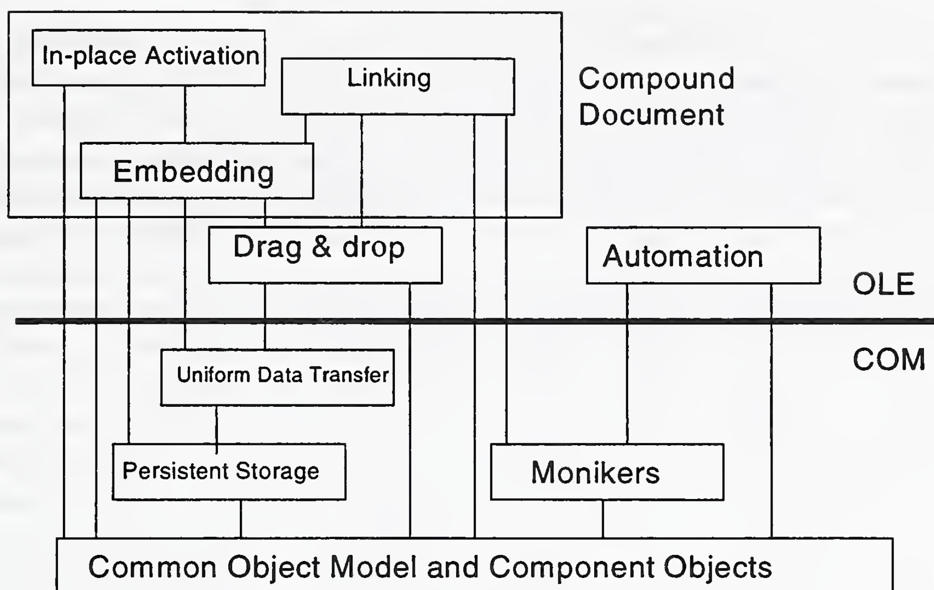

Figure 2. Components of OLE and COM

The services provided by COM are used as building blocks for OLE proper. The structures in OLE are more complex and at a higher level. One key concept is the idea of compound documents. For example, a word processing document can contain a spreadsheet created from a different application. A compound document is a document made up of other documents. An application which can store other documents inside of its data files is called a container. The application which creates the objects that are inside containers are known as servers. An object can be both a container and a server at the same time. The contained document can be either stored as a part of the top level document, in which case it is an embedded object, or the document can be stored externally and only a link stored in the enclosing document, in which case it is a linked object. When a contained object is activated for editing, the menus and toolbars of the application which created the contained object will merge with the controls of the container object. This is called menu merging. For example, assume that a spreadsheet is embedded inside a word processing document. The user can select the spreadsheet to be edited which will invoke the spreadsheet program. The menus and toolbars of the spreadsheet program will merge with the controls of the word processing program. Once the spreadsheet object is deselected, the original controls of the word processor will be restored.

Automation allows a program to control and send instructions to another program. The program that is being controlled is called an automation server. The program doing the controlling is the automation controller. A common automation server is a database program. A controller can invoke the server to supply data and to update the database.

OLE can not yet be used to build distributed applications. Many of the mechanisms are in place but not yet completed. The inter-machine communication will be Microsoft RPC which is based on DCE RPC. If the communicating processes are located on the same machine, then a special lightweight Remote Procedure Call protocol will be used. This protocol will bypass the

overhead of converting formats for network transmission. Network OLE is intended to be part of Microsoft Windows NT 4.0 which is currently targeted for 1997 or 1998.

OLE can be used at many different levels. There is a C language API defined. This level is very powerful but complicated to use. There are C++ frameworks that encapsulate the C functions that greatly simplify programming. There is also an IDL supplied similar to that of CORBA. The IDL has not yet attracted much use so far. A third method to program OLE is through Microsoft Visual Basic or with one of the programming languages associated with various Microsoft products, such as Microsoft Access. These are not as flexible as working with a C or C++ API, but they are much simpler.

No security is built directly into OLE. Since OLE is document based, file level security will serve to protect objects that are either embedded or linked into other objects.

2.3 SQL/RDA

SQL is a popular relational database language first standardized in 1986 by the American National Standards Institute (ANSI). Since then, it has been formally adopted as an International Standard by the International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC)[ISO9075]. It has also been adopted as a Federal Information Processing Standard (FIPS) for the U.S. government.

The SQL standard is very popular with a large and increasing number of conforming implementations. It is, or soon will be, the basis of definition for a majority of Federal databases and database applications involving structured data.

The basic structure of the relational model is a table, consisting of rows and columns. Data definition includes declaring the name of each table to be included in a database, the names and datatypes of all columns of each table, constraints on the values in and among columns, and the granting of table manipulation privileges to prospective users. Tables can be accessed by inserting new rows, deleting or updating existing rows, or selecting rows that satisfy a given search condition for output. Tables can be manipulated to produce new tables by Cartesian products, unions, intersections, joins on matching columns, or projections on given columns.

The purpose of the SQL language standard is to provide portability of database definitions and database application programs among conforming implementations. Use of the SQL language standard is appropriate in all cases where there is to be some interchange of database information between systems. The SQL definition language may be used to interchange database definitions and application specific views. The SQL data manipulation language provides the data operations that make it possible to interchange complete application programs.

RDA (Remote Database Access) is a communications protocol for remote database access that has been adopted as an ISO/IEC. This standard is in two parts:

1. Generic RDAANSI/ISO/IEC 9579-1:1993[ISO9579-1]
2. SQL Specialization ANSI/ISO/IEC 9579-2:1993[ISO9579-2].

Part 1 specifies the generic model, service, and protocol for arbitrary database connection and Part 2 specifies additional protocols for connecting databases conforming to the Database Language SQL.

RDA provides standard protocols for establishing a remote connection between a database client and a database server. The client is acting on behalf of an application program while the server is interfacing to a process that controls data transfers to and from a database. The goal is to promote the interconnection of database applications in a multivendor environment.

RDA is appropriate for remote access to a database in any context where lower layer transport protocols have already been established. RDA protocols have been shown to work properly in both OSI and Internet communications environments. The Internet RFC1006 is the guide used for executing RDA over a TCP/IP connection.

The RDA Service Interface consists of service elements for association control, for transfer of database operations and parameters from client to server, for transfer of resulting data from server to client, and for transaction management. Association control includes establishing an association between the client and server remote sites and managing connections to specific databases at the server site. Database operations are sent as character strings conforming to the SQL language. Resulting data and/or errors and exceptions are described and represented using the ISO ASN.1 standard. Transaction management includes capabilities for both one-phase and two-phase commit protocols.

RDA is appropriate in situations where it is not desirable, or possible, to run the same vendor's software at both ends of a communication line. Interconnection among database products from the same vendor will likely continue to use vendor specific communication and interchange forms.

Security work for RDA has been on-going. The RDA protocol maintains the security already inherent in a relational database with regard to access control. And the standard itself has left 'placeholders' for added security needs in the area of authentication. As with any standard for interoperability, the algorithms and methods must be agreed upon by all potential users.

Figure 3 depicts how SQL/RDA can be used in a multivendor network environment. The RDA protocol is used to communicate between three different SQL databases, manufactured by three different vendors, on three different hardware platforms. This configuration demonstrates the viability of the RDA standard with SQL databases in a heterogeneous environment.

The application program accesses each database server by means of a standard application program interface (API). The attached sample program in figure 4 provides an example of how this API may be used. The client application makes a connection to the server by specifying the server name (machine name in this case) and the data resource to be opened (the username on the database). The client can then initiate transactions on the database by sending an SQL string, and get back a table of results.

Work to standardize this interface will soon be completed, and is currently being prototyped at NIST. The Call Level Interface(CLI) (ISO/IEC 9075-3:1995) will provide a standard API to the RDA protocol. The CLI is a super-set of the familiar and very popular ODBC (Open Database Connectivity) defacto standard. By adding RDA to an ODBC/CLI API, the need for numerous drivers on both the client and server will disappear.

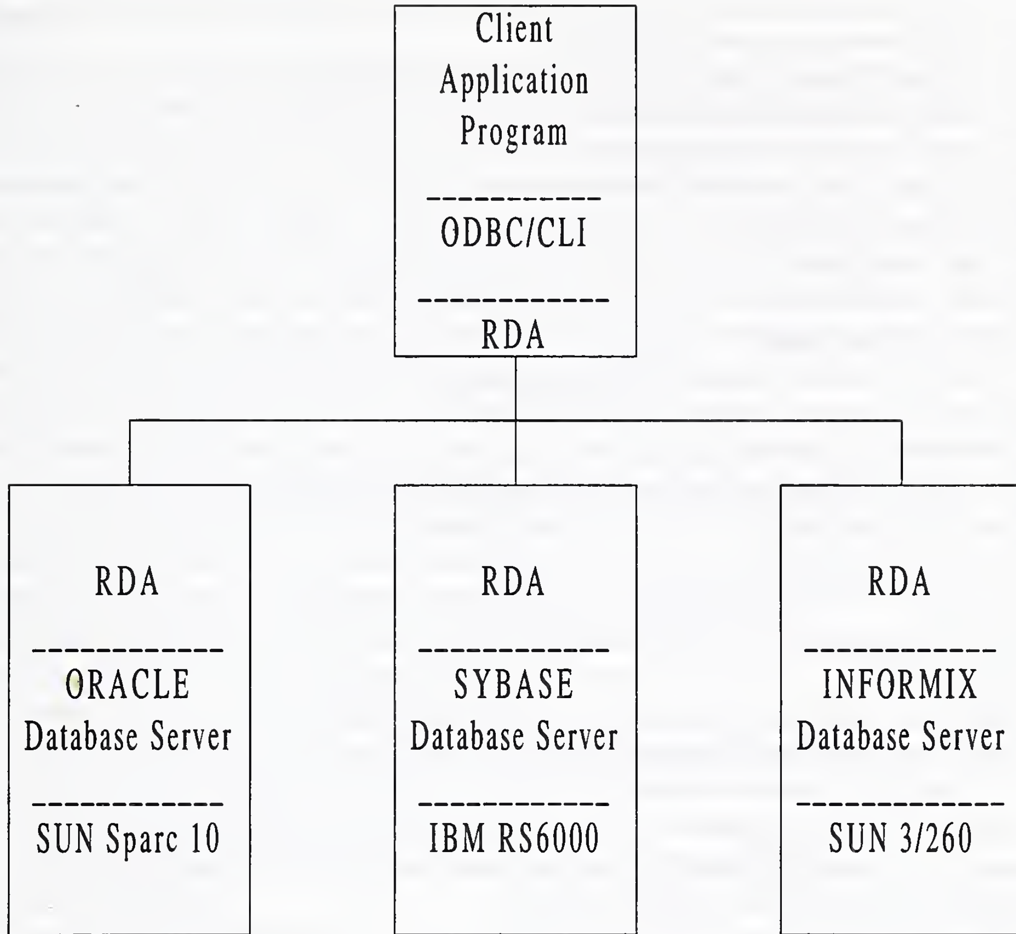

Figure 3. Illustration of Multi-vendor Database Environment

```
// assume the following are already defined
// hostname  - char*, name of host where RDA server is running
// server    - char*, name of the RDA server
// rname     - char*, name of the resource
// password  - char*, password to server
// status    - int, return value
// dia_id    - int, dialogue id
// rsc_id    - int, resource id
// colc      - int, column count
// colv      - char**, pointer to array of column names
// rowc      - int, row count
// rowv      - char**, pointer to array of row values
// sqlcmd    - char*, ASCII string of the SQL command to execute
// errorcode - int, error code
// errortext - char*, error message string

        /* init a dialogue */
        rda_setup();
        status = rda_init(&dia_id, hostname, server, password, 0);
        if(rda_error(status)) rda_signal( status, 0);

        /* Open the data resource */
        status = rda_open( did, &rsc_id, rname, password, RDA_C_UPDATE);
        if(rda_error(status)) rda_signal( status, 0);

        /* start the transaction */
        status = rda_begin( dia_id);
        if(rda_error( status)) rda_signal( status, 0);

        /* perform the transaction */
        status = rda_execSQL( dia_id, rsc_id, sqlcmd, &colc, &colv,
                &rowc, &rowv, &errorcode, &errortext);

        /* commit the transaction */
        status = rda_commit(dia_id);
        rda_signal(status, 0);
```

Figure 4. SQL/RDA using an API

2.4 Sockets

Sockets is an inter-process communication mechanism that was introduced in 1981 as part of BSD 4.2, the Berkeley distribution of Unix. It is the standard method of inter-machine communication. Sockets is not tied to any particular transport mechanism, although the most popular inter-machine protocol is TCP/IP. Communication between machines can either be connection oriented with reliably delivery (TCP) or connectionless with unreliable delivery(UDP). Some systems also support reliable connectionless connections. Winsock is a PC adaptation of POSIX sockets [IEEE1003.1g] with some changes and extensions. The extensions are mainly for asynchronous communication to allow socket programming to better interact with the Microsoft Windows operating system.

A number of support routines are included. There are functions to convert hostnames to IP (Internet Protocol) addresses and other network nameserver services. There are routines provided for simple conversions of integers to a network format for inter-machine communication. Conversions of more complex data types to a machine neutral format can be done with the XDR (external data representation) library.

Sockets is a low level mechanism with a correspondingly low level API. This allows for a large amount of control, but it also means that a large number of steps are needed to perform operations. The procedure required to send a message using TCP is

1. create a socket using the socket function;
2. find the IP address on the host using the gethostbyname function;
3. copy over the required datafields, being sure to convert to network format;
4. connect to the correct port number on the server machine;
5. convert the data to a machine independent format using XDR; and
6. send the message.

The code to perform this procedure is shown in figure 5. Notice that error checking has to be performed at all steps of the process to ensure that the program does not terminate.

Sockets is best used for high performance bulk transfer applications. The only language binding commonly used is C although some of the scripting languages, such as Perl, encapsulate some of the features of sockets.

There is no standard security mechanism built into sockets at this time although the SSL (Socket Security Layer) proposal [SSL] is gaining wide support. SSL was originally developed for use in WWW browsers to allow secure money transactions.

```
// client.cpp

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stream.h>
#include <rpc/types.h>
#include <rpc/xdr.h>
#include <stdio.h>
#include <stdlib.h>

int main( int argc, char** argv) {
  struct  hostent* host;
  struct sockaddr_in addr;
  int value, svalue;
  char buff[ 32];
  int sock;
  XDR xdr;

  if( argc != 3) {
    cerr << "usage: client host value\n";
    return 1; }

  sock = socket( AF_INET, SOCK_STREAM, 0);                          // create the socket
  if( sock == -1) {
    cerr << "client can not create socket\n";
    return 1; }

  addr.sin_family = AF_INET;                                         // get the host address
  host = gethostbyname( argv[1]);
  if( !host) {
    cerr << "can not find host " << argv[1] << endl;
    close( sock);
    return 1; }

  memcpy( (char*) &addr.sin_addr, (char*) host->h_addr, host->h_length);   // fill in the address
  addr.sin_port = htons( 2000);                                      // fill in the destination port
  if( connect( sock, (struct sockaddr*) &addr, sizeof addr) == -1) {  // make a TCP connection
    cerr << "client can not connect to server\n";
    close( sock);
    return 1; }

  value = atoi( argv[2]);                                            // create the data for a simple packet
  svalue = value * value;

  xdrmem_create( &xdr, buff, 32, XDR_ENCODE);                        // create the XDR stream
  xdr_int( &xdr, &value);                                            // encode the data
  xdr_int( &xdr, &svalue);
  write( sock, buff, 32);                                            // send the data packet
  close( sock);                                                      // clean up
  return 0;
}
```

Figure 5. Source Code for Writing to a Socket

2.5 RPC

RPC, Remote Procedure Call, provides an application the ability to request services from other processes, usually remote, by means of a function call. The RPC concept is based on a client/server framework. The client application can execute a procedure on a local/remote machine, pass data to it and retrieve the result. When the machine is remote, RPC uses the communication resources of the underlying network.

The two most widely used RPC implementations are ONC (Open Network Computing) RPC and DCE (Distributed Computing Environment) RPC. ONC/RPC, sometimes referred to as Sun/RPC, was developed by Sun Microsystems. ONC/RPC was one of the first commercial implementations of RPC. The success of ONC/RPC is in some measure related to the widespread use of NFS which is implemented using ONC/RPC. NFS has been implemented in many diverse environments, e.g., IBM MVS, DEC VMS, and Novell Netware.

DCE/RPC was developed by the Open Software Foundation (OSF). The DCE/RPC protocol is used as an optional protocol in CORBA (see section 2.0). DCE/RPC is also the protocol used by COM to extend COM functionality over a network (see section 2.1).

The basic operation of RPC is illustrated in figure 6. The paradigm of RPC is based on the concept of a function call in a programming language. The semantics of RPC are almost identical to the semantics of the traditional function call. The major difference is that while a normal procedure call takes place between procedures of a single process in the same memory space on a single system, RPC takes place between a client process on one system and a server process on another system where both the client system and the server system are connected to a network.

A client application issues a normal function call to a client stub. The client stub receives arguments from and returns arguments to the calling function. An argument may instantiate an input parameter, an output parameter or an input/output parameter.

The client stub converts the input arguments from the local data representation to a common data representation, creates a message containing the input arguments in their common data representation, and calls the client runtime, usually a library of routines that supports the functioning of the client stub. The client runtime transmits the message with the input arguments to the server runtime which is usually an object library that supports the functioning of the server stub. The server runtime issues a call to the server stub which takes the input arguments from the message, converts them from the common data representation to the local data representation of the server, and calls the server application which does the processing.

When the server application has completed, it returns results to the server stub in the output arguments. The server stub converts the output arguments from the data representation of the server to the common data representation for transmission on the network and encapsulates the output arguments into a message which is passed to the server runtime. The server runtime transmits the message to the client runtime which passes the message to the client stub. Finally, the client stub extracts the arguments from the message and returns them to the calling procedure in the required local data representation.

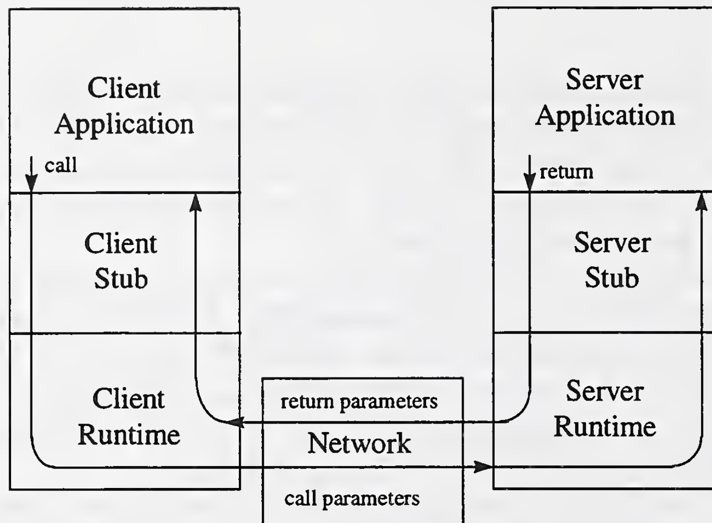

Figure 6. RPC Function Call over a Network

Applications which use RPC programs are developed by using an IDL similar to the approach used by CORBA. The only language binding supported is C. RPCs support persistent servers, which is where a single server exists all the time, and unshared server policies, where a server is created for each call.

The transport protocols supported by ONC/RPC are TCP and UDP. DCE/RPC also supports DECNET. The reliability of delivery under ONC/RPC is that of the underlying transport layer, DCE/RPC, on the other hand, guarantees delivery independently of the transport used.

ONC/RPC uses the XDR protocol to communicate data between heterogeneous hosts. XDR uses a single basic format for transferring data where both sender and receiver perform data translation. NDR, the format used by DCE/RPC, has a list of sixteen formats. The sender specifies the format sent and the receiver of the message is responsible for data translation, if needed. The sending host specifies within the protocol which data format was used

Security in ONC/RPC, called secure RPC, is provided by NIS+. NIS+ provides both authorization and authentication. Every object in the namespace specifies the type of object it accepts and from whom. For each access request on the namespace, the originator is identified and a determination is made whether to provide access. There are three levels of authentication available:

1. no security
2. traditional process permission: uid, gid, supplementary groups, and machine name are provided
3. verification of identification provided (based on Diffie/Hellman key distribution DES encryption techniques).

Security in DCE/RPC is provided by kerberos. DCE/RPC provides authentication, authorization and access control services. Both client and servers can be authenticated. Authorization services enable a user, host, or server to determine the rights of other users, hosts, or servers.

ONC/RPC uses NIS+ to handle a global name service. NIS+ is used to associate a service with the service name and the domain where it is located. NIS+ tables are administered through NIS+ administration commands. The namespace is arranged into configurations called domains. Domains contain directories, tables, and groups (which are denoted as "objects"). Every domain is supported by a set of NIS+ servers, which handle the NIS+ client requests for the domain. NIS+ maintains an RPC table containing the RPC program name, program number, and its aliases.

DCE/PRC uses a global directory service based on X.500 Directory Services (XDS). GDS offers the opportunity for access to worldwide resources. GDS provides a lookup service for all the networked services and machines that work together. Servers export their bindings (RPC protocol type, host network address, and transport endpoint) and the objects they manage to an entry in the GDS. A server that can accept multiple RPC protocols, will export a binding for each protocol it supports. Clients, employing automatic binding, locate the compatible server via IDL generated stubs that search the CDS. The appropriate RPC protocol binding is the only binding provided to the client.

3. Technical Overview of Role Based Access Control

There are two basic types of access control mechanisms used to protect information from unauthorized access: discretionary access controls (DAC) and mandatory access controls (MAC). Because DAC places the decision of who can access information at the discretion of the creator of the information, DAC is not applicable to the majority of health care information. Because MAC requires all those who create, access, and maintain information to follow rules set by administrators, MAC is the kind of access control mechanism required of health care information.

The most commonly used MAC is the multi-level security mechanism used by the Department of Defense (DOD). This is the mechanism which associates information with such labels as TOP SECRET, SECRET, and CONFIDENTIAL. It has become apparent that this type of MAC is not sufficiently flexible for industry use. This type of MAC is also not adequate for the needs of health care.

Role Based Access Control (RBAC)[RBAC] is a MAC which has been developed at NIST to meet the needs of industry. Rather than labeling information, it associates roles with each individual who might have a need to access information. Each role defines a specific set of operations that the individual acting in that role may perform. The operations may be broad or very specific, e.g., when a diagnosis is entered into a patient record, the symptoms leading to that diagnosis must also be entered. Once an individual has been properly identified and that identification authenticated, the individual chooses a role that has been assigned and accesses information according to the operations assigned to the role.

This project determines the applicability of RBAC to health care information. While it is generally accepted that RBAC is more suited to health care than others, the question remains as to whether RBAC meets all of the requirements for the security of health care information. Moreover, there are several variations on the RBAC model and there is the question of which variations are most suitable for health care information.

In order to illustrate the usefulness of RBAC to health care, this project also produces a demonstration of the use of RBAC with patient records. The demonstration suggests different roles that are appropriate with patient records and defines sample operations associated with those roles.

A sample RBAC policy related to clinical and administrative patient data has been identified. This draft specification [GRIEW], represents some degree of consensus on a policy for patient information access. The UK policy is RBAC with the addition of the capability of labeling information that is only available to the patient and the doctor. It specifies roles and the level of access permitted by each role.

3.1 Implementing Role Based Access Control Using Object Technology

With Role Based Access Control (RBAC), each role is associated with a set of operations which a user in that role may perform. The power of RBAC as an access control mechanism is the concept that an operation may theoretically be anything. This is contrasted to other access control mechanisms where bits or labels are associated with information blocks. These bits or labels indicate relatively simple operations, such as, read or write, which can be

performed on an information block. Operations in RBAC may be arbitrarily complex, e.g., "a night surgical nurse can only append surgical information to a patient record from a workstation in the operating theater while on duty in that operating theater from midnight to 8 AM." A goal for implementing RBAC is to allow operations associated with roles to be as general as possible while not adversely impacting the administrative flexibility or the behavior of applications.

Consider the possible activities associated with defining and modifying roles:

- Add a role and its associated operations.[1]
- Remove a role and its associated operations.
- Modify an existing role:
    - ◊ Add an operation.
    - ◊ Remove an operation.
    - ◊ Modify an existing operation.

Information is usually accessed by applications based on a fixed set of operations defined by the mechanism or processor which is used to access the information. Applications are built based on a fixed set of operations which they routinely perform. For example, Unix files are accessed by the operations defined by the procedures: *open()*, *close()*, *read()*, *write()*, *fseek()*, etc.; tables in a relational data base are accessed by the operations defined by SQL.

Modifying the operations available to an application can have a great impact on an existing application. Removing an operation or modifying the semantics of an operation seriously affects an application's functioning and can produce very unpredictable results.

One approach which can be used to maintain flexible administration, minimize impact on applications, and maintain a significant capability for defining complex role operations is to use Object Technology as in figure 7.

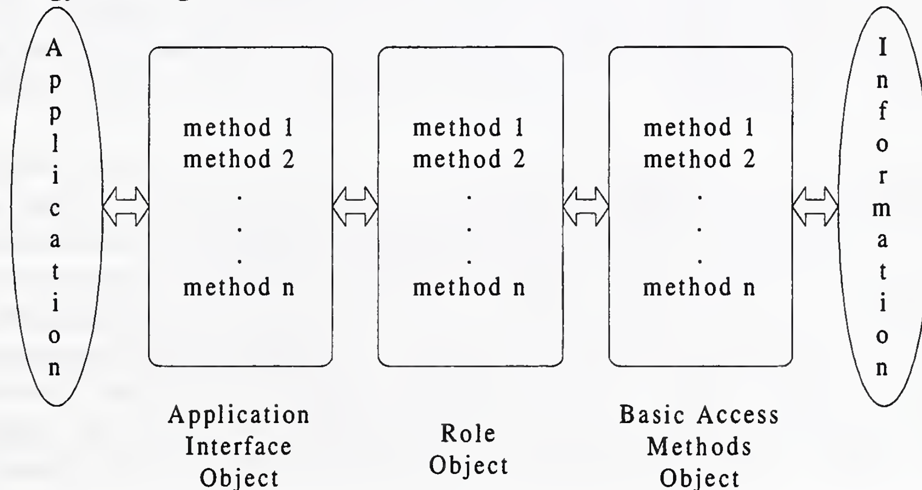

Figure 7. Implementing RBAC with Layered Objects

[1] Some operations may be available to more than one role, e.g., a credit account may be read by both a bank teller and a bank supervisor

A complete set of operations based on access methods associated with the information storage mechanism is defined and held fixed. These are the operations that are made available to an application. These operations become the methods in a *basic access methods* class. Access control for the basic access methods class is provided by *role* classes, one for each defined role. The methods of the role classes have the same names, types and parameters as the methods of the basic access methods class. Access control to the information accessed by the basic access methods class is located exclusively in the role classes and not in any other part of the application. The bodies of the methods in the role classes are restricted to conditionals which determine access for the role associated with that role class and/or filters which constrict the flow of information between the application interface and the basic access methods.

If access is permitted for a role, the methods of the role class then invoke the corresponding methods of the basic access methods class. If not all information obtained by the basic access methods is permitted to a role, then the parts of the information not permitted can be filtered out. Filtering may be more desirable in an application rather than generating an access violation for the entire information block.

The methods of the application interface class also have the same names, types and parameters as the methods of the *basic access methods* class. The methods of the application interface class invoke the corresponding methods of the role classes. It is the methods of an application interface object which the application invokes. Given the current role associated with the application, the methods of the application interface object select the appropriate role object.

This approach has several advantages. One advantage is that applications need not change when access conditions for roles are changed. Applications use the methods of the application interface class whose methods have the same names, types, and parameters as the methods in the basic access methods class. The methods of the application interface class and the methods of the basic access methods class are fixed and remain constant over time. When access conditions for roles change, applications fail only because of access violations. This type of failure is comparable to the failures that typically occur when information protection bits or labels are changed. Applications are normally implemented to be able to handle access violations.

Another advantage of this approach is that access conditions for roles are easily changed. Access conditions for roles are located exclusively within the role classes. Consequently, role policy changes do not require modifications to the applications themselves. One can conceive of a simple language, suitable for use by data and security administrators, for expressing access conditions restricted to conditionals and filters. A processor for such a language could generate the role objects and place them in the libraries used by applications. Most environments today support dynamically linked libraries which link when an application is loaded into memory for execution. Thus, applications do not need to be relinked when role classes are changed. This ability to easily change access conditions associated with roles permits rapid response to policy changes.

Figure 8 gives an illustration of this approach using C++. The complete source code which can be compiled and run is given in Appendix 1. In actual practice, RBAC roles, operations, and policy can be numerous and complex. In order to simplify this example, only a small subset of the roles, operations, and policy that would normally be required are illustrated.

This example has the following operations which can be performed by applications on a patient record database:

- **Get patient ID list** This operation obtains a complete list of patient names and their IDs
- **Get patient record** This operation obtains the patient record given the patient ID.

The basic access methods class *Access_PRDBO* which has methods *GetIDinfo()* and *GetPR()* for performing these operations. Also shown are the role classes associated with a patient *Pat_PRDBO* and doctor role *Doc_PRDBO*. These role classes inherit from an abstract base class *Role_PRDBO* which defines the names, types, and parameters for the methods which correspond to the methods in the basic access methods class.

The patient and doctor role classes together implement the following RBAC policy:

- Only Doctors are permitted to read the list of patient names and IDs.
- Doctors are permitted to read the records for all patients.
- Patients are only permitted to read their own record.

In order to ensure that patients only access their own records, the patient role object *Pat_PRDBO* calls a system procedure which returns the patient ID for the user.

The application interface class *PRDBO* is used by applications. When an object of this class is instantiated and a method of that object is called, that method first calls a system procedure *get_role()* which returns the user's current role. The method then calls another system procedure *get_role_obj()* which returns a pointer to the role object for that role. Finally, the method calls its corresponding method in the role object passing its input arguments to the role object method.

```
class Access_PRDBO{
      public:
            Idlist GetIdinfo();
            Patrec GetPR(Patid pid);       };

class Role_PRDBO{
      public:
            virtual Idlist GetIdinfo()=0;
            virtual Patrec GetPR(Patid patid)=0;      };

 class Pat_PRDBO:public Role_PRDBO{
      public:
            virtual Idlist GetIdinfo(){
              return("ERROR: patient cannot access patient id list\n");
            };
            virtual Patrec GetPR(Patid pid){
              if (pid == get_user_pid())
                  return(access_prdbo.GetPR(pid));
              else
                  return("ERROR: patients cannot get other's records\n");
            };     };

class Doc_PRDBO:public Role_PRDBO{
      public:
            virtual Idlist GetIdinfo(){
              return(access_prdbo.GetIdinfo());
            };
            virtual Patrec GetPR(Patid pid){
              return(access_prdbo.GetPR(pid));
            };    };
```

Figure 8. Source Code for RBAC Example

```
class PRDBO{
	public:
		Idlist GetIdinfo(){
		  char * role_name;
		  Role_PRDBO *roleobj;
		  role_name = get_role();
		  roleobj = get_role_obj(role_name);
		  if (roleobj == (Role_PRDBO *)NULL)
				return("ERROR: no such role\n");
		  return(roleobj->GetIdinfo());
		};
		Patrec GetPR(Patid patid){
		  char * role_name;
		  Role_PRDBO *roleobj;
		  role_name = get_role();
		  roleobj = get_role_obj(role_name);
		  if (roleobj == (Role_PRDBO *)NULL)
				return("ERROR: no such role\n");
		  return(roleobj->GetPR(patid));
		}; };

Role_PRDBO *get_role_obj(char *role_name){
	struct{
	char role_name[ROLE_NAME_LENGTH];
	Role_PRDBO *role_object;
	} role_tab[NUMBER_OF_ROLES] =
		{
		 {"patient", &pat_prdbo},
		 {"doctor", &doc_prdbo}
		};
	for(int i=0; i<NUMBER_OF_ROLES; i++)
		if (strcmp(role_name, role_tab[i].role_name) == 0)
			return(role_tab[i].role_object);
	return((Role_PRDBO *) NULL);
};
```

Source Code for RBAC Example(continued)

4. Demonstration Applications

The demonstration illustrating the capabilities of each technology studied in the project consists of a distributed application for clinical and administrative patient data access. For this demo, a patient record data base object (PRDBO) is defined. This object provides a consistent view of the patient information. The concept is to access patient data through this object whose methods provide a consistent specification for accessing the data. How the data is actually stored is independent of how the object client accesses the data. The methods in the object implementation access the data however and wherever the data is actually stored. CORBA is being used as a means of implementing the PRDBO.

The PRDBO organizes patient information into groups. Figure 9 shows the information groups of the PRDBO and how they relate to each other. The Identification Information Group contains information like name, address, and patient ID. The Demographic Information Group contains information like birth date and sex. The Encounter Information Group contains information like encounter date, physician seen, symptoms, and diagnosis. The Encounter Notes Group contains physician notes on the encounter. The Diagnostic Data Group contains the results of diagnostic procedures (e.g., X-rays) associated with the encounter. The Data Annotations Group contains annotations to the diagnostic data, such as, notations on an X-ray highlighting abnormalities. The Diagnostic Data and Data Annotations Groups usually contain multimedia information such as images and sound.

Pieces of information within a group have a one-to-one relationship to each other. For example, within the Demographic Information Group, each patient has only one birth date and is of only one sex.

The information groups can relate to each other in either a one-to-one relationship or a one-to-N relationship. For example, the Identification Information Group and the Demographic Information Group have a one-to-one relationship. Each patient has only one group of identification and demographic information. However, for each patient, there may be several visits to a physician. Consequently, there may be several Encounter Information Groups associated with each patient.

The information groups may be thought of as elements of sets. The PRDBO is a set whose elements are information groups or sets of information groups for each patient. Each element of the PRDBO set has as elements: the Identification Information Group, the Demographic Information Group, and a set of information about each encounter.

Figure 10 shows the architecture of the distributed application. Two clients of the PRDBO are being developed. One illustrates access from within the organization that created the information. This client is being developed using Object Linking and Embedding (OLE) on the PC. The other illustrates access from outside of the organization that created the data. This client is being developed for use with World Wide Web browsers. The arrows indicate the direction of patient information flow. The OLE application is capable of both reading and writing information to the data repositories within the organization which created the information. WWW browsers are capable only of reading information and they provide access to information created within an organization to those external to that organization.

The data repositories contain data suitable for traditional relational databases and multimedia data. The PRDBO Implementation and the OLE client are capable of performing SQL queries on relational databases. Where these databases are remote, the Remote Database Access (RDA) protocol with the SQL specialization (RDA/SQL) is used.

Multimedia data is best transmitted by means of the sockets interface of the Protocol Independent Interfaces IEEE Standard (PII/sockets). The PII/sockets interface is derived from the Berkeley sockets interface and is useful for transmitting large amounts of data.

The following sections will cover each of the two demonstration projects in more detail.

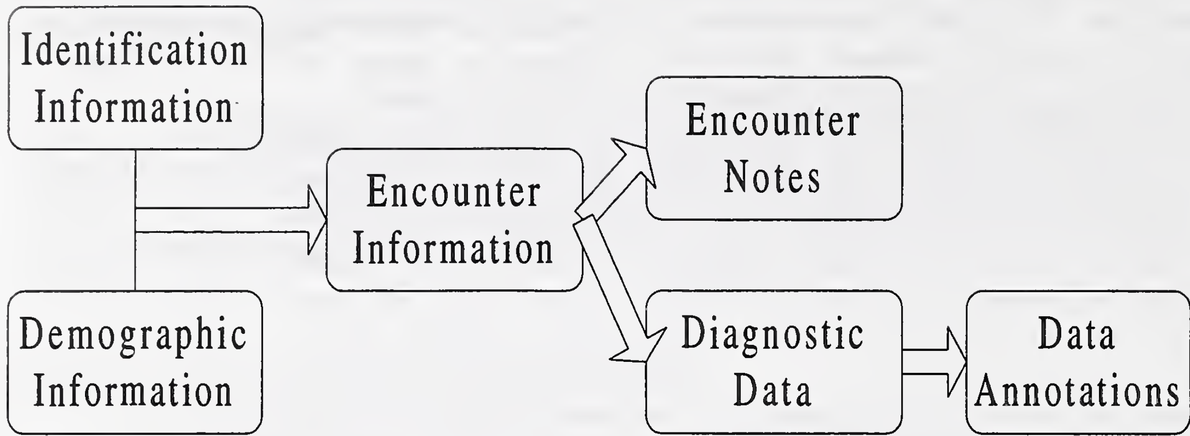

Figure 9. Patient Record Database Object Entity Relationships

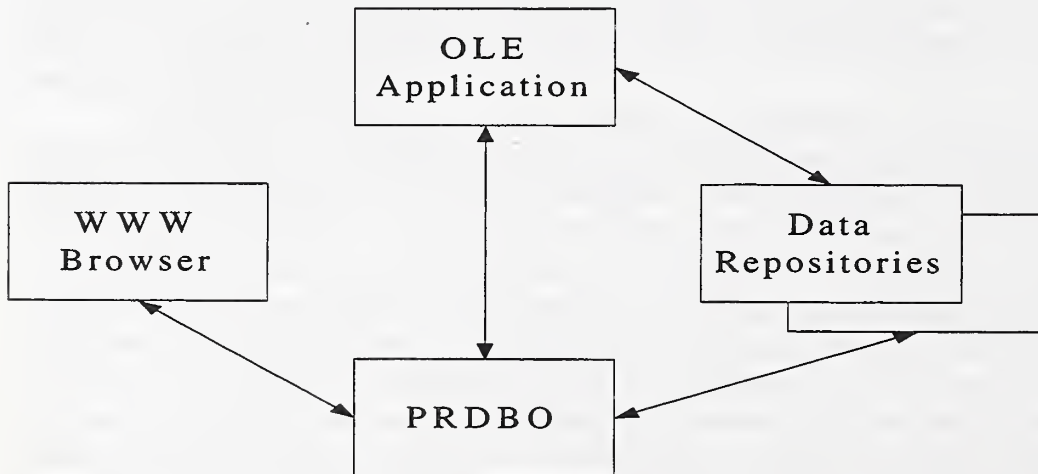

Figure 10. Block Diagram of Distributed Health Care Project

4.1 POSIX Demo

The health care demonstration system follows a distributed, client-server model. Clients send messages containing requests to server objects. The server objects verify the request, and send results back to the client in messages.

The model used for client-server communication is object-based. Clients send messages to these objects in order to retrieve data from the patient record database. From the perspective of the client, these objects are local; the client is unaware that the actual object implementation is running across the network, possibly on a different machine.

The server objects retain no state. These objects only provide an abstract interface to the patient record database. Each method call is independent, and there is no prescribed ordering to the calling sequence. The server objects encapsulate the access to the patient record database. The goal is to maintain the same interface, independent of changes in databases, systems, or networks.

Because the server objects are stateless, all access controls are enforced by each method. A different model would use the concept of sessions between the client and server objects. This can be accomplished by having the server objects retain some state, such as the user id if the requester. Each method then checks the state for the proper access being granted. This technique would require that one method be called before any others. This method would establish the access privileges based on the role of the requester. (This could be done by the object's constructor) Further method calls would assume that the access has been granted.

By using the idea of a session between client and server object, we have two choices for server objects to maintain the session. One choice is to have one server object, which maintain lists of sessions in its state data. The other choice is to have the server object dedicated to one session only. The first option would require that the server object implementation run continuously, accepting requests and establishing sessions. In the second option, the lifetime of the server object is limited to the session lifetime.

The Common Object Request Broker Architecture (CORBA) provides a means to specify the interface for a server object independent of the server's implementation. The advantages include allowing the client programs to be independent of changes in the server, and a consistent means of interfacing to servers is provided. We have used a CORBA compliant product in the development of the health care demonstration.

Client programs (such as the CGIBIN scripts for HTTP) do not communicate directly with the patient record database. The interface to the database is specified in CORBA IDL (interface definition language). A server program implements the operations specified in the IDL for the database server.

The objects developed for the server provide a wrapper to the underlying patient record database. The IDL language used is presented in Appendix B. There are two levels of wrappers provided in the demonstration project. The first level provides a basic interface to each of the database tables. This interface is named the patient_record_server_type. This interface provides methods to retrieve a piece of the patient record, as implemented by the underlying database. Each method provides controls on the access to the information based on the role of the requester.

Methods *GetIdRecordList*, *GetAdministrativeList*, *GetEncounterList*, *GetEncounterNotesList*, *GetDiagnosticList*, and *GetAnnotationList* perform query-by-example. An input parameter, of the same type as the objects returned in the list, contains the criteria for performing the query on the patient record database. For example, for method *GetIdRecordList*, to search on last name of "Smith," the criteria's record must have last_name set to "Smith."

Table 1 provides a description of the methods provided by the patient record server object.

The higher-level interfaces provided are patient_role_server_type and doctor_role_server_type. These interfaces provide the methods unique to their respective roles. For example, the patient role server provides a method GetPatientRecord that retrieves the entire patient record for a given patient id. Both the patient and doctor servers rely on the lower-level patient record server interface to implement the operations. Table 2 gives the description of the patient server, and Table 3 gives the description for the doctor server.

Method Name	Description
GetAccessInfo	Returns access control information for a user name if access is verified for given name and password
GetIdInfo	Returns a patient ID record for a specific patient ID
GetIdRecordList	Returns a list of patient ID records
GetAdministrativeList	Returns a list of administrative records
GetEncounterList	Returns a list of encounter records
GetEncounterNot	Returns a list of encounter notes
GetDiagnosticList	Returns a list of diagnostic records
GetAnnotationList	Returns a list of annotations

Table 1. Methods of the patient record server objectesList

Method Name	Description
GetPatientRecord	Returns the entire patient record for a given patient ID; the requester's role must be patient

Table 2. Methods for patient role server object

Method Name	Description
GetIdRecords	Returns a list of ID records for all patients
GetPatientRecord	Returns the entire patient record for a given patient ID

Table 3. Methods for doctor role server object

4.1.1 Operation of the server object methods

Each of the server methods has several responsibilities. The access controls are described below. Besides access control, the server methods must communicate with the patient record database. This communication is done via the Remote Database Access (RDA) architecture. The use of RDA allows for the server objects to connect to the database, no matter on what machine the database is located.

We now have two levels of independence in the demonstration project. The server objects do not depend on the location of the database, and the database access procedures. The interface to the database handles all necessary connections, such as locating the database on the network, and supplying commands and returning data.

The other level of independence is client to server. The server objects can be running on any machine on the network (presuming the existence of an object request broker). Client programs send messages to the server objects, and the underlying object request broker (ORB) delivers the messages to the appropriate server implementation. Furthermore, if there is no server implementation running, the ORB can be directed to start one. To the client, the only visible interface is that of method calls. There is no need for the client to know the machine name, port numbers, or other communication level information. In this way, the methods are more than remote procedure calls, and provide a higher abstraction for the clients.

4.1.2 Role-Based Access Control in the Server

The assumption made by the server programs is that the clients are unable to do any form of role-based access control. Therefore, all access control is done by the servers. Access control is enforced by each method in the patient record server object. The user name, role, and password (encrypted) is sent to each method, along with the request for data. The user name/password combination is verified in the method. If the access check fails, no data is returned to the caller, and an error message is written into the access control block, which is returned.

If the access check is successful, the query on the database is performed. Before data is returned, any data items that are not accessible to the requester are eliminated from the data. This control allows pieces of the database tables to be accessed, while other pieces are suppressed. The server method uses a table-driven approach to remove any data not accessible because of the requester's role.

In the case of a patient requesting the entire patient record, the patient role server performs another form of verification. The request to the patient record server is made using the patient id as the criteria. The actual value of the id is retrieved from the verification database when the user-name/password check is made. The id value passed by the requester is not trusted, and is therefore not used.

The UK policy was incorporated into the POSIX demonstration project. At this point in the development of the demo, the UK policy has been somewhat simplified by eliminating the labeling and limiting the number of roles. We have defined seven roles for the health care demonstration. Table 4 gives the roles and the accessible data for each. In order to change the

roles, or change the access for a role, all that is needed is to change a table that the server methods use to control access. The operation of the methods is to replace any inaccessible data with blank strings before returning the data.

Role ID	Access Extent
Patient	All information for the patient
Doctor	All information
Voluntary Caring Agency	Name, address, clinical data
Researcher	Age, sex, clinical data
Epidemiologist	Age, sex, clinical data
Environmental Health Officer	Name, ID, address
Organization Staff	Name and ID

Table 4. Roles and Access Extents

Every requester, on entry to the "Patient Record Health Care Demonstration Project," must first be identified and take on a role. This is accomplished by requiring the requester to specify a name, password, and to select a role for the entire session. The information provided is verified before access to any patient records is allowed. If the information provided is found to be in error, the requester may correct and resubmit the information.

Once a requester is verified, the ID type associated with the specific requester is provided to all of the access requests initiated along with the requester's name, encoded requester's password, and role.

4.2 The PC Demo

The PC component of the demonstration was built on OLE. Since distributed OLE is not yet available the PC demonstration emphasized the use of OLE in interacting with other applications.

The program manipulates patient record files. These are simple OLE container documents that have no contents except other OLE objects. The user can insert objects by one of two ways. The first way is the usual method of selecting insert object from the menu. The dialog box is a standard system dialog that allows either embedded or linked objects to be added. The second way to insert an object is to query the database. Selecting query from the menu of the main window brings up a dialog box with buttons for queries on identification number or diagnostic information. The dialog also tracks the currently selected patient identification number. Each button brings up a dialog box specific for each query.

The dialog boxes have editable textboxes for the fields of the database. The queries are performed by simple QBE (Query By Example) forms. The buttons at the bottom of the dialog boxes allow the user to perform a query on the database, scroll through the database records in the current view or select the current record. Selecting a record on the Id dialog makes the id of the current record become the currently selected patient identification number on the main dialog. This number will be automatically entered in the text field for id on the Diagnostic Query box. Pushing select on the Diagnostic Query box will embed the file associated with that record into the main view. The button controls, text fields and several invisible controls are all VBX controls. The database communication is performed using the ODBC protocol.

A block diagram of the components of the PC demonstration is shown in figure 8. The program was compiled with Borland C++ 4.51 using vendor and third party libraries, targeted for Windows 3.1/3.11. OCF (Object Component Frameworks) is an object oriented encapsulation of the OLE library. OWL (Object Windows Library) is an object oriented GUI (Graphical User Interface) library for Microsoft Windows. The VBX control interfaces with a local database over a ODBC connection.

4.2.1 OLE Objects Used in the Viewer

The viewer was developed using the Multiple Document Interface (MDI). MDI is a standard method that allows an application to edit several documents of the same type simultaneously. Each document has its own main window that is enclosed by the main window of the application. Each document window can be minimized or have its size changed independently. The viewer also uses the Doc/View model where the contents of the document are separated from the way the contents are displayed. This makes the display of embedded OLE objects easier. The document controls the storage of the patient record as an OLE compound document. The document is implemented as the C++ class *HealthDocument*. The display of the data is done by the class *HealthView* that encapsulates the view. An external data record from the database is inserted using the code in figure 9. All redrawing of the window and other GUI activities are handled by the OWL and OCF library code.

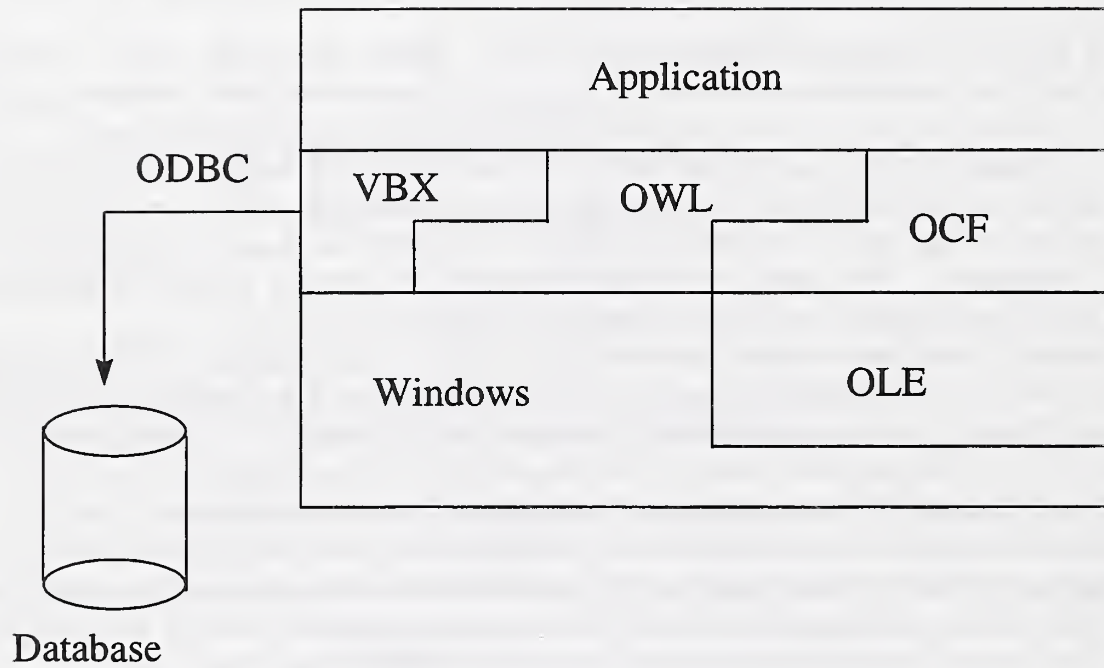

Figure 11. PC Demonstration Application Block Diagram

```
void HealthView::insert_file( const char* fname) {
 static int x = 0;
 static int y = 0;

 // OcView is an object of type TOcView which manages presentation of compound documents
 // iwFile means take the data from a file; ihLink means link rather than embedded
 TOcInitInfo initInfo( ihLink, iwFile, OcView);

 // space out the inserted objects
 TRect rect( 10 + x * 100, 10 + y * 100, (x + 1) * 100, (y + 1) * 100);
 if( x == 5) { x = 0; ++y; }
 else ++x;

 initInfo.HIcon = 0;                                     // no icon
 initInfo.Path = (LPOLESTR) fname;                       // file where the data is

 // create the new part for the document and make it the currently selected item
 SetSelection( new TOcPart(*GetOcDoc(), initInfo, rect));

 // tell the view something has be added and redraw the view
 OcView->Rename();
 InvalidatePart(invView);
}
```

Figure 12. Source Code to link OLE COM Object

4.3 Other Distributed Communication Methods

Sockets were not implemented in the demo projects. A possible use was determined for them. Sockets are most useful for bulk communication while CORBA does poorly. The best use would be for large transfers, such as bitmaps, to be performed by connection oriented sockets. CORBA would be used to transmit the location of the bitmap to the CORBA client. The CORBA client would then make a connection to a file server on the host with the bitmap and retrieve the bitmap to the local machine. That way the CORBA layer is not responsible for transmission of the bitmap data itself.

RPC was not implemented in the demo. CORBA used RPC as a transport mechanism and supersedes it.

5. Conclusions

In this report we have examined five distributed communication mechanisms. Table 5 presents our conclusions in tabular form. Each of the transport methods have different advantages and are best suited for different situations. Some are very specialized and are only good for certain tasks. SQL/RDA is an example of this type of method. It was designed to allow queries of remote databases. It does this task well but it does not support general communication. OLE does support communication between application, but best support is only provided in the Microsoft Windows environment. By limiting the environment this way, OLE can provide integration at the level of the user interface. Distributed COM will most likely be at the same level as CORBA.

The three general mechanisms trade efficiency for ease of use. The lowest level is the socket interface, which is very fast and efficient, but must be programmed at the lowest level. RPC is a higher level transport that also supplies a programming language independent IDL. The highest support is supplied by CORBA. A CORBA user can ask for an object and let the CORBA implementation worry about where the object can be found. The tradeoff is that communication using CORBA is inefficient.

In the demonstration viewers we attempted to show a way to use the various mechanisms to their best advantage. By combining the transport methods, we were able to get support when needed and efficiency when that was needed. Using each of the distributed transport mechanisms where it was most effective allowed us to achieve the best overall performance for the entire system. This is more complicated in terms of program development, but the payoffs can be quite high.

	CORBA	OLE	SQL/RDA	sockets	RPC
ease of use	higher level Object Oriented API; Interface Definition Language	low level API encapsulating Object Oriented frameworks exist; Interface Definition Language	only requires knowledge of SQL	low level API large amount of work to use, but very flexible	higher level Interface Definition Language
language bindings	C, C++, Smalltalk, Ada	C, C++, Visual Basic, various product control languages	ASCII text using SQL	C	C
class of application	distributed object systems	Microsoft Windows application integration	distributed database operations	high performance bulk transport	distributed client/server
security	under development	based on security available at the file level	Access Control Lists at item level; authentication strings	none (some research efforts)	kerebos secure RPC
protocols supported	TCP/IP; work on Interoperability	not yet distributed	TCP/IP	TCP/IP	TCP/IP
performance of transport service	slow, still under development	N/A	fast	fastest	slow

Table 5. Project Conclusions

6. Appendix A - Code for Role-based Access Control using Object Technology

```
//
//                  C++ Example of Role Based Access Control
//                   Implementation Using Object Technology
//
//                              John Barkley
//                       (barkley@sst.ncsl.nist.gov)
//
//     This C++ program illustrates the implementation of RBAC using Object
//     Technology. In this example, there are two methods which provide
//     a healthcare application with access to patient records:
//          GetIdinfo() - provides a list of patient names and their IDs
//          GetPR(pid) - given a patient ID, returns the patient record
//     These two methods are associated with several classes:
//          Access_PRDBO - this class provides the basic access methods
//                    to patient information
//          Role_PRDBO - the abstract role class
//          Pat_PRDBO - the patient role class
//          Doc_PRDBO - the doctor role class
//          PRDBO - the application programming interface class
//
//     For each role, there is a role class for that role derived from the
//     abstract role class. The methods in each role class contain the
//     conditions under which a user in that role may perform the
//     corresponding methods in a basic access methods class (Access_PRDBO in
//     this example). The methods in the basic access methods class perform
//     actions on the information. The methods in each role class are invoked
//     by corresponding methods in the application programming interface
//     class (PRDBO in this example).
//
```

```
//        This approach permits much of the generality of the RBAC concept of
//        "action" to be realized, i.e., once the basic actions on information
//        have been established, any conditions permitting actions on information
//        specified in an RBAC policy may be implemented. In addition,
//        this approach permits roles to be created, removed, and modified
//        without having to recompile either the application or the basic access
//        methods class. When a role is added, removed, or modified in the
//        policy, a role class is added, removed, or modified.
//
//        This example was compiled using the GNU C++ compiler.
//

#include <stdio.h>
#include <iostream.h>
#include <strstream.h>

const int ROLE_NAME_LENGTH = 50;
const int NUMBER_OF_ROLES = 2;

typedef char *Idlist;
typedef char *Patrec;
typedef int Patid;

extern char * get_role();
extern int get_user_pid();
extern "C" void exit(int);

//         basic access methods class
class Access_PRDBO{
        public:
          Idlist GetIdinfo(){ return("Here's the list of patients and their IDs\n"); };
          Patrec GetPR(Patid pid){
                  const int BUFLEN = 128;
                  static char buf[BUFLEN];
                  static ostrstream oss(buf, BUFLEN, ios::out);

                  oss.seekp(ios::beg);
                  oss << "Here's the patient record for patient ID: "
                        << pid << endl << ends;
                  return(buf);
                }; };
Access_PRDBO access_prdbo;
```

```
//      role classes:
//        one for each role derived from the abstract class Role_PRDBO
class Role_PRDBO{
        public:
                virtual Idlist GetIdinfo()=0;
                virtual Patrec GetPR(Patid patid)=0;
};


class Pat_PRDBO:public Role_PRDBO{
        public:
                //      the policy does not permit patients to access
                //        the list of patient names and their IDs

          virtual Idlist GetIdinfo(){
                return("ERROR: patient cannot access patient id list\n");
                };
                //      the policy only permits a patient to have access
                //        to his own patient information

          virtual Patrec GetPR(Patid pid){
                if (pid == get_user_pid())
                        return(access_prdbo.GetPR(pid));
                else
                        return("ERROR: patients cannot get other's records\n");
          };
};
static Pat_PRDBO pat_prdbo;
```

```
class Doc_PRDBO:public Role_PRDBO{
        public:
                //      the policy permits doctors to have access
                //        to all information on any patient

            virtual Idlist GetIdinfo(){
                  return(access_prdbo.GetIdinfo());
                };
            virtual Patrec GetPR(Patid pid){
                  return(access_prdbo.GetPR(pid));
                };
};
static Doc_PRDBO doc_prdbo;

//      this procedure, which must be changed when roles are added or deleted,
//      would be a system call which finds the the role object given the
//      user's role

Role_PRDBO *get_role_obj(char *role_name){
        struct{
        char role_name[ROLE_NAME_LENGTH];
        Role_PRDBO *role_object;
        } role_tab[NUMBER_OF_ROLES] =
                {
                 {"patient", &pat_prdbo},
                 {"doctor", &doc_prdbo}
                };
        for(int i=0; i<NUMBER_OF_ROLES; i++)
                if (strcmp(role_name, role_tab[i].role_name) == 0)
                        return(role_tab[i].role_object);
        return((Role_PRDBO *) NULL);
};
```

```
//          application interface class
class PRDBO{
        public:
                Idlist GetIdinfo(){
                  char * role_name;
                  Role_PRDBO *roleobj;
                  role_name = get_role();
                  roleobj = get_role_obj(role_name);
                  if (roleobj == (Role_PRDBO *)NULL)
                                        return("ERROR: no such role\n");
            return(roleobj->GetIdinfo());
                };
                Patrec GetPR(Patid patid){
            char * role_name;
                  Role_PRDBO *roleobj;
                  role_name = get_role();
                  roleobj = get_role_obj(role_name);
                  if (roleobj == (Role_PRDBO *)NULL)
                                        return("ERROR: no such role\n");
            return(roleobj->GetPR(patid));
                };
};
PRDBO prdbo;

//        this procedure would be a system call to return the user's current role

char * get_role(){
      static char role_name[ROLE_NAME_LENGTH];
         cout << "Enter role name: ";
         cin >> role_name;
         return(role_name);
};

//        this procedure would be a system call to return the user's patient ID

int get_user_pid(){
      int pid;
         cout << "Enter user's patient id: ";
         cin >> pid;
         return(pid);
};
```

```
main(){
	char opt;
	Patid pid;
	while(1){
		cout << "Enter i-GetIdlist, r-GetPR: " ;
		cin >> opt;
		if ( !cin ) {cout << endl; exit(0); };
		switch (opt) {
			case 'i' : cout << prdbo.GetIdinfo() << endl;
					break;
			case 'r' : cout << "Enter patient id: ";
				   cin >> pid;
				   cout << prdbo.GetPR(pid) << endl;
					break;
		};
	};
};
```

7. Appendix B - IDL Description of Patient Record Object

```
//
// hc_types.idl
//
// IDL declarations for the patient record structures
//
// First, define the basic data types used in terms of CORBA types
//
typedef string name_type;
typedef string login_type;
typedef string<9> id_type;
typedef string<2> state_type;
typedef string<10> zipcode_type;
typedef string complaint_type;       // string of keywords
typedef string symptom_type;         // string of keywords
typedef string<255> URL_type;        // string to hold Universal Resource Locator
typedef string treatment_type;       // unstructured text for storing treatment information
typedef char sex_type;               // Male, Female
typedef string role_type;            // type for storing the role identifier
typedef string<10> password_type;    // type for storing passwords
typedef string<10> date_type;        // date in mm/dd/yyyy format
typedef string<16> date_time_type;   // date and time in "mm/dd/yyyy hh:mm" format
typedef string name_address_type;    // used for storing name and address of companies, etc.
typedef string<3> time_zone_type;    // type for time zone indicator
//
```

```
// Next, define the compund data types in terms of the basic data
// types.  These definitions form the basic patient record information
// structure.
//
struct address_type {
  string street;
  string city;
  state_type state;
  zipcode_type zipcode;
};
enum date_data_format {SINGLE, RANGE};
struct date_data_type
{
  date_data_format format;             // set to SINGLE or RANGE to indicate
                                       // whether a single date or a date range
  time_zone_type time_zone;            // stores the time zone indicator
  date_type begin_date;
  date_type end_date;
};
struct date_time_data_type
{
  date_data_format format;             // set to SINGLE or RANGE to indicate
                                       // whether a single date or a date range
  time_zone_type time_zone;            // stores the time zone indicator
  date_time_type begin_date_time;
  date_time_type end_date_time;
};
struct patient_record_id_type {
  id_type patient_id;
  name_type last_name;
  name_type middle_name;
  name_type first_name;
  address_type address;
};
struct patient_record_administrative_type {
  id_type patient_id;
  sex_type sex;
  date_data_type date_of_birth;
  date_data_type date_of_death;
};
```

```
struct patient_record_encounter_type {
  id_type patient_id;
  date_time_data_type encounter_date;
  complaint_type complaint;
  symptom_type symptoms;
  id_type doctor_id;
  treatment_type treatment;
};
struct patient_record_encounter_notes_type {
  id_type patient_id;
  date_time_data_type encounter_date;
  date_time_data_type notes_date;
  id_type doctor_id;
  URL_type notes_URL;                  // doctor notes are stored in a file and the
                                       // URL to the file is given to the client

};
struct patient_record_diagnostic_type {
  id_type patient_id;
  date_time_data_type encounter_date;
  date_time_data_type diagnostic_date;
  name_address_type diagnostic_center;
  URL_type diagnostic_URL;             // diagnostic data is stored in a file and the
                                       // URL to the file is given to the client

};
struct patient_record_annotation_type {
  id_type patient_id;
  date_time_data_type encounter_date;
  date_time_data_type diagnostic_date;
  date_time_data_type annotation_date;
  id_type doctor_id;
  URL_type annotation_URL;             // annotation data is stored in a file and the
                                       // URL to the file is given to the client

};
//
```

```
// Next, define a record type to hold the information required
// for access control on the server.  A record of this type
// is passed to each server method.  The methods verify the
// role access before providing data.
//
struct access_control_information_type {
  id_type requestor_id;
  login_type requestor_login;
  role_type requestor_role;
  password_type requestor_password;
  string access_result;                    // contains string which indicates result of access
                                           // verification when there is an error
                                           // this is null when no error.
};
//
// Next, define some sequences which are used to store the lists
// of objects being returned by the server
//
typedef sequence<id_type> patient_id_list_type;
                    // a list of patient identifiers;
typedef sequence<patient_record_id_type> patient_record_id_list_type;
                    // a list of patient id records
typedef sequence<patient_record_administrative_type>
    patient_record_administrative_list_type;
                    // a list of administrative records
typedef sequence<patient_record_encounter_type> patient_encounter_list_type;
                    // a list of patient encounters
typedef sequence<patient_record_encounter_notes_type>
   patient_encounter_notes_list_type;
                    // a list of patient encounter notes
typedef sequence<patient_record_diagnostic_type> patient_diagnostic_list_type;
                    // a list of diagnostic data records
typedef sequence<patient_record_annotation_type> patient_record_annotation_list_type;
                    // a list of annotation data records
```

```
//
//
// Define the structure to hold the complete patient record
//
struct patient_record_type {
  patient_record_id_type id_record;
  patient_record_administrative_type admin_record;
  patient_encounter_list_type encounter_list;
                              // a list of patient encounters
  patient_encounter_notes_list_type encounter_notes_list;
                              // a list of patient encounter notes
  patient_diagnostic_list_type diagnostic_list;
                              // a list of diagnostic data records
  patient_record_annotation_list_type annotation_list;
                              // a list of annotation data records
};
```

```
//
// hc.idl
//
#include "hc_types.idl"
// IDL declarations for interface to patient record server objects
//
interface patient_record_server_type {
                                    // returns an entire patient record
  patient_record_type GetPatientRecord(inout access_control_information_type
                role_access_info, in id_type patient_id);

                                    // returns a list of patient id records that have matches to the
                                    // criteria specified in the input parameter
  patient_record_id_list_type GetIdRecordList
    (in patient_record_id_type id_criteria,
     inout access_control_information_type role_access_info);

                                    // return a list of administrative records based on matching fields
                                    // in the criteria record
  patient_record_administrative_list_type GetAdministrativeList
      (in patient_record_administrative_type admin_criteria,
       inout access_control_information_type role_access_info);

                                    // returns a list of encounters based on matching fields
                                    // in the criteria record
  patient_encounter_list_type GetEncounterList
      (in patient_record_encounter_type encounter_criteria,
       inout access_control_information_type role_access_info);
```

```
                    // returns a list of encounter notes based on matching fields
                    // in the criteria record
patient_encounter_notes_list_type GetEncounterNotesList
  (in patient_record_encounter_notes_type encounter_notes_criteria,
   inout access_control_information_type role_access_info);

                    // returns a list of diagnostic records based on matching fields
                    // in the criteria record
patient_diagnostic_list_type GetDiagnosticList
  (in patient_record_diagnostic_type diagnostic_criteria,
   inout access_control_information_type role_access_info);

                    // returns a list of annotation records based on matching fields
                    // in the criteria record
patient_record_annotation_list_type GetAnnotationList
  (in patient_record_annotation_type annotation_criteria,
   inout access_control_information_type role_access_info);
};
```

8. Glossary

access control policy - Specifies which users are or are not entitled to an application's services.
authentication - Clients and servers capable of proving their identities to each other.
authorization - Means of access to information managed based on identity of the user.
automation - OLE mechanism to allow an application to control another application directly.
binding - The act of associating a server with a socket. Logical association between a client and a server.
bridging - A mapping between two domains.
broker or binding service - An intermediary between clients and servers designated to assist in network resource communications.
client application - A user-written program that performs function calls to be executed by a server application.
client process - A process that executes the client application.
client/server model - Processing environment where one set of entities requests work to be done and another set actually performs the work.
client stub - Code module that is generated by the special Newark interface compiler. A client stub provides:

- filters to encode and marshal the IN arguments
- a call to the requested server
- filters to unmarshal and decode the reply OUT arguments
- time-out of the operation if needed

CORBA2/Interoperable - CORBA2/Core and the CORBA2/Internet IOP
domain - A set of objects sharing a common characteristic or abiding by common rules.
embedding - OLE technique that allows a container document to completely hold another COM object.
interface - The collection of remote procedures that a client and server share.
linking - OLE technique that allows a container document to hold a reference to another COM object. The object itself is extenal to the container document.
marshaling - Packaging the input parameters and sending them to the remote process.
NIS - Stores network information on servers and provides it to any workstation that asks for it.
OSF - A consortium of DEC, IBM, HP/Apollo, and other major UNIX hardware vendors.
port - a logical network communication channel.
server application - A user-written program that handles and replies to request(s) from a client application.
server process - A process that executes the server application.
server stub - Code module that is generated by the special network interface compiler. A server stub provides:

- registration of the service with the proper lookup utility
- filters to unmarshal and decode the IN arguments

- invocation of service routine requested
- filters to encode and marshal the OUT arguments
- results back to client

unmarshaling - Unpacking the input parameters and calling the requested procedure using the unpacked arguments.

9. References

BOOCH94

G. Booch, Object-Oriented Analysis and Design with Applications, Benjamin/Cummings Publishing Company, Inc., 1994.

CHINITZ94

J. Chinitz, "It's Not Your Father's RPC", SunExpert, June 1994.

CORBAIIOP

OMG TC Document 95.3.xx [REVISED 1.8 jm], CORBA 2.0/Interoperability Universal Networked Objects, March 20, 1995.

CORBASPEC

OMG Document Number 93.12.43, Revision 1.2, The Common Object Request Broker: Architecture and Specification, Draft 29 December 1993.

CORBATOUR

OMG, "A Tour of CORBA", ftp://omg.org/pub/presentations/corba.ps.

GRIEW

A. Griew, "A Strategy for Security of the Clinical Record and its Transfer", Institute for Health Informatics, Aberystwyth, DRAFT.

IEEE1003.1c

IEEE Std 1003.1c-1995, "Portable Operating System Interface for Computer Environments (POSIX) - Part 1: System Application Program Interface (API) [C language]" , The Institute of Electrical and Electronics Engineers, Inc., June, 1990.

IEEE1003.1g

IEEE Std 1003.1g-1995, "Information Technology - Portable Operating System Interface (POSIX) - part xx: Protocol Independent Interfaces (PII)", The Institute of Electrical and Electronics Engineers, Inc., December, 1995.

IEEE1003.6

IEEE Std 1003.6.1, "Draft Standard for Information Technology - Portable System Interface (POSIX) - Part 1: System Application Program Interface (API) - Protection, Audit and Control Interfaces [C language]" , The Institute of Electrical and Electronics Engineers, Inc., Novemember, 1992.

IEEE610

ANSI/IEEE Std 610.12-1990, "Glossary of Software Engineering Terminology", The Institute of Electrical and Electronics Engineers, Inc., February, 1991.

ISO9075-3

ISO/IEC 9075-2:1995, "Database Language SQL - Part 3: Call Level Interface (SQL/CLI)", International Organization for Standardization, 1995.

ISO9579-1

ISO/IEC 9579-1:1993, "Information Technology - Open Systems Interconnection - Remote Database Access - Part 1: Generic Model, Service and Protocol",. International Organization for Standardization, 1993.

ISO9579-2

ISO/IEC 9579-2:1993, "Information Technology - Open Systems Interconnection - Remote Database Access - Part 2: SQL specialization",. International Organization for Standardization, 1993.

RBAC

"An Introduction to Role-Based Access Control", CSL Bulletin, National Institute of Standards and Technology, December, 1995.

ROSENBERRY

W. Rosenberry, D. Kennedy and G. Fisher, Understanding DCE, O'Reilly & Associates, Inc, October, 1992.

SSL

"THE SSL Protocol"; Internet Draft (Working Document), unpublished document located at http://home.netscape.com/newsref/std/SSL.html.

SUNADMIN

SunOS 5.3 Administering NIS+ and DNS, SunSoft.

SUNNET

SunOS 5.3 Network Interfaces Programmer's Guide, SunSoft.

VINOSKI93

S. Vinoski, "Distributed Object Computing With CORBA", C++ Report, July/August 1993.

VINOSKI95

S. Vinoski, "Object Connections", C++ Report, September 1995.

A Note on WWW References:

As the World Wide Web (WWW) gains in size, it has to be considered as an increasingly important mechanism for research. This is especially true since the turnaround time on the WWW is faster than it is for printed material